

Contents

Get Started

[What is Xamarin.Forms](#)

Installation

[Installing Xamarin on Windows](#)

[Installing Xamarin Previews \(Windows\)](#)

[Uninstalling Xamarin from Visual Studio](#)

[Installing Visual Studio for Mac](#)

[Installing Xamarin Previews \(Mac\)](#)

[Uninstalling Visual Studio for Mac](#)

[Xamarin Firewall Configuration Instructions](#)

Supported platforms

First App

Quickstarts

[File > New](#)

[Multipage](#)

[Database](#)

[Styling](#)

[Deep dive](#)

Tutorials

[Stack Layout](#)

[Label](#)

[Button](#)

[Text entry](#)

[Text editor](#)

[Images](#)

[Grid Layout](#)

[Lists](#)

[Pop-ups](#)

[App lifecycle](#)

- [Local database](#)
 - [Web services](#)
 - [Learn about Xamarin](#)
 - [.NET developers](#)
 - [Java developers](#)
 - [Objective-C developers](#)
 - [Azure](#)
 - [Development guidance](#)
- ## XAML
- [Overview](#)
 - [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 Controls](#)
 - [XAML Compilation](#)
 - [XAML Markup Extensions](#)
 - [Consuming XAML Markup Extensions](#)
 - [Creating XAML Markup Extensions](#)
- ## Tooling
- [XAML Hot Reload](#)
 - [Live Visual Tree](#)
 - [XAML Toolbox](#)
 - [XAML Previewer](#)
 - [Design-time data](#)
 - [Custom controls](#)
- ## Namespaces
- [XAML Namespaces](#)
 - [XAML Custom Namespace Schemas](#)
 - [XAML Namespace Recommended Prefixes](#)

[Additional Capabilities](#)

[Bindable Properties](#)

[Attached Properties](#)

[Resource Dictionaries](#)

[Passing Arguments](#)

[Generics](#)

[Field Modifiers](#)

[Loading XAML at Runtime](#)

[Xamarin.Forms XAML on Q&A](#)

[Application Fundamentals](#)

[Overview](#)

[Accessibility](#)

[Automation Properties](#)

[Keyboard Accessibility](#)

[App Class](#)

[App Lifecycle](#)

[Application Indexing and Deep Linking](#)

[Behaviors](#)

[Introduction](#)

[Attached Behaviors](#)

[Xamarin.Forms Behaviors](#)

[Reusable EffectBehavior](#)

[Custom Renderers](#)

[Introduction](#)

[Renderer Base Classes and Native Controls](#)

[Customizing an Entry](#)

[Customizing a ContentPage](#)

[Customizing a Map Pin](#)

[Customizing a ListView](#)

[Customizing a ViewCell](#)

[Customizing a WebView](#)

[Implementing a View](#)

Data Binding

[Basic Bindings](#)

[Binding Mode](#)

[String Formatting](#)

[Binding Path](#)

[Binding Value Converters](#)

[Relative Bindings](#)

[Binding Fallbacks](#)

[Multi-Bindings](#)

[The Command Interface](#)

[Compiled Bindings](#)

DependencyService

[Introduction](#)

[Registration and Resolution](#)

[Picking from the Photo Library](#)

Dual-screen

[Get started](#)

[Dual-screen patterns](#)

[TwoPaneView layout](#)

[DualScreenInfo helper class](#)

[Dual-screen triggers](#)

Effects

[Introduction](#)

[Effect Creation](#)

[Passing Parameters](#)

[Parameters as CLR Properties](#)

[Parameters as Attached Properties](#)

[Invoking Events](#)

[Reusable RoundEffect](#)

Gestures

[Tap](#)

[Pinch](#)

- [Pan](#)
- [Swipe](#)
- [Drag and Drop](#)
- [Local Notifications](#)
- [Localization](#)
 - [String and Image Localization](#)
 - [Right-to-Left Localization](#)
- [MessagingCenter](#)
- [Navigation](#)
 - [Hierarchical Navigation](#)
 - [TabbedPage](#)
 - [CarouselPage](#)
 - [FlyoutPage](#)
 - [Modal Pages](#)
- [Shell](#)
 - [Introduction](#)
 - [Create a Shell application](#)
 - [Flyout](#)
 - [Tabs](#)
 - [Pages](#)
 - [Navigation](#)
 - [Search](#)
 - [Lifecycle](#)
 - [Custom Renderers](#)
- [Templates](#)
 - [Overview](#)
 - [Control Templates](#)
 - [Data Templates](#)
 - [Introduction](#)
 - [Data Template Creation](#)
 - [Data Template Selection](#)
- [Triggers](#)

User Interface

[Overview](#)

[Controls reference](#)

[Overview](#)

[Pages](#)

[Layouts](#)

[Views](#)

[Cells](#)

[Common properties, methods, and events](#)

[Third-party controls](#)

[Present data](#)

[BoxView](#)

[Image](#)

[Label](#)

[Map](#)

[Overview](#)

[Initialization and Configuration](#)

[Map Control](#)

[Position and Distance](#)

[Pins](#)

[Polygons, Polylines, and Circles](#)

[Geocoding](#)

[Launch the Native Map App](#)

[Shapes](#)

[Overview](#)

[Ellipse](#)

[Fill rules](#)

[Geometries](#)

[Line](#)

[Paths](#)

[Path](#)

[Path markup syntax](#)

[Path transforms](#)

[Polygon](#)

[Polyline](#)

[Rectangle](#)

[WebView](#)

[Initiate commands](#)

[Button](#)

[ImageButton](#)

[RadioButton](#)

[RefreshView](#)

[SearchBar](#)

[SwipeView](#)

[Set values](#)

[CheckBox](#)

[DatePicker](#)

[Slider](#)

[Stepper](#)

[Switch](#)

[TimePicker](#)

[Edit text](#)

[Editor](#)

[Entry](#)

[Indicate activity](#)

[ActivityIndicator](#)

[ProgressBar](#)

[Display collections](#)

[CarouselView](#)

[Introduction](#)

[Data](#)

[Layout](#)

[Interaction](#)

[EmptyView](#)

- [Scrolling](#)
- [CollectionView](#)
- [Introduction](#)
- [Data](#)
- [Layout](#)
- [Selection](#)
- [EmptyView](#)
- [Scrolling](#)
- [Grouping](#)
- [IndicatorView](#)
- [ListView](#)
 - [Data Sources](#)
 - [Cell Appearance](#)
 - [List Appearance](#)
 - [Interactivity](#)
 - [Performance](#)
- [Picker](#)
 - [Setting a Picker's ItemsSource Property](#)
 - [Adding Data to a Picker's Items Collection](#)
- [TableView](#)
- [Additional controls](#)
 - [MenuItem](#)
 - [ToolbarItem](#)
- [Concepts](#)
 - [Animation](#)
 - [Simple Animations](#)
 - [Easing Functions](#)
 - [Custom Animations](#)
 - [Brushes](#)
 - [Overview](#)
 - [Solid Colors](#)
 - [Gradients](#)

[Overview](#)

[Linear Gradients](#)

[Radial Gradients](#)

[Colors](#)

[Display pop-ups](#)

[Fonts](#)

[Graphics with SkiaSharp](#)

[Splash screen](#)

[Styles](#)

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

[Introduction](#)

[Explicit Styles](#)

[Implicit Styles](#)

[Global Styles](#)

[Style Inheritance](#)

[Dynamic Styles](#)

[Device Styles](#)

[Style Classes](#)

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

[Theming](#)

[Theme an Application](#)

[Respond to System Theme Changes](#)

[Visual](#)

[Material Visual](#)

[Create a Visual Renderer](#)

[Visual state manager](#)

[Layouts](#)

[Overview](#)

[Choose a Layout](#)

[Core layouts](#)

[AbsoluteLayout](#)

[FlexLayout](#)

- [Grid](#)
- [RelativeLayout](#)
- [StackLayout](#)
- [Additional layouts](#)
 - [ContentView](#)
 - [Frame](#)
 - [ScrollView](#)
- [Concepts](#)
 - [Bindable Layouts](#)
 - [Custom Layouts](#)
 - [Device Orientation](#)
 - [LayoutOptions](#)
 - [Layout Compression](#)
 - [Margin and Padding](#)
 - [Tablet & Desktop](#)
- [Platform Features](#)
 - [Overview](#)
 - [Android](#)
 - [Overview](#)
 - [AndroidX Migration](#)
 - [Button Padding and Shadows](#)
 - [Entry Input Method Editor Options](#)
 - [ImageButton Drop Shadows](#)
 - [ListView Fast Scrolling](#)
 - [NavigationPage Bar Height](#)
 - [Page Lifecycle Events](#)
 - [Soft Keyboard Input Mode](#)
 - [SwipeView Swipe Transition Mode](#)
 - [TabbedPage Page Swiping](#)
 - [TabbedPage Page Transition Animations](#)
 - [TabbedPage Toolbar Placement and Color](#)
 - [ViewCell Context Actions](#)

[VisualElement Elevation](#)

[VisualElement Legacy Color Mode](#)

[WebView Mixed Content](#)

[WebView Zoom](#)

[iOS](#)

[Overview](#)

[Accessibility Scaling for Named Font Sizes](#)

[Cell Background Color](#)

[DatePicker Item Selection](#)

[Entry Cursor Color](#)

[Entry Font Size](#)

[FlyoutPage Shadow](#)

[Formatting](#)

[Modal Page Presentation Style](#)

[Large Page Titles](#)

[ListView Group Header Style](#)

[ListView Row Animations](#)

[ListView Separator Style](#)

[MainThread Control Updates](#)

[NavigationPage Bar Separator](#)

[NavigationPage Bar Text Color Mode](#)

[NavigationPage Bar Translucency](#)

[Page Home Indicator Visibility](#)

[Page Status Bar Visibility](#)

[Picker Item Selection](#)

[Safe Area Layout Guide](#)

[ScrollView Content Touches](#)

[SearchBar Style](#)

[Simultaneous Pan Gesture Recognition](#)

[Slider Thumb Tap](#)

[SwipeView Swipe Transition Mode](#)

[TabbedPage Translucent TabBar](#)

- TimePicker Item Selection
- VisualElement Blur
- VisualElement Drop Shadows
- VisualElement Legacy Color Mode
- VisualElement First Responder

- Windows

- Overview
- Default Image Directory
- FlyoutPage Navigation Bar
- InputView Reading Order
- ListView SelectionMode
- Page Toolbar Placement
- Platform Setup
- RefreshView Pull Direction
- SearchBar Spell Check
- TabbedPage Icons
- VisualElement Access Keys
- VisualElement Legacy Color Mode
- WebView Execution Mode
- WebView JavaScript Alerts

- Create Platform-Specifics

- Device Class

- Native Forms

- Native Views

- Native Views in XAML

- Native Views in C#

- Sign In with Apple

- Setup for iOS

- Setup for other platforms

- Use Sign In with Apple

- Other Platforms

- GTK#

[Mac](#)
[Tizen](#)
[WPF](#)

[Xamarin.Essentials](#)

[Get Started](#)

[Platform & Feature Support](#)

[Accelerometer](#)

[App Actions](#)

[App Information](#)

[App Theme](#)

[Barometer](#)

[Battery](#)

[Clipboard](#)

[Color Converters](#)

[Compass](#)

[Connectivity](#)

[Contacts](#)

[Detect Shake](#)

[Device Display Information](#)

[Device Information](#)

[Email](#)

[File Picker](#)

[File System Helpers](#)

[Flashlight](#)

[Geocoding](#)

[Geolocation](#)

[Gyroscope](#)

[Haptic Feedback](#)

[Launcher](#)

[Magnetometer](#)

[Main Thread](#)

[Maps](#)

- [Media Picker](#)
- [Open Browser](#)
- [Orientation Sensor](#)
- [Permissions](#)
- [Phone Dialer](#)
- [Platform Extensions](#)
- [Preferences](#)
- [Screenshot](#)
- [Secure Storage](#)
- [Share](#)
- [SMS](#)
- [Text-to-Speech](#)
- [Unit Converters](#)
- [Version Tracking](#)
- [Vibrate](#)
- [Web Authenticator](#)
- [Xamarin.Essentials release notes](#)
- [Troubleshooting](#)
- [Xamarin.Essentials on Q&A](#)
- [Data & Azure Cloud Services](#)
 - [Overview](#)
 - [Local data storage](#)
 - [Overview](#)
 - [File Handling](#)
 - [Local Databases](#)
 - [Azure Services](#)
 - [Azure services overview](#)
 - [Azure Mobile Apps](#)
 - [Azure Cosmos DB Document Database](#)
 - [Azure Notification Hubs](#)
 - [Azure Storage](#)
 - [Azure Search](#)

[Azure Functions](#)

[Azure Cognitive Services](#)

[Cognitive services overview](#)

[Introduction](#)

[Speech Recognition](#)

[Spell Check](#)

[Text Translation](#)

[Perceived Emotion Recognition](#)

[Web Services](#)

[Web services overview](#)

[Introduction](#)

[ASMX](#)

[WCF](#)

[REST](#)

[Authentication](#)

[Authentication overview](#)

[REST](#)

[Azure Active Directory B2C](#)

[Azure Cosmos DB Authentication](#)

[Deployment & Testing](#)

[Overview](#)

[Improve Performance](#)

[Hot Restart](#)

[Automate Testing with Visual Studio App Center](#)

[Publish iOS apps](#)

[Publish Android apps](#)

[Publish UWP apps](#)

[Publish Mac apps](#)

[Advanced Concepts and Internals](#)

[Overview](#)

[Controls Class Hierarchy](#)

[Dependency Resolution](#)

[Experimental Flags](#)

[Fast Renderers](#)

[Source Link](#)

[Troubleshooting](#)

[Frequently Asked Questions](#)

[How do I migrate my app to Xamarin.Forms 5.0?](#)

[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?](#)

[Xamarin.Forms on Q&A](#)

[Release notes](#)

[Samples](#)

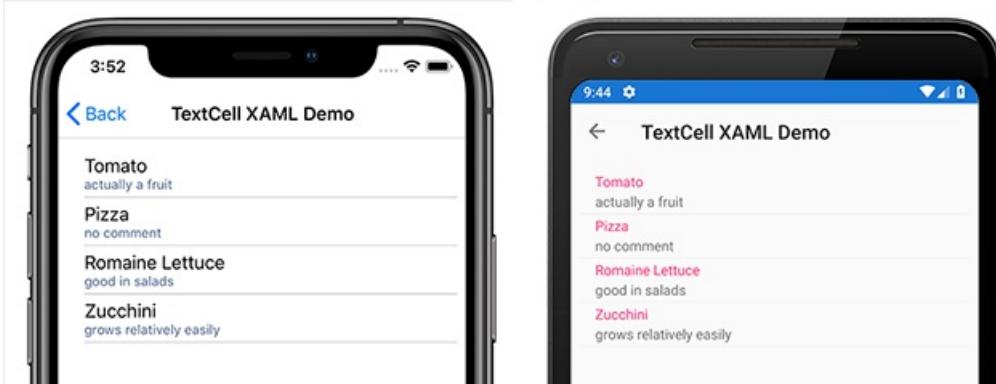
[Creating Mobile Apps with Xamarin.Forms Book](#)

[Enterprise Application Patterns eBook](#)

[SkiaSharp Graphics in Xamarin.Forms](#)

What is Xamarin.Forms?

8/4/2022 • 2 minutes to read • [Edit Online](#)



Xamarin.Forms is an open-source UI framework. Xamarin.Forms allows developers to build Xamarin.Android, Xamarin.iOS, and Windows applications from a single shared codebase.

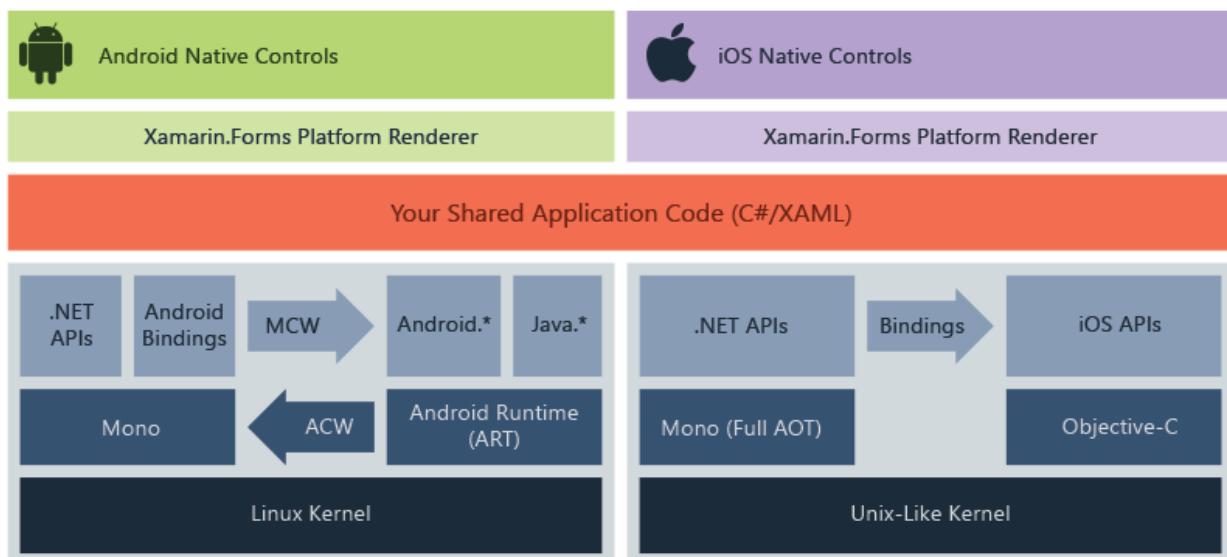
Xamarin.Forms allows developers to create user interfaces in XAML with code-behind in C#. These interfaces are rendered as performant native controls on each platform.

Who Xamarin.Forms is for

Xamarin.Forms is for developers with the following goals:

- Share UI layout and design across platforms.
- Share code, test and business logic across platforms.
- Write cross-platform apps in C# with Visual Studio.

How Xamarin.Forms works



Xamarin.Forms provides a consistent API for creating UI elements across platforms. This API can be implemented in either XAML or C# and supports databinding for patterns such as Model-View-ViewModel (MVVM).

At runtime, Xamarin.Forms utilizes platform renderers to convert the cross-platform UI elements into native

controls on Xamarin.Android, Xamarin.iOS and UWP. This allows developers to get the native look, feel and performance while realizing the benefits of code sharing across platforms.

Xamarin.Forms applications typically consist of a shared .NET Standard library and individual platform projects. The shared library contains the XAML or C# views and any business logic such as services, models or other code. The platform projects contain any platform-specific logic or packages the application requires.

Xamarin.Forms uses the Xamarin platform to run .NET applications natively across platforms. For more information about the Xamarin platform, see [What is Xamarin?](#).

Additional functionality

Xamarin.Forms has a large ecosystem of libraries that add diverse functionality to applications. This section describes some of this additional functionality.

Xamarin.Essentials

Xamarin.Essentials is a library that provides cross-platform APIs for native device features. Like Xamarin itself, Xamarin.Essentials is an abstraction that simplifies the process of accessing native utilities. Some examples of utilities provided by Xamarin.Essentials include:

- Device info
- File system
- Accelerometer
- Phone dialer
- Text-to-speech
- Screen lock

For more information, see [Xamarin.Essentials](#).

Shell

Xamarin.Forms Shell reduces the complexity of mobile application development by providing the fundamental features that most applications require. Some examples of features provided by Shell include:

- Common navigation experience
- URI-based navigation scheme
- Integrated search handler

For more information, see [Xamarin.Forms Shell](#)

Platform-specifics

Xamarin.Forms provides a common API that renders native controls across platforms, but a specific platform may have functionality that doesn't exist on other platforms. For example, the Android platform has native functionality for Fast Scrolling in a `ListView` but iOS does not. Xamarin.Forms platform-specifics allow you to utilize functionality that is only available on a specific platform without creating custom renderers or effects.

Xamarin.Forms includes pre-built solutions for a variety of platform-specific functionality. For more information, see:

- [Xamarin.Forms platform-specifics](#)
- [Android platform-specifics](#)
- [iOS platform-specifics](#)
- [Windows platform-specifics](#)

Material Visual

Xamarin.Forms Material Visual is used to apply Material Design rules to Xamarin.Forms applications.

Xamarin.Forms Material Visual utilizes the `Visual` property to selectively apply custom renderers to the UI, resulting in an application with a consistent look and feel across iOS and Android.

For more information, see [Xamarin.Forms Material Visual](#)

Related links

- [Get started with Xamarin.Forms](#)
- [Xamarin.Essentials](#)
- [Xamarin.Forms Shell](#)
- [Xamarin.Forms Material Visual](#)

Installing Xamarin

8/4/2022 • 2 minutes to read • [Edit Online](#)

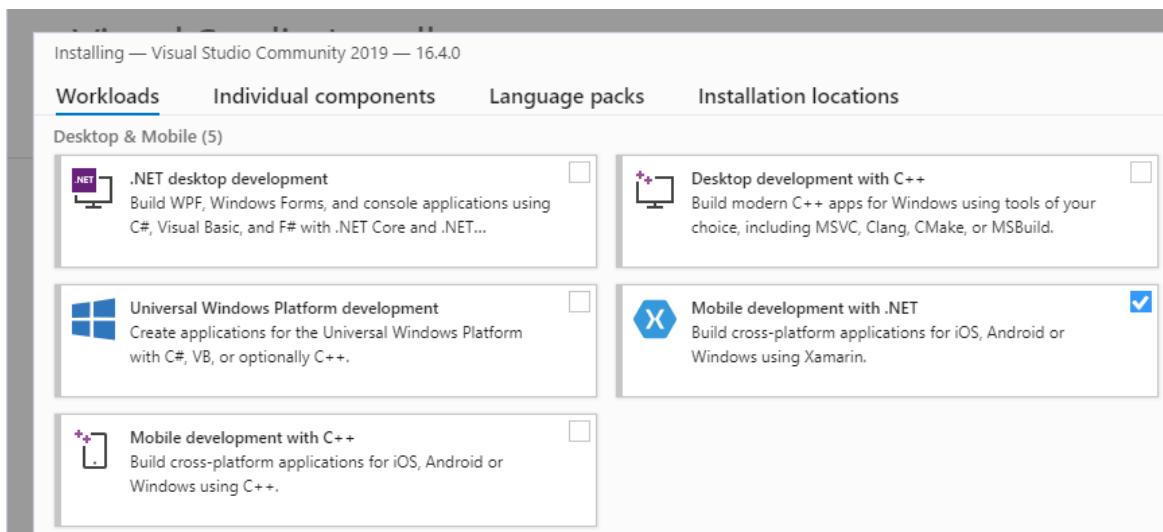
How to set up Visual Studio and Xamarin to start building mobile apps with .NET.

Installing Xamarin on Windows

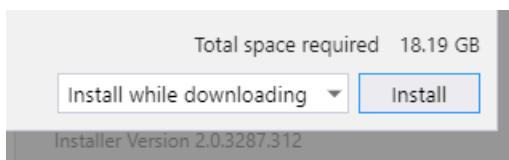
Step-by-step instructions

Xamarin can be installed as part of a *new* Visual Studio 2019 installation, with the following steps:

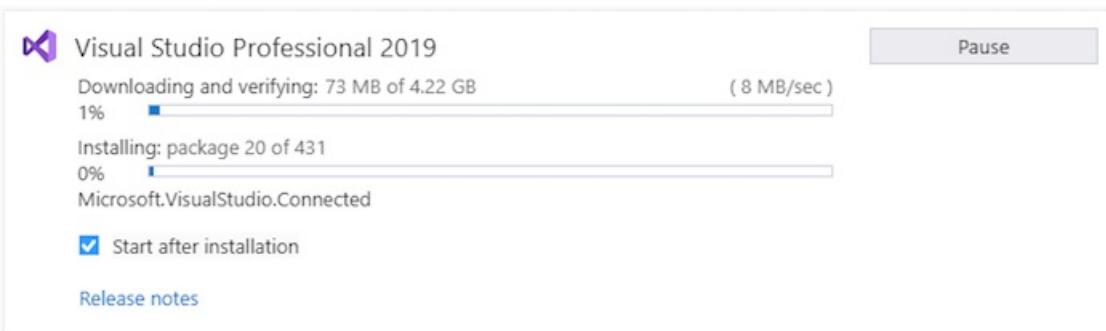
1. Download Visual Studio 2019 Community, Visual Studio Professional, or Visual Studio Enterprise from the [Visual Studio](#) page (download links are provided at the bottom).
2. Double-click the downloaded package to start installation.
3. Select the **Mobile development with .NET** workload from the installation screen:



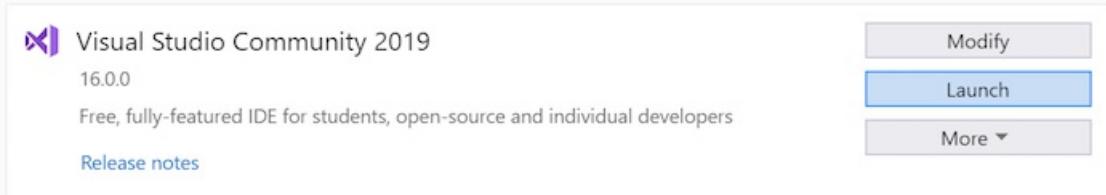
4. When you are ready to begin Visual Studio 2019 installation, click the **Install** button in the lower right-hand corner:



Use the progress bars to monitor the installation:



- When Visual Studio 2019 installation has completed, click the **Launch** button to start Visual Studio:



Adding Xamarin to Visual Studio 2019

If Visual Studio 2019 is already installed, add Xamarin by re-running the Visual Studio 2019 installer to modify workloads (see [Modify Visual Studio](#) for details). Next, follow the steps listed above to install Xamarin.

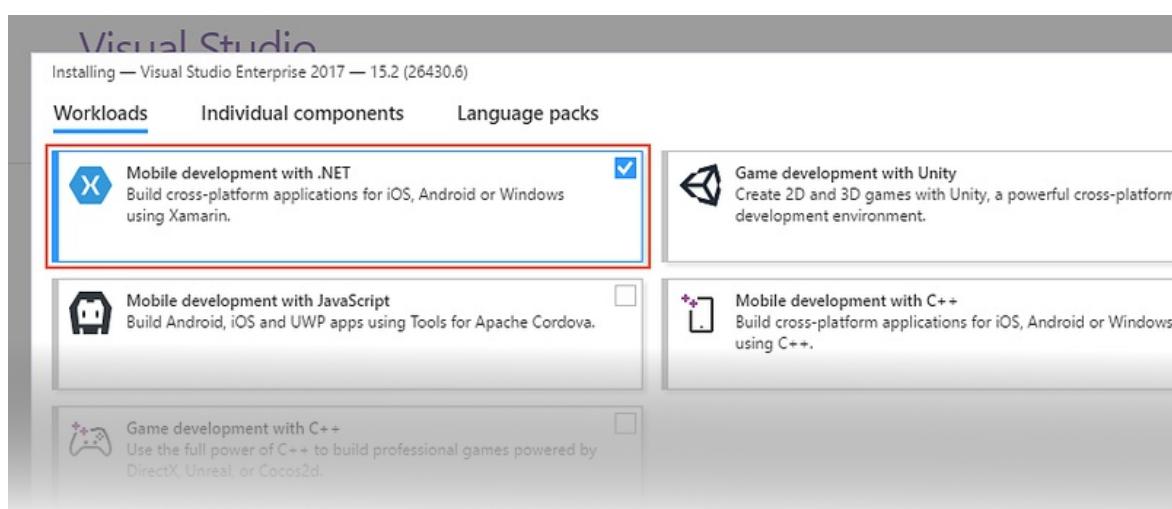
For more information about downloading and installing Visual Studio 2019, see [Install Visual Studio 2019](#).

Installing Xamarin on Windows

Step-by-step instructions

Xamarin can be installed as part of a *new* Visual Studio 2017 installation, with the following steps:

- Download Visual Studio 2017 Community, Visual Studio Professional, or Visual Studio Enterprise from the [Visual Studio](#) page (download links are provided at the bottom).
- Double-click the downloaded package to start installation.
- Select the **Mobile development with .NET** workload from the installation screen:



- While **Mobile development with .NET** is selected, have a look at the **Installation details** panel on the right. Here, you can deselect mobile development options that you do not want to install.

Installation details

✓ Visual Studio core editor

The Visual Studio core shell experience, including syntax-aware code editing, source code control and work item management.

✓ Mobile development with .NET

Included

- ✓ Xamarin
- ✓ .NET Framework 4.6.1 development tools
- ✓ C# and Visual Basic
- ✓ .NET Portable Library targeting pack

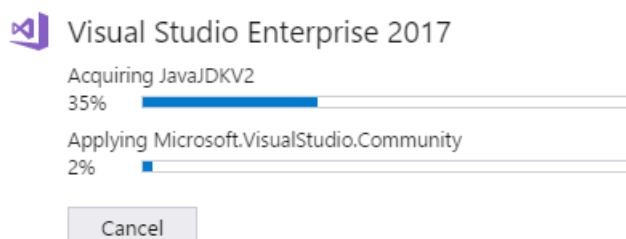
Optional

- Android SDK setup (API level 27)
- Java SE Development Kit (8.0.1120.15)
- Google Android Emulator (API Level 27)
- Xamarin Workbooks
- Intel Hardware Accelerated Execution Manager (HA...)
- Universal Windows Platform tools for Xamarin

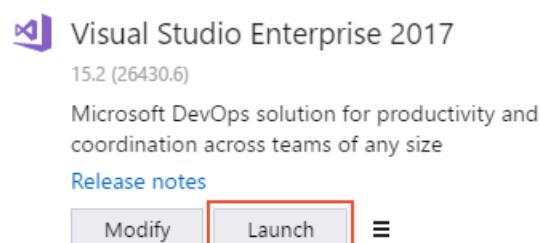
5. When you are ready to begin Visual Studio 2017 installation, click the **Install** button in the lower right-hand corner:



Depending on which edition of Visual Studio 2017 you are installing, the installation process can take a long time to complete. You can use the progress bars to monitor the installation:



6. When Visual Studio 2017 installation has completed, click the **Launch** button to start Visual Studio:



Adding Xamarin to Visual Studio 2017

If Visual Studio 2017 is already installed, add Xamarin by re-running the Visual Studio 2017 installer to modify workloads (see [Modify Visual Studio](#) for details). Next, follow the steps listed above to install Xamarin.

For more information about downloading and installing Visual Studio 2017, see [Install Visual Studio 2017](#).

Installing Xamarin on macOS

Step-by-step instructions

In addition to this video, there is a [step-by-step installation guide](#) that covers Visual Studio for Mac and Xamarin.

Related Links

- [Uninstalling Xamarin](#)
- [Xamarin Firewall Configuration Instructions](#)

Installing Xamarin in Visual Studio 2019

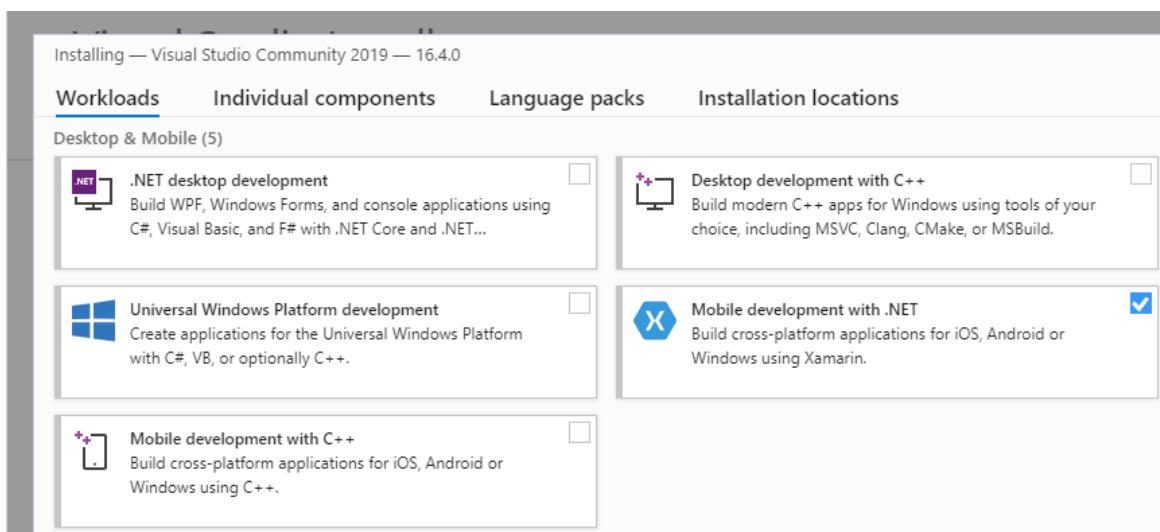
8/4/2022 • 2 minutes to read • [Edit Online](#)

Check the [system requirements](#) before you begin.

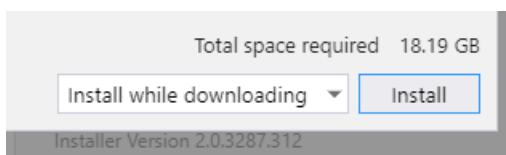
Installation

Xamarin can be installed as part of a *new* Visual Studio 2019 installation, with the following steps:

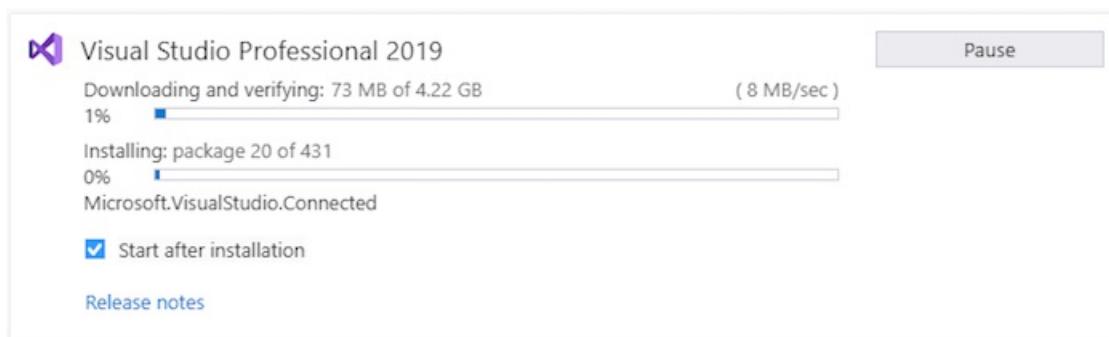
1. Download Visual Studio 2019 Community, Visual Studio Professional, or Visual Studio Enterprise from the [Visual Studio](#) page (download links are provided at the bottom).
2. Double-click the downloaded package to start installation.
3. Select the **Mobile development with .NET** workload from the installation screen:



4. When you are ready to begin Visual Studio 2019 installation, click the **Install** button in the lower right-hand corner:



Use the progress bars to monitor the installation:



5. When Visual Studio 2019 installation has completed, click the **Launch** button to start Visual Studio:

Visual Studio Community 2019
16.0.0
Free, fully-featured IDE for students, open-source and individual developers
Release notes

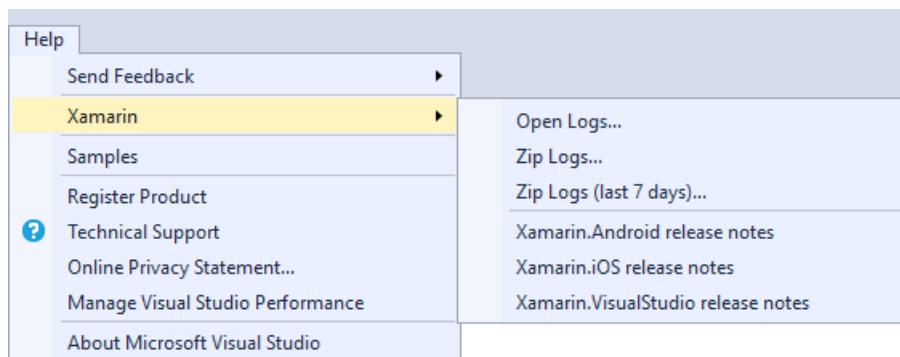
Modify
Launch
More ▾

Adding Xamarin to Visual Studio 2019

If Visual Studio 2019 is already installed, add Xamarin by re-running the Visual Studio 2019 installer to modify workloads (see [Modify Visual Studio](#) for details). Next, follow the steps listed above to install Xamarin.

For more information about downloading and installing Visual Studio 2019, see [Install Visual Studio 2019](#).

In Visual Studio 2019, verify that Xamarin is installed by clicking the **Help** menu. If Xamarin is installed, you should see a **Xamarin** menu item as shown in this screenshot:



You can also click **Help > About Microsoft Visual Studio** and scroll through the list of installed products to see if Xamarin is installed:

The screenshot shows the 'About Microsoft Visual Studio' dialog. In the 'Installed products:' section, the 'Xamarin' entry is highlighted with a red box. The 'Product details:' section contains the text: 'C# components used in the IDE. Depending on your project type and settings, a different version of the compiler may be used.' At the bottom, a warning message reads: 'Warning: This computer program is protected by copyright law and international treaties. Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law.' There are 'OK' and 'Cancel' buttons at the bottom right.

About Microsoft Visual Studio

Visual Studio

Microsoft Visual Studio Community 2017
Version 15.8.1
© 2017 Microsoft Corporation.
All rights reserved.

Microsoft .NET Framework
Version 4.7.02556
© 2017 Microsoft Corporation.
All rights reserved.

Installed products:

- Visual Basic Tools – 2.9.0-beta8-63208-01
- Visual F# Tools 10.2 for F# 4.5 – 15.8.0.0. Commit Hash: c55dd2c3d618eb93a8d16e503947342b1fa93556.
- Visual Studio Code Debug Adapter Host Package – 1.0
- VisualStudio.Mac – 1.0
- Xamarin – 4.11.0.732 (d15-8@33e83e124)
- Xamarin Designer – 4.14.218 (79f535bdd)
- Xamarin Templates – 1.1.113 (e1d02a7)
- Xamarin.Android SDK – 9.0.0.18 (HEAD/3d8a28f1a)
- Xamarin.iOS and Xamarin.Mac SDK – 11.14.0.13 (373c313)

Product details:

C# components used in the IDE. Depending on your project type and settings, a different version of the compiler may be used.

Warning: This computer program is protected by copyright law and international treaties. Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under the law.

OK

For more information about locating version information, see [Where can I find my version information and](#)

logs?

Next steps

Installing Xamarin in Visual Studio 2019 allows you to start writing code for your apps, but does require additional setup for building and deploying your apps to simulator, emulator, and device. Visit the following guides to complete your installation and start building cross platform apps.

iOS

For more detailed information, see the [Installing Xamarin.iOS on Windows](#) guide.

1. [Install Visual Studio for Mac](#)
2. [Connect Visual Studio to your Mac build host](#)
3. [iOS Developer Setup](#) - Required to run your application on device
4. [Remoted iOS Simulator](#)
5. [Introduction to Xamarin.iOS for Visual Studio](#)

Android

For more detailed information, see the [Installing Xamarin.Android on Windows](#) guide.

1. [Xamarin.Android Configuration](#)
2. [Using the Xamarin Android SDK Manager](#)
3. [Android SDK Emulator](#)
4. [Set Up Device for Development](#)

Installing Xamarin Preview on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)

Visual Studio 2019 and Visual Studio 2017 do not support alpha, beta, and stable channels in the same way as earlier versions. Instead, there are just two options:

- **Release** – equivalent to the *Stable* channel in Visual Studio for Mac
- **Preview** – equivalent to the *Alpha* and *Beta* channels in Visual Studio for Mac

TIP

To try out pre-release features, you should [download the Visual Studio Preview installer](#), which will offer the option to install **Preview** versions of Visual Studio side-by-Side with the stable (Release) version. More information on What's new in Visual Studio 2019 can be found in the [release notes](#).

The Preview version of Visual Studio may include corresponding Preview versions of Xamarin functionality, including:

- Xamarin.Forms
- Xamarin.iOS
- Xamarin.Android
- Xamarin Profiler
- Xamarin Inspector
- Xamarin Remote iOS Simulator

The **Preview Installer** screenshot below shows both Preview and Release options (notice the grey version numbers: version 15.0 is release and version 15.1 is a Preview):

Available

[Preview](#)



Visual Studio Enterprise 2017

Microsoft DevOps solution for productivity and coordination across teams of any size

[License terms](#) | [Release notes](#)

15.1 (26304.0 Preview)

[Install](#)

[Release](#)



Visual Studio Community 2017

Free, fully-featured IDE for students, open-source and individual developers

[License terms](#) | [Release notes](#)

15.0 (RTW 26228.4)

[Install](#)

During the installation process, an **Installation Nickname** can be applied to the side-by-side installation (so they can be distinguished in the Start menu), as shown below:

Installation nickname

Install size: 40.49 GB

[Install](#)

Uninstalling Visual Studio 2019 Preview

The **Visual Studio Installer** should also be used to un-install preview versions of Visual Studio 2019. Read the [uninstalling Xamarin guide](#) for more information.

Uninstall Xamarin from Visual Studio

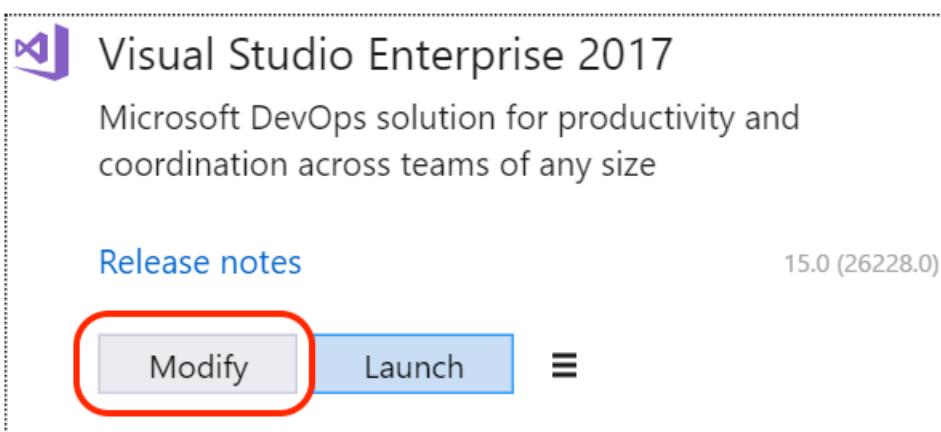
8/4/2022 • 2 minutes to read • [Edit Online](#)

This guide explains how to remove Xamarin from Visual Studio on Windows.

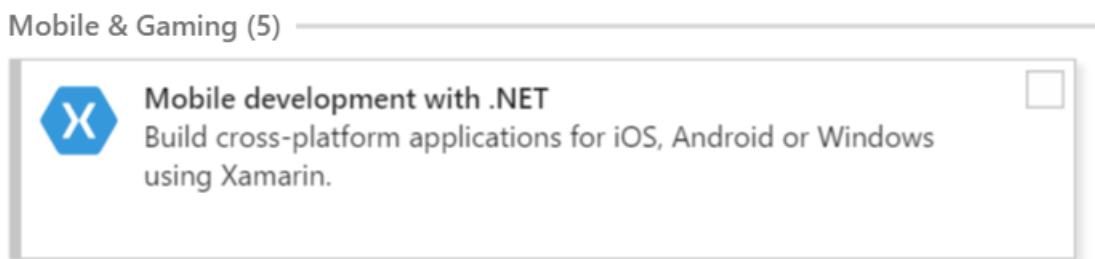
Visual Studio 2019 and Visual Studio 2017

Xamarin is uninstalled from Visual Studio 2019 and Visual Studio 2017 using the installer app:

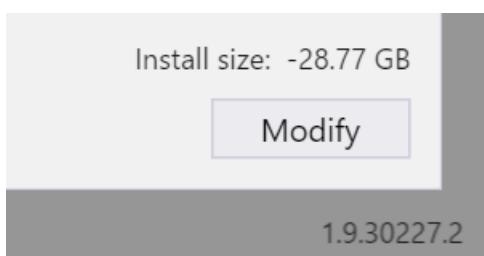
1. Use the **Start menu** to open the **Visual Studio Installer**.
2. Press the **Modify** button for the instance you wish to change.



3. In the **Workloads** tab, de-select the **Mobile Development with .NET** option (in the **Mobile & Gaming** section).



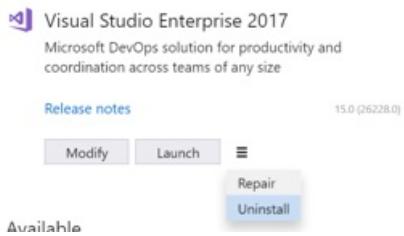
4. Click the **Modify** button in the bottom right of the window.
5. The installer will remove the de-selected components (Visual Studio 2017 must be closed before the installer can make any changes).



Individual Xamarin components (such as the Profiler or Workbooks) can be uninstalled by switching to the **Individual Components** tab in step 3, and unchecking specific components:



To uninstall Visual Studio 2017 completely, choose **Uninstall** from the three-bar menu next to the **Launch** button.



IMPORTANT

If you have two (or more) instances of Visual Studio installed side-by-side (SxS) – such as a Release and a Preview version – uninstalling one instance might remove some Xamarin functionality from the other Visual Studio instance(s), including:

- Xamarin Profiler
- Xamarin Workbooks/Inspector
- Xamarin Remote iOS Simulator
- Apple Bonjour SDK

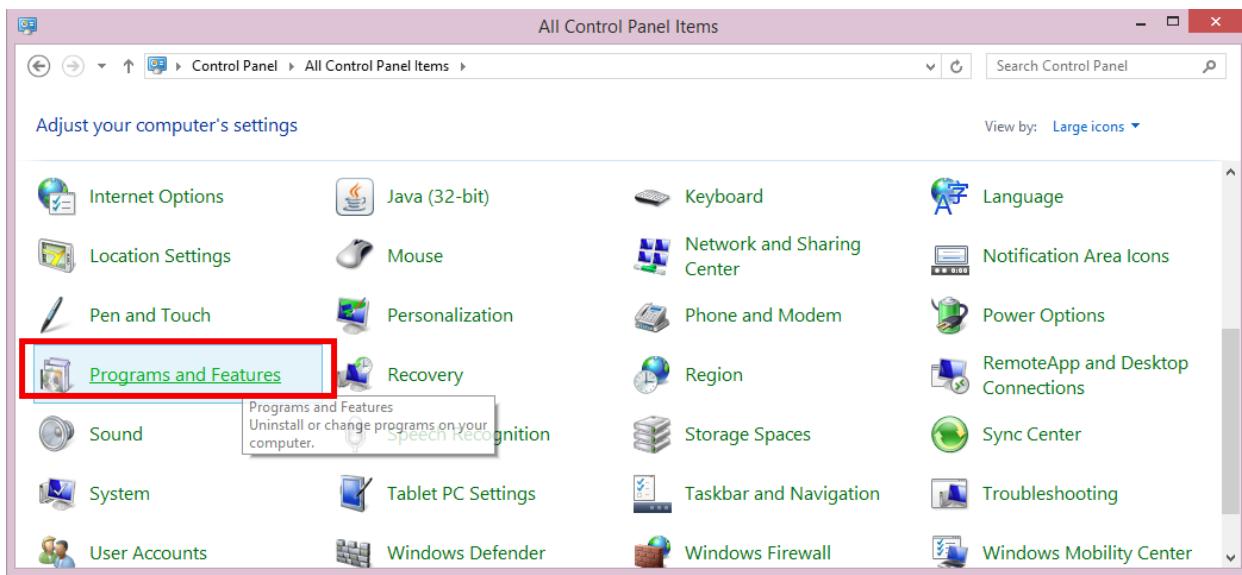
Under certain conditions, uninstalling one of the SxS instances can result in the incorrect removal of these features. This may degrade the performance of the Xamarin Platform on the Visual Studio instance(s) that remain on the system after the uninstallation of the SxS instance.

This is resolved by running the **Repair** option in the Visual Studio installer, which will re-install the missing components.

Visual Studio 2015 and earlier

To uninstall Visual Studio 2015 completely, use [the support answer on visualstudio.com](#).

Xamarin can be uninstalled from a Windows machine through **Control Panel**. Navigate to **Programs and Features** or **Programs > Uninstall a Program** as illustrated below:



From the Control Panel, uninstall any of the following that are present:

- Xamarin
- Xamarin for Windows
- Xamarin.Android
- Xamarin.iOS
- Xamarin for Visual Studio

In Explorer, delete any remaining files from the Xamarin Visual Studio extension folders (all versions, including both Program Files and Program Files (x86)):

```
C:\Program Files*\Microsoft Visual Studio 1*.0\Common7\IDE\Extensions\Xamarin
```

Delete Visual Studio's MEF component cache directory, which should be located in the following location:

```
%LOCALAPPDATA%\Microsoft\VisualStudio\1*.0\ComponentModelCache
```

Check in the **VirtualStore** directory to see if Windows might have stored any overlay files for the **Extensions\Xamarin** or **ComponentModelCache** directories there:

```
%LOCALAPPDATA%\VirtualStore
```

Open the registry editor (regedit) and look for the following key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\SharedDlls
```

Find and delete any entries that match this pattern:

```
C:\Program Files*\Microsoft Visual Studio 1*.0\Common7\IDE\Extensions\Xamarin
```

Look for this key:

```
HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\1*.0\ExtensionManager\PendingDeletions
```

Delete any entries that look like they might be related to Xamarin. For example, anything containing the terms

`mono` or `xamarin`.

Open an administrator cmd.exe command prompt, and then run the `devenv /setup` and `devenv /updateconfiguration` commands for each installed version of Visual Studio. For example, for Visual Studio 2015:

```
"%ProgramFiles(x86)%\Microsoft Visual Studio 14.0\Common7\IDE\devenv.exe" /setup  
"%ProgramFiles(x86)%\Microsoft Visual Studio 14.0\Common7\IDE\devenv.exe" /updateconfiguration
```

Xamarin firewall configuration instructions

8/4/2022 • 2 minutes to read • [Edit Online](#)

A list of hosts that you need to allow in your firewall to allow Xamarin's platform to work for your company.

In order for Xamarin products to install and work properly, certain endpoints must be accessible to download the required tools and updates for your software. If you or your company have strict firewall settings, you may experience issues with installation, licensing, components, and more. This document outlines some of the known endpoints that need to be allowed in your firewall in order for Xamarin to work. This list does not include the endpoints required for any third-party tools included in the download. If you are still experiencing trouble after going through this list, refer to the Apple or Android installation troubleshooting guides.

Endpoints to allow

Xamarin installer

The following known addresses will need to be added in order for the software to install properly when using the latest release of the Xamarin installer:

- xamarin.com (installer manifests)
- dl.xamarin.com (Package download location)
- dl.google.com (to download the Android SDK)
- download.oracle.com (JDK)
- visualstudio.com (Setup packages download location)
- go.microsoft.com (Setup URL resolution)
- aka.ms (Setup URL resolution)

If you are using a Mac and are encountering Xamarin.Android install issues, please ensure that macOS is able to download Java.

NuGet (including Xamarin.Forms)

The following addresses will need to be added to access NuGet (Xamarin.Forms is packaged as a NuGet):

- www.nuget.org (to access NuGet)
- globalcdn.nuget.org (NuGet downloads)
- dl-ssl.google.com (Google components for Android and Xamarin.Forms)

Software updates

The following addresses will need to be added to ensure that software updates can download properly:

- software.xamarin.com (updater service)
- download.visualstudio.microsoft.com
- dl.xamarin.com

Xamarin Mac Agent

To connect Visual Studio to a Mac build host using the Xamarin Mac Agent requires the SSH port to be open. By default this is **Port 22**.

Xamarin.Forms supported platforms

8/4/2022 • 2 minutes to read • [Edit Online](#)

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

- iOS 9 or higher.
- Android 4.4 (API 19) or higher ([more details](#)). However, Android 5.0 (API 21) is recommended as the minimum API. This ensures full compatibility with all the Android support libraries, while still targeting the majority of Android devices.
- Windows 10 Universal Windows Platform, build 10.0.16299.0 or greater for .NET Standard 2.0 support. However, build 10.0.18362.0 or greater is recommended.

Xamarin.Forms apps for iOS, Android, and the Universal Windows Platform (UWP) can be built in Visual Studio. However, a networked Mac is required for iOS development using the latest version of Xcode and the minimum version of macOS specified by Apple. For more information, see [Windows requirements](#).

Xamarin.Forms apps for iOS and Android can be built in Visual Studio for Mac. For more information, see [macOS requirements](#).

NOTE

Developing apps using Xamarin.Forms requires familiarity with [.NET Standard](#).

Additional platform support

Xamarin.Forms supports additional platforms beyond iOS, Android, and Windows:

- Samsung Tizen
- macOS 10.13 or higher
- GTK#
- WPF

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

Android platform support

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](#)



Deprecated platforms

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

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

Build your first Xamarin.Forms App

8/4/2022 • 3 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

 [Download the sample](#)

Follow these steps along with the video above:

1. Choose **File > New > Project...** or press the **Create new project...** button:

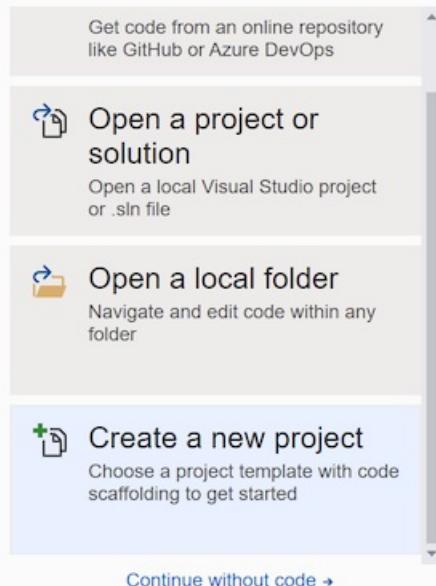
Visual Studio 2019

Open recent

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

Get started

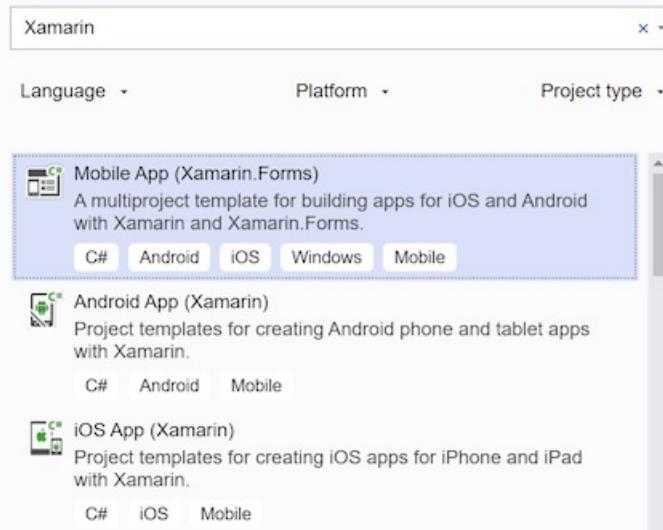


2. Search for "Xamarin" or choose **Mobile** from the **Project type** menu. Select the **Mobile App (Xamarin.Forms)** project type:

Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.



3. Choose a project name – the example uses "AwesomeApp":

Configure your new project

Mobile App (Xamarin.Forms) C# Android iOS Windows Mobile

Project name

AwesomeApp

Location

C:\Users\yourname\source\repos

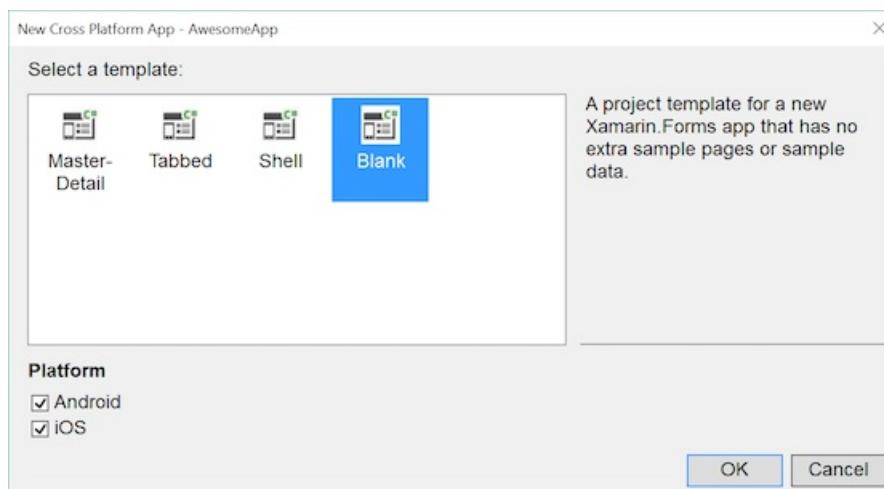


Solution name ⓘ

AwesomeApp

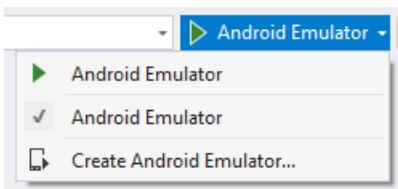
Place solution and project in the same directory

4. Click on the **Blank** project type and ensure **Android** and **iOS** are selected:



5. Wait until the NuGet packages are restored (a "Restore completed" message will appear in the status bar).

6. New Visual Studio 2019 installations won't have an Android emulator configured. Click the dropdown arrow on the **Debug** button and choose **Create Android Emulator** to launch the emulator creation screen:



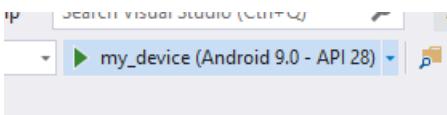
7. In the emulator creation screen, use the default settings and click the **Create** button:



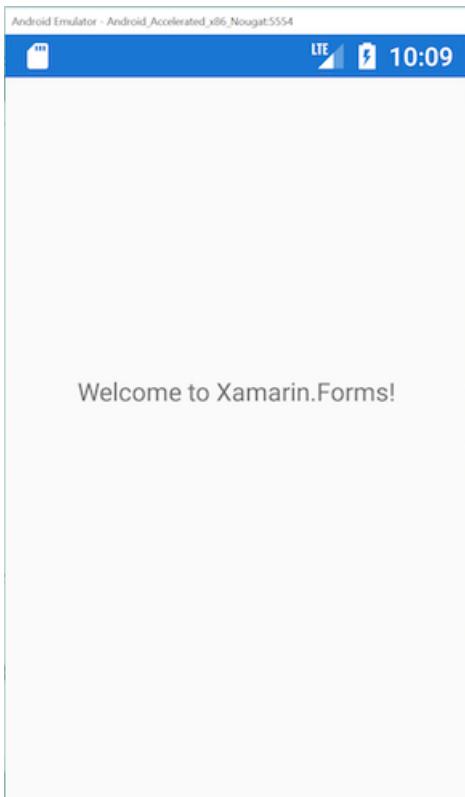
8. Creating an emulator will return you to the Device Manager window. Click the **Start** button to launch the new emulator:

Name	OS	Processor	Memory	Resolution	
My Device + Google Play	Pie 9.0 – API 28	x86	1 GB	1080 x 1920 420 dpi	Start

9. Visual Studio 2019 should now show the name of the new emulator on the **Debug** button:



10. Click the **Debug** button to build and deploy the application to the Android emulator:



Customize the application

The application can be customized to add interactive functionality. Perform the following steps to add user interaction to the application:

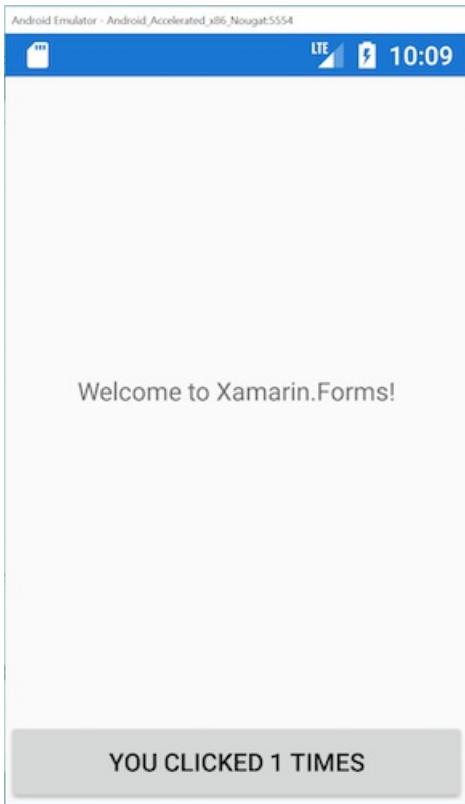
1. Edit **MainPage.xaml**, adding this XAML before the end of the `</StackLayout>`:

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

2. 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.";
}
```

3. Debug the app on Android:



NOTE

The sample application includes the additional interactive functionality that is not covered in the video.

Build an iOS app in Visual Studio 2019

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

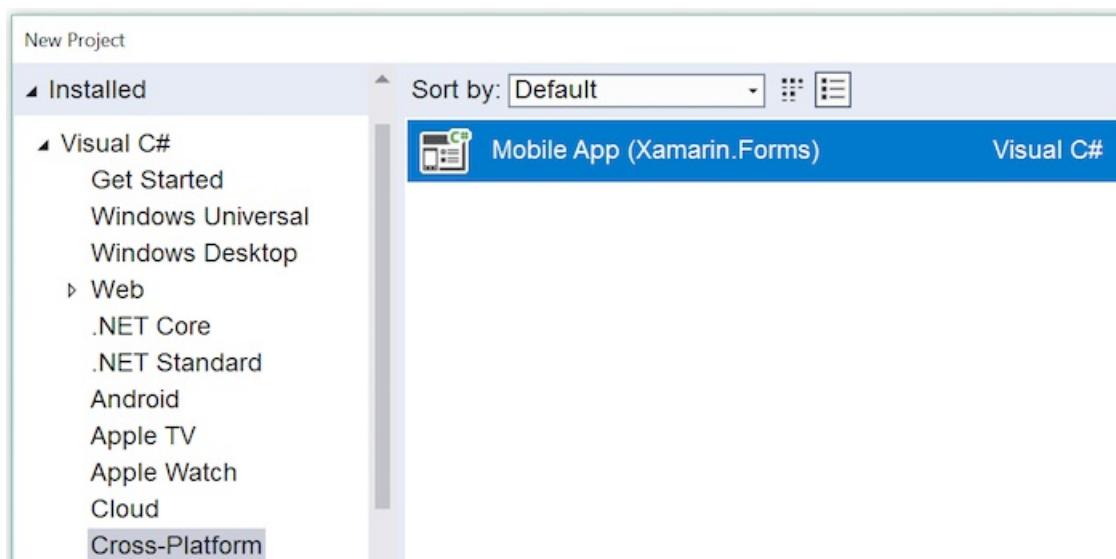
This video covers the process of building and testing an iOS app using Visual Studio 2019 on Windows:

Step-by-step instructions for Windows

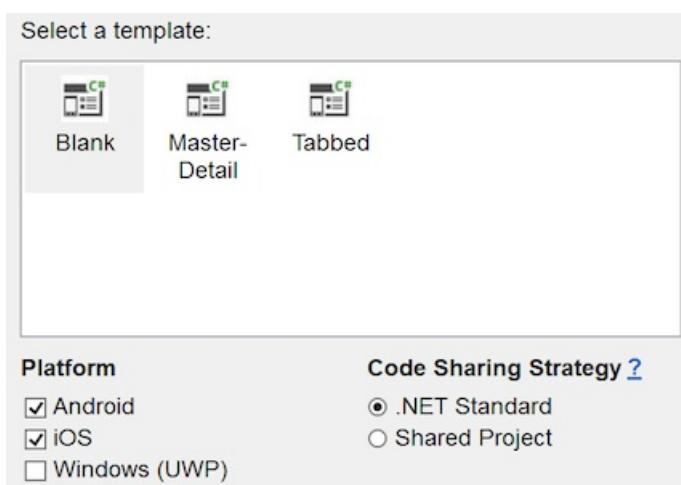
 [Download the sample](#)

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:



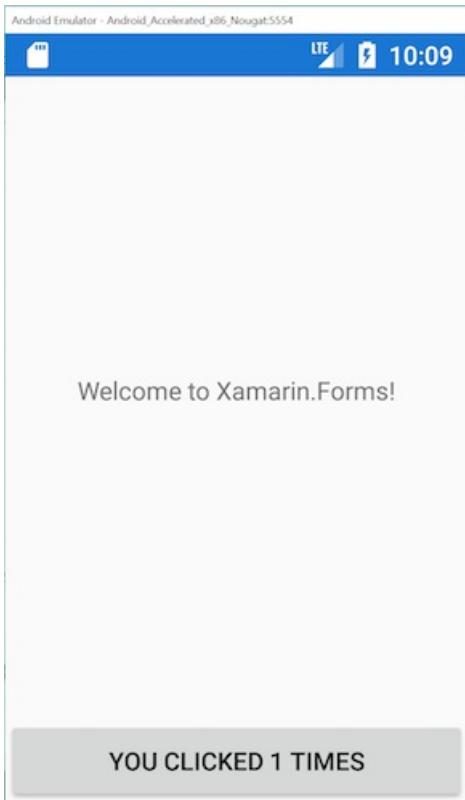
3. Wait until the NuGet packages are restored (a "Restore completed" message will appear in the status bar).
4. Launch Android emulator by pressing the debug button (or the **Debug > Start Debugging** menu item).
5. Edit **MainPage.xaml**, adding this XAML before the end of the `</StackLayout>`:

```
<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.

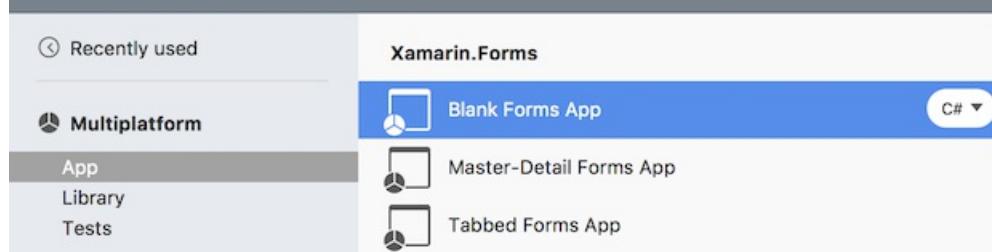
Step-by-step instructions for Mac

 [Download the sample](#)

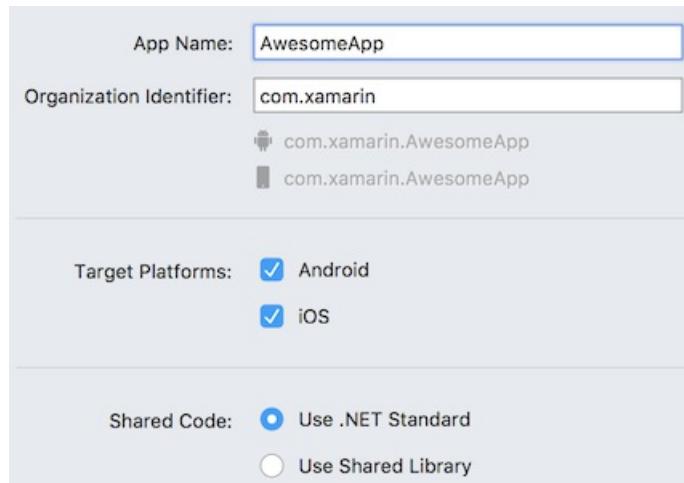
Follow these steps along with the video above:

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

Choose a template for your new project



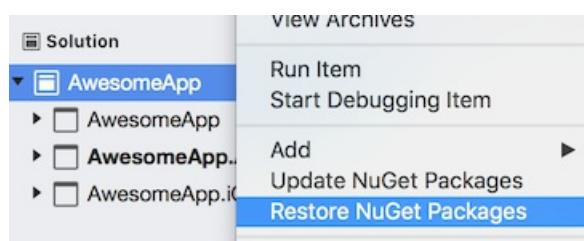
2. Ensure Android and iOS are selected, with .NET Standard code sharing:



NOTE

Only A-Z, a-z, '_', '.', and numbers are supported characters for your App Name and Organization Identifier.

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 `</StackLayout>`:

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

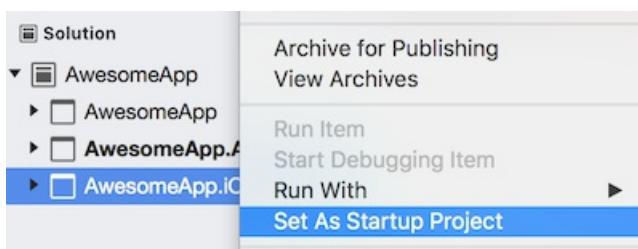
6. Edit **MainPage.xaml.cs**, adding this code to the end of the class:

```
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

- [Single Page Quickstart](#) – Build a more functional app.
- [Xamarin.Forms Samples](#) – Download and run code examples and sample apps.
- [Creating Mobile Apps ebook](#) – In-depth chapters that teach Xamarin.Forms development, available as a PDF and including hundreds of additional samples.

Xamarin.Forms quickstarts

8/4/2022 • 2 minutes to read • [Edit Online](#)

Learn how to create mobile applications with Xamarin.Forms.

Create a Xamarin.Forms application

Learn how to create a cross-platform Xamarin.Forms application, which enables you to enter a note and persist it to device storage.

Perform navigation in a Xamarin.Forms application

Learn how to turn the application, capable of storing a single note, into an application capable of storing multiple notes.

Store data in a local SQLite.NET database

Learn how to store data in a local SQLite.NET database.

Style a cross-platform Xamarin.Forms application

Learn how to style a cross-platform Xamarin.Forms application with XAML styles.

Quickstart deep dive

Read about the fundamentals of application development using Xamarin.Forms, with a focus on the application developed throughout the quickstarts.

Create a Xamarin.Forms application quickstart

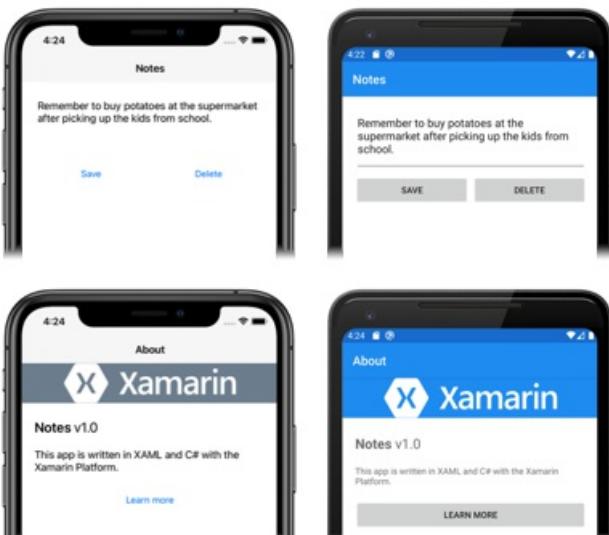
8/4/2022 • 17 minutes to read • [Edit Online](#)

 [Download the sample](#)

In this quickstart, you will learn how to:

- Create a Xamarin.Forms Shell application.
- Define the user interface for a page using eXtensible Application Markup Language (XAML), and interact with XAML elements from code.
- Describe the visual hierarchy of a Shell application by subclassing the `Shell` class.

The quickstart walks through how to create a cross-platform Xamarin.Forms Shell application, which enables you to enter a note and persist it to device storage. The final application is shown below:



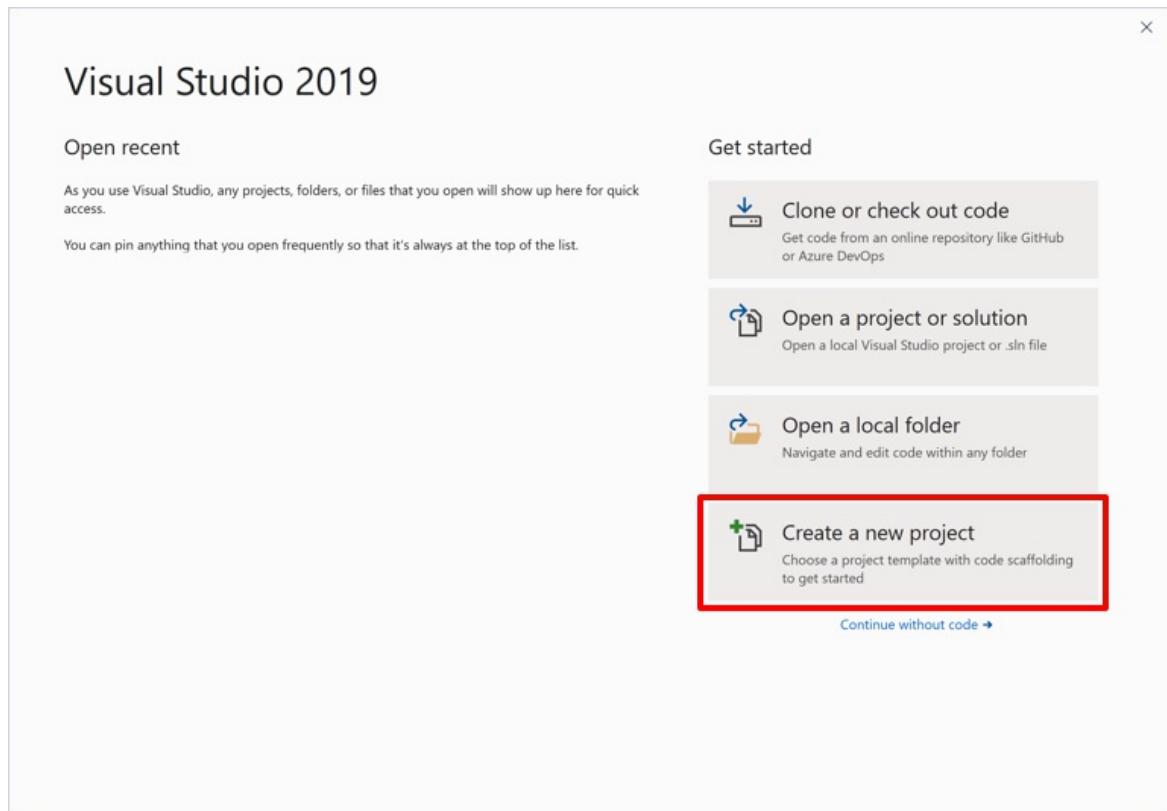
Prerequisites

- Visual Studio 2019 (latest release), with the **Mobile development with .NET** workload installed.
- Knowledge of C#.
- (optional) A paired Mac to build the application on iOS.

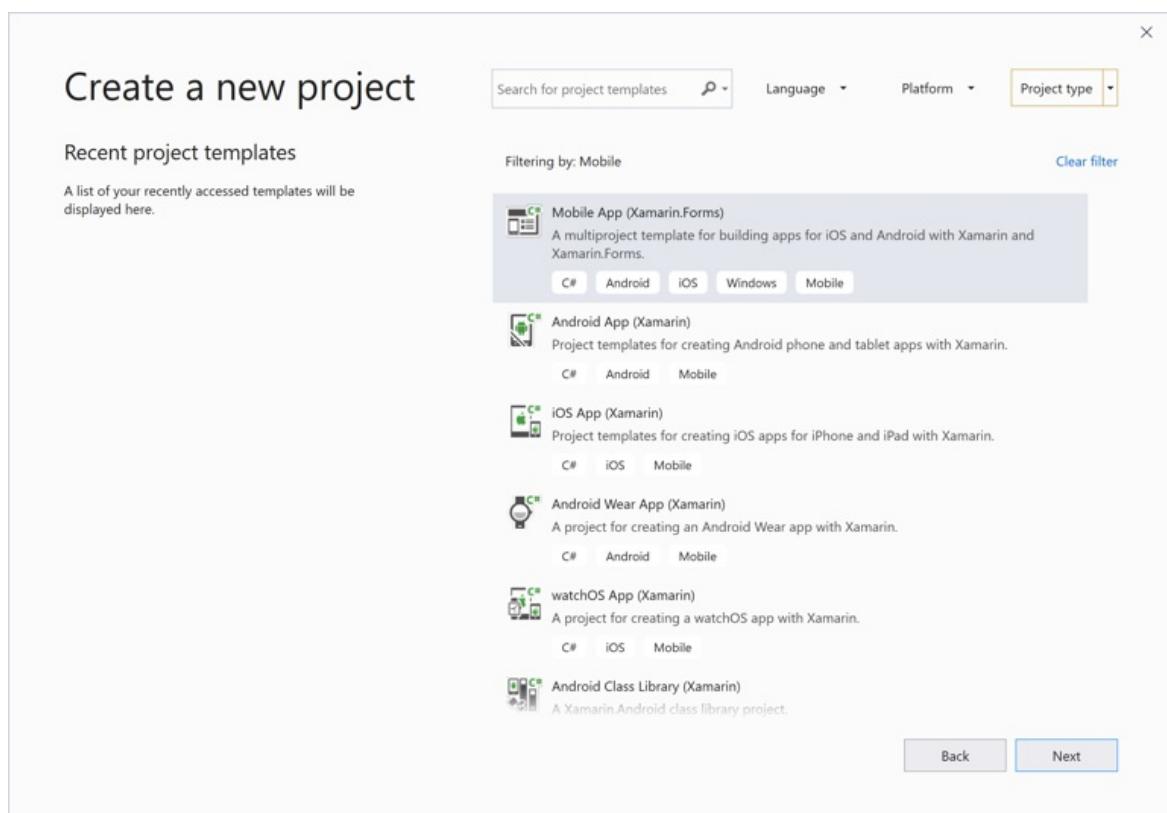
For more information about these prerequisites, see [Installing Xamarin](#). For information about connecting Visual Studio 2019 to a Mac build host, see [Pair to Mac for Xamarin.iOS development](#).

Get started with Visual Studio 2019

1. Launch Visual Studio 2019, and in the start window click **Create a new project** to create a new project:



2. In the **Create a new project** window, select **Mobile** in the **Project type** drop-down, select the **Mobile App (Xamarin.Forms)** template, and click the **Next** button:



3. In the **Configure your new project** window, set the **Project name** to **Notes**, choose a suitable location for the project, and click the **Create** button:

Configure your new project

Mobile App (Xamarin.Forms) C# Android iOS Windows Mobile

Project name

Location

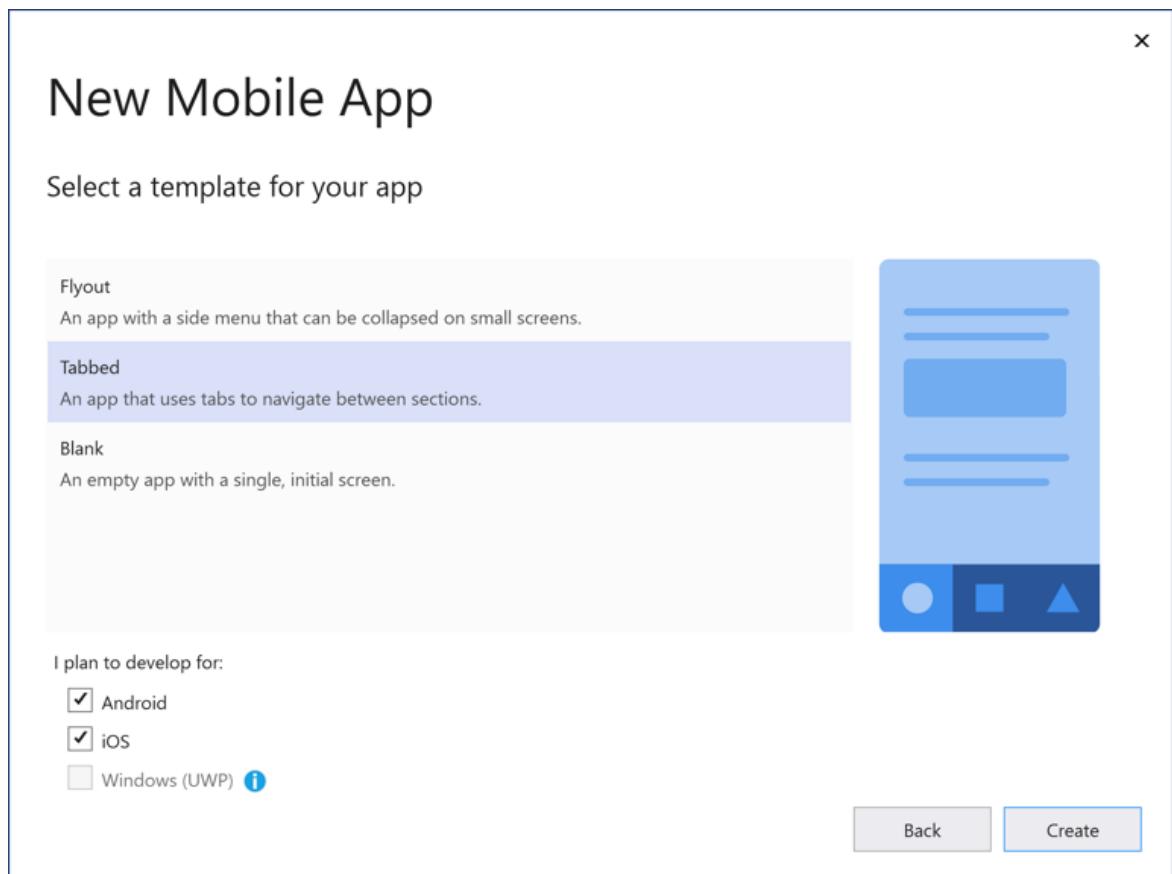
Solution name ⓘ

Place solution and project in the same directory

IMPORTANT

The C# and XAML snippets in this quickstart requires that the solution and project are both named **Notes**. Using a different name will result in build errors when you copy code from this quickstart into the project.

4. In the **New Mobile App** dialog, select the **Tabbed** template, and click the **Create** button:



When the project has been created, close the **GettingStarted.txt** file.

For more information about the .NET Standard library that gets created, see [Anatomy of a Xamarin.Forms Shell application](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

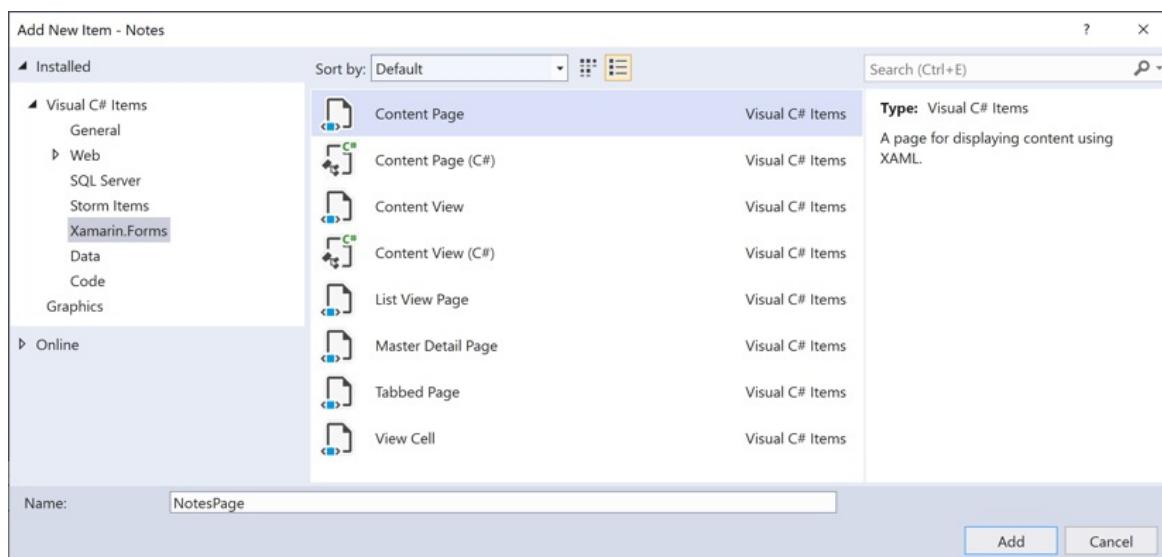
5. In Solution Explorer, in the **Notes** project, delete the following folders (and their contents):

- **Models**
- **Services**
- **ViewModels**
- **Views**

6. In Solution Explorer, in the **Notes** project, delete **GettingStarted.txt**.

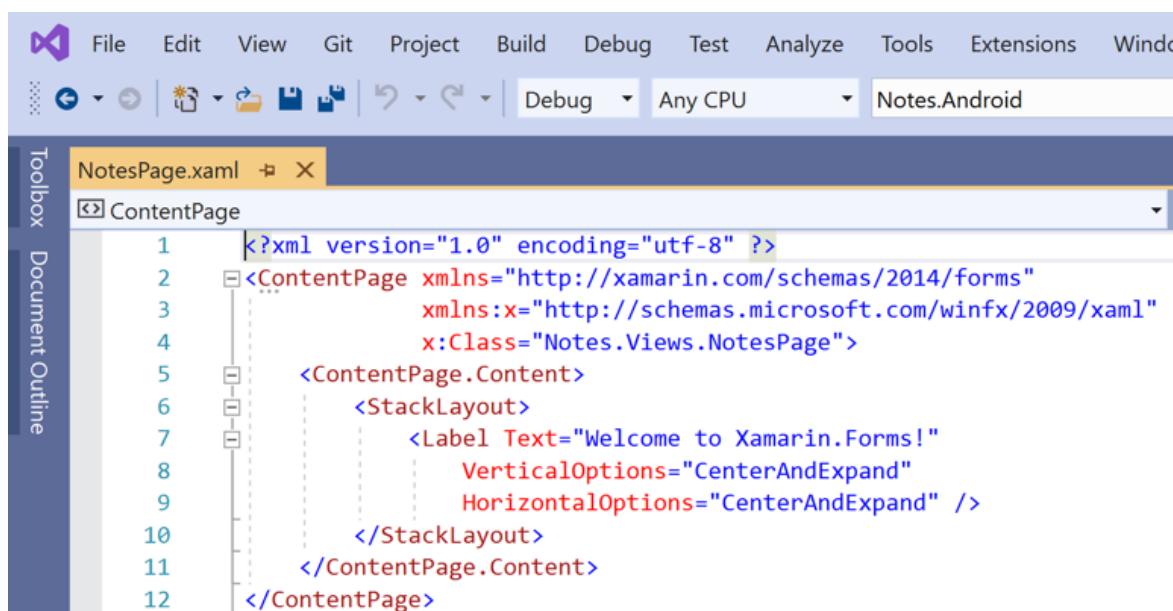
7. In Solution Explorer, in the **Notes** project, add a new folder named **Views**.

8. In Solution Explorer, in the **Notes** project, select the **Views** folder, right-click, and select **Add > New Item....** In the **Add New Item** dialog, select **Visual C# Items > Xamarin.Forms > Content Page**, name the new file **NotesPage**, and click the **Add** button:



This will add a new page named **NotesPage** to the **Views** folder. This page will be the main page in the application.

9. In Solution Explorer, in the **Notes** project, double-click **NotesPage.xaml** to open it:



10. In **NotesPage.xaml**, remove all of the template code and replace it with the following code:

```

<?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="Notes.Views.NotesPage"
    Title="Notes">
    <!-- Layout children vertically -->
    <StackLayout Margin="20">
        <Editor x:Name="editor"
            Placeholder="Enter your note"
            HeightRequest="100" />
        <!-- Layout children in two columns -->
        <Grid ColumnDefinitions="*,*">
            <Button Text="Save"
                Clicked="OnSaveButtonClicked" />
            <Button Grid.Column="1"
                Text="Delete"
                Clicked="OnDeleteButtonClicked"/>
        </Grid>
    </StackLayout>
</ContentPage>

```

This code declaratively defines the user interface for the page, which consists of an `Editor` for text input, and two `Button` objects that direct the application to save or delete a file. The two `Button` objects are horizontally laid out in a `Grid`, with the `Editor` and `Grid` being vertically laid out in a `StackLayout`. For more information about creating the user interface, see [User interface](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `NotesPage.xaml` by pressing **CTRL+S**.

- In Solution Explorer, in the **Notes** project, double-click `NotesPage.xaml.cs` to open it:

```

File Edit View Git Project Build Debug Test Analyze Tools Extensions
Debug Any CPU Notes.Android
NotesPage.xaml.cs X
C# Notes Notes.Views.NotesPage
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 using Xamarin.Forms;
8 using Xamarin.Forms.Xaml;
9
10 namespace Notes.Views
11 {
12     [XamlCompilation(XamlCompilationOptions.Compile)]
13     public partial class NotesPage : ContentPage
14     {
15         public NotesPage()
16         {
17             InitializeComponent();
18         }
19     }
20 }

```

- In `NotesPage.xaml.cs`, remove all of the template code and replace it with the following code:

```

using System;
using System.IO;
using Xamarin.Forms;

namespace Notes.Views
{
    public partial class NotesPage : ContentPage
    {
        string _fileName =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "notes.txt");

        public NotesPage()
        {
            InitializeComponent();

            // Read the file.
            if (File.Exists(_fileName))
            {
                editor.Text = File.ReadAllText(_fileName);
            }
        }

        void OnSaveButtonClicked(object sender, EventArgs e)
        {
            // Save the file.
            File.WriteAllText(_fileName, editor.Text);
        }

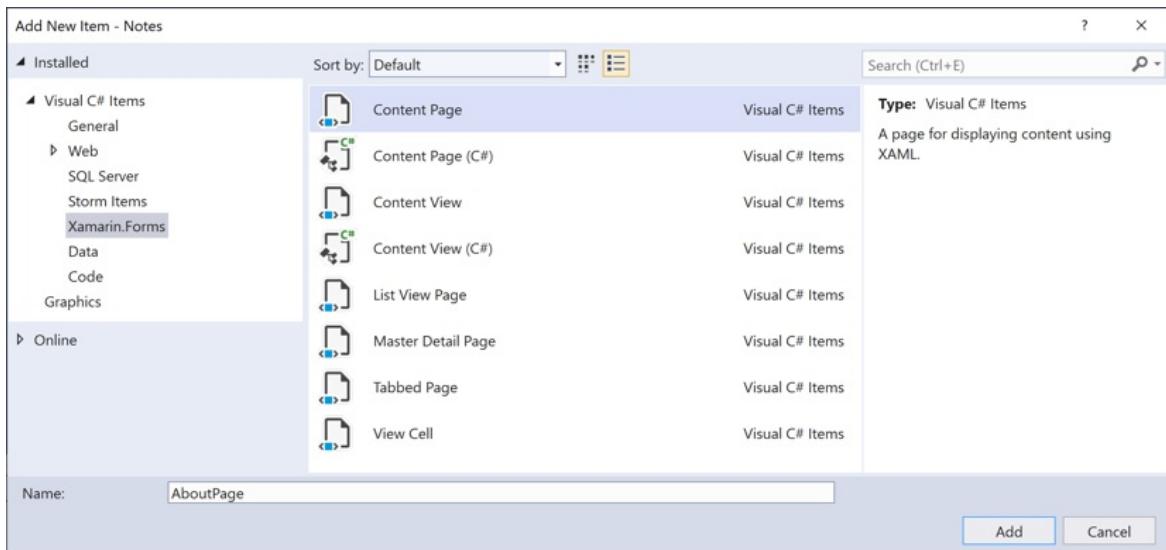
        void OnDeleteButtonClicked(object sender, EventArgs e)
        {
            // Delete the file.
            if (File.Exists(_fileName))
            {
                File.Delete(_fileName);
            }
            editor.Text = string.Empty;
        }
    }
}

```

This code defines a `_fileName` field, which references a file named `notes.txt` that will store note data in the local application data folder for the application. When the page constructor is executed the file is read, if it exists, and displayed in the `Editor`. When the `Save` `Button` is pressed the `OnSaveButtonClicked` event handler is executed, which saves the content of the `Editor` to the file. When the `Delete` `Button` is pressed the `OnDeleteButtonClicked` event handler is executed, which deletes the file, provided that it exists, and removes any text from the `Editor`. For more information about user interaction, see [Responding to user interaction](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

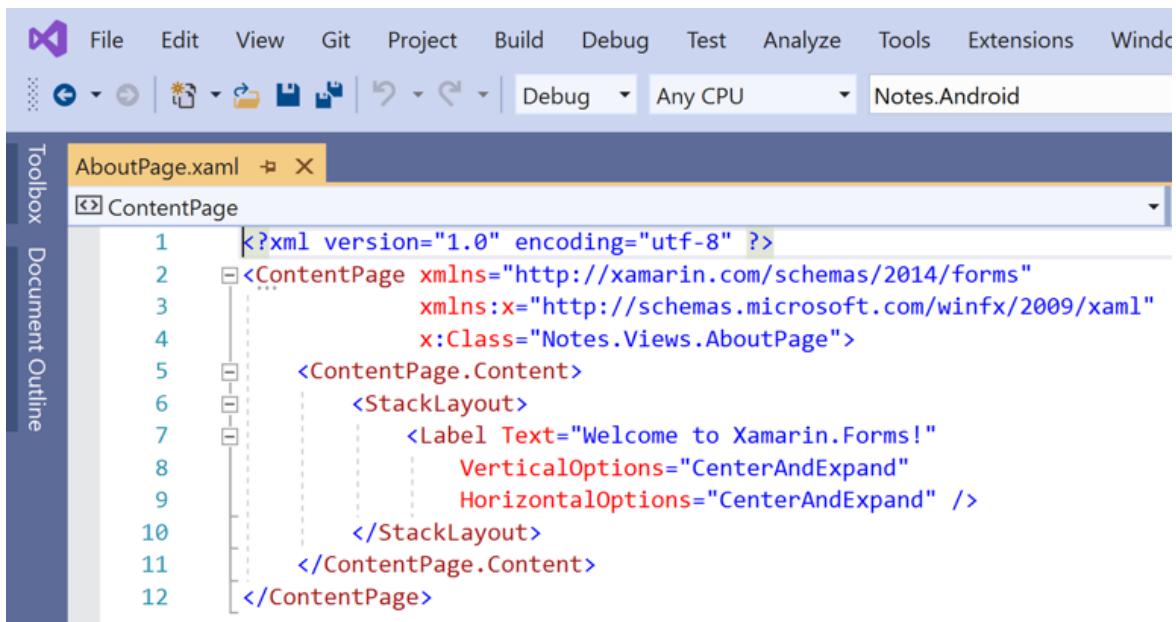
Save the changes to `NotesPage.xaml.cs` by pressing **CTRL+S**.

13. In **Solution Explorer**, in the **Notes** project, select the **Views** folder, right-click, and select **Add > New Item....** In the **Add New Item** dialog, select **Visual C# Items > Xamarin.Forms > Content Page**, name the new file **AboutPage**, and click the **Add** button:



This will add a new page named **AboutPage** to the **Views** folder.

14. In **Solution Explorer**, in the **Notes** project, double-click **AboutPage.xaml** to open it:



15. In **AboutPage.xaml**, remove all of the template code and replace it with the following code:

```

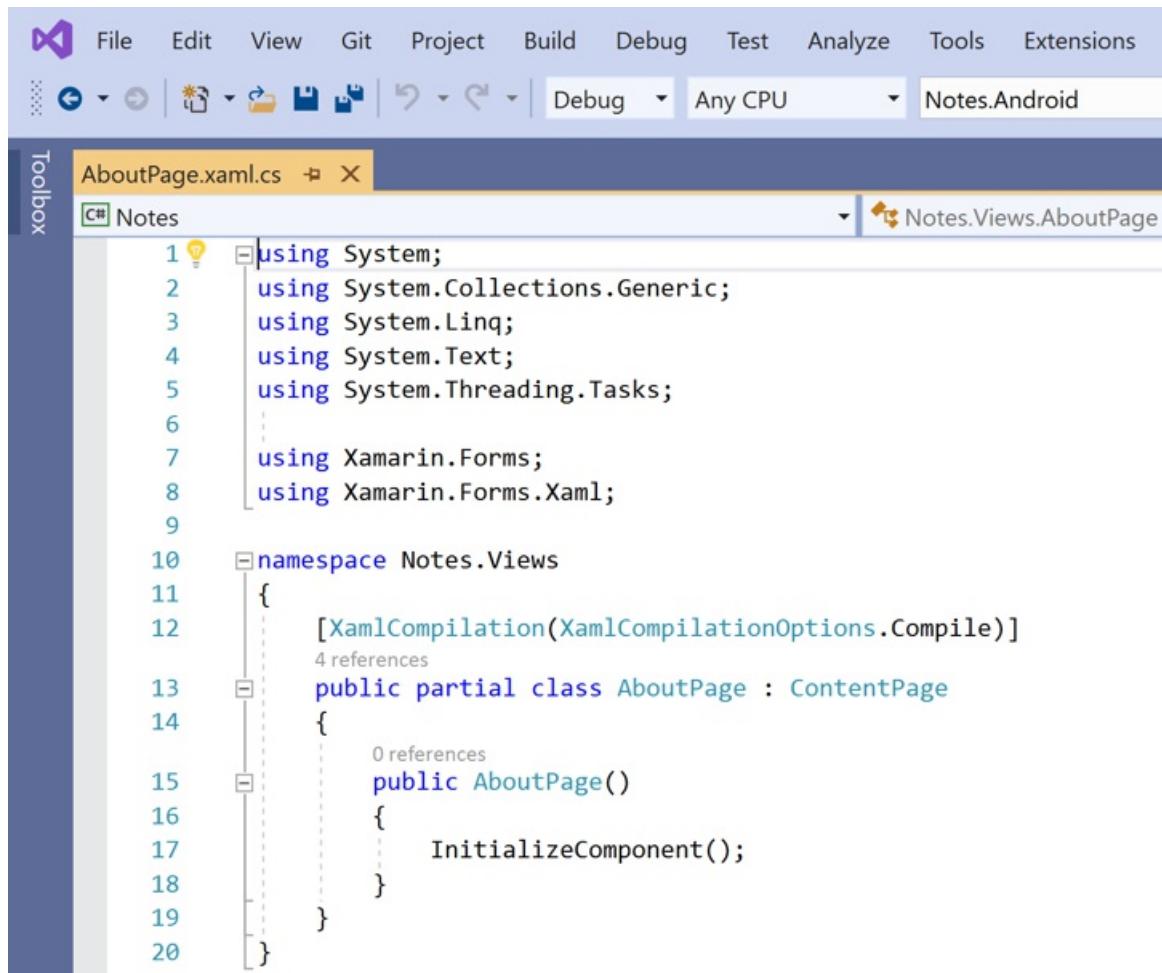
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Notes.Views.AboutPage"
    Title="About">
    <!-- Layout children in two rows -->
    <Grid RowDefinitions="Auto,*">
        <Image Source="xamarin_logo.png"
            BackgroundColor="{OnPlatform iOS=LightSlateGray, Android=#2196F3}"
            VerticalOptions="Center"
            HeightRequest="64" />
        <!-- Layout children vertically -->
        <StackLayout Grid.Row="1"
            Margin="20"
            Spacing="20">
            <Label FontSize="22">
                <Label.FormattedText>
                    <FormattedString>
                        <FormattedString.Spans>
                            <Span Text="Notes"
                                FontAttributes="Bold"
                                FontSize="22" />
                            <Span Text=" v1.0" />
                        </FormattedString.Spans>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
            <Label Text="This app is written in XAML and C# with the Xamarin Platform." />
            <Button Text="Learn more"
                Clicked="OnButtonClicked" />
        </StackLayout>
    </Grid>
</ContentPage>

```

This code declaratively defines the user interface for the page, which consists of an `Image`, two `Label` objects that display text, and a `Button`. The two `Label` objects and `Button` are vertically laid out in a `StackLayout`, with the `Image` and `StackLayout` being vertically laid out in a `Grid`. For more information about creating the user interface, see [User interface](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `AboutPage.xaml` by pressing **CTRL+S**.

16. In **Solution Explorer**, in the **Notes** project, double-click `AboutPage.xaml.cs` to open it:



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 using Xamarin.Forms;
8 using Xamarin.Forms.Xaml;
9
10 namespace Notes.Views
11 {
12     [XamlCompilation(XamlCompilationOptions.Compile)]
13     public partial class AboutPage : ContentPage
14     {
15         public AboutPage()
16         {
17             InitializeComponent();
18         }
19     }
20 }
```

17. In `AboutPage.xaml.cs`, remove all of the template code and replace it with the following code:

```
using System;
using Xamarin.Essentials;
using Xamarin.Forms;

namespace Notes.Views
{
    public partial class AboutPage : ContentPage
    {
        public AboutPage()
        {
            InitializeComponent();
        }

        async void OnButtonClicked(object sender, EventArgs e)
        {
            // Launch the specified URL in the system browser.
            await Launcher.OpenAsync("https://aka.ms/xamarin-quickstart");
        }
    }
}
```

This code defines the `OnButtonClicked` event handler, which is executed when the **Learn more** `Button` is pressed. When the button is pressed, a web browser is launched and the page represented by the URI argument to the `OpenAsync` method is displayed. For more information about user interaction, see [Responding to user interaction](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `AboutPage.xaml.cs` by pressing **CTRL+S**.

18. In **Solution Explorer**, in the **Notes** project, double-click `AppShell.xaml` to open it.

The screenshot shows the Visual Studio interface with the XAML editor open. The title bar says "Notes.Android". The code editor displays the following XAML code:

```
<?xml version="1.0" encoding="UTF-8"?>
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:local="clr-namespace:Notes.Views"
       Title="Notes"
       x:Class="Notes.AppShell">

    <!--
        The overall app visual hierarchy is defined here, along with navigation.
        https://docs.microsoft.com/xamarin/xamarin-forms/app-fundamentals/shell/
    -->

    <Shell.Resources>
        <ResourceDictionary>
            <Style x:Key="BaseStyle" TargetType="Element">
                <Setter Property="Shell.BackgroundColor" Value="{StaticResource Primary}" />
                <Setter Property="Shell.ForegroundColor" Value="White" />
                <Setter Property="Shell.TitleColor" Value="White" />
                <Setter Property="Shell.DisabledColor" Value="#B4FFFFFF" />
                <Setter Property="Shell.UnselectedColor" Value="#95FFFFFF" />
                <Setter Property="Shell.TabBarBackgroundColor" Value="{StaticResource Primary}" />
                <Setter Property="Shell.TabBarForegroundColor" Value="White"/>
                <Setter Property="Shell.TabBarUnselectedColor" Value="#95FFFFFF"/>
                <Setter Property="Shell.TabBarTitleColor" Value="White"/>
            </Style>
        </ResourceDictionary>
    </Shell.Resources>

```

19. In AppShell.xaml, remove all of the template code and replace it with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Notes.Views"
       x:Class="Notes.AppShell">
    <!-- Display a bottom tab bar containing two tabs -->
    <TabBar>
        <ShellContent Title="Notes"
                      Icon="icon_feed.png"
                      ContentTemplate="{DataTemplate views:NotesPage}" />
        <ShellContent Title="About"
                      Icon="icon_about.png"
                      ContentTemplate="{DataTemplate views:AboutPage}" />
    </TabBar>
</Shell>
```

This code declaratively defines the visual hierarchy of the application, which consists of a `TabBar` containing two `ShellContent` objects. These objects don't represent any user interface elements, but rather the organization of the application's visual hierarchy. Shell will take these objects and produce the user interface for the content. For more information about creating the user interface, see [User interface](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `AppShell.xaml` by pressing **CTRL+S**.

20. In **Solution Explorer**, in the **Notes** project, expand `AppShell.xaml`, and double-click `AppShell.xaml.cs` to open it.

The screenshot shows the Visual Studio IDE with the 'Notes' project selected. The current file is 'AppShell.xaml.cs'. The code editor displays the following C# code:

```
1  using Notes.ViewModels;
2  using Notes.Views;
3  using System;
4  using System.Collections.Generic;
5  using Xamarin.Forms;
6
7  namespace Notes
8  {
9      public partial class AppShell : Xamarin.Forms.Shell
10     {
11         public AppShell()
12         {
13             InitializeComponent();
14             Routing.RegisterRoute(nameof(ItemDetailPage), typeof(ItemDetailPage));
15             Routing.RegisterRoute(nameof(NewItemPage), typeof(NewItemPage));
16         }
17     }
18 }
19 }
```

21. In **AppShell.xaml.cs**, remove all of the template code and replace it with the following code:

```
using Xamarin.Forms;

namespace Notes
{
    public partial class AppShell : Shell
    {
        public AppShell()
        {
            InitializeComponent();
        }
    }
}
```

Save the changes to **AppShell.xaml.cs** by pressing **CTRL+S**.

22. In **Solution Explorer**, in the **Notes** project, double-click **App.xaml** to open it:

The screenshot shows the Visual Studio IDE with the 'Notes' project selected. The current file is 'App.xaml'. The code editor displays the following XAML code:

```
<?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="Notes.App">
    <!--
        Define global resources and styles here, that apply to all pages in your app.
    -->
    <Application.Resources>
        <ResourceDictionary>
            <Color x:Key="Primary">#2196F3</Color>
            <Style TargetType="Button">
                <Setter Property="TextColor" Value="White"></Setter>
                <Setter Property="VisualStateManager.VisualStateGroups">
                    <VisualStateGroupList>
                        <VisualStateGroup x:Name="CommonStates">
                            <VisualState x:Name="Normal">
                                <VisualState.Setters>
                                    <Setter Property="BackgroundColor" Value="{StaticResource Primary}" />
                                </VisualState.Setters>
                            </VisualState>
                        </VisualStateGroup>
                    </VisualStateGroupList>
                </Setter>
            </Style>
        </ResourceDictionary>
    </Application.Resources>

```

23. In **App.xaml**, remove all of the template code and replace it with the following code:

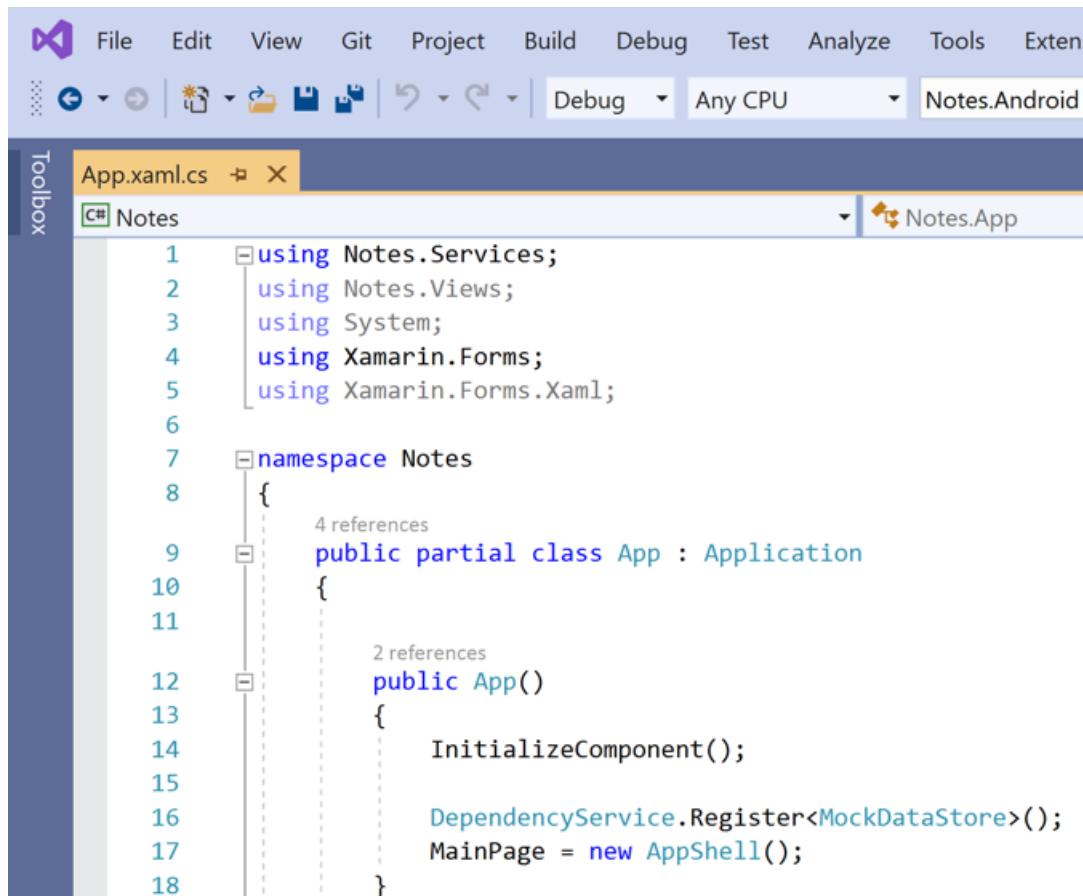
```
<?xml version="1.0" encoding="utf-8" ?>
<Application xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Notes.App">

</Application>
```

This code declaratively defines an `App` class, which is responsible for instantiating the application.

Save the changes to `App.xaml` by pressing **CTRL+S**.

24. In **Solution Explorer**, in the **Notes** project, expand `App.xaml`, and double-click `App.xaml.cs` to open it:



25. In `App.xaml.cs`, remove all of the template code and replace it with the following code:

```

using Xamarin.Forms;

namespace Notes
{
    public partial class App : Application
    {

        public App()
        {
            InitializeComponent();
            MainPage = new AppShell();
        }

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

        protected override void OnResume()
        {
        }
    }
}

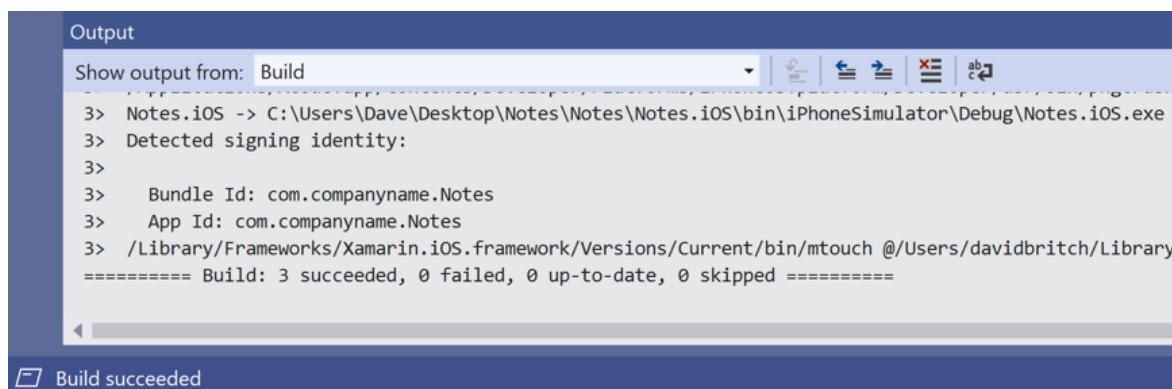
```

This code defines the code-behind for the `App` class, that is responsible for instantiating the application. It initializes the `MainPage` property to the subclassed `Shell` object.

Save the changes to `App.xaml.cs` by pressing **CTRL+S**.

Building the quickstart

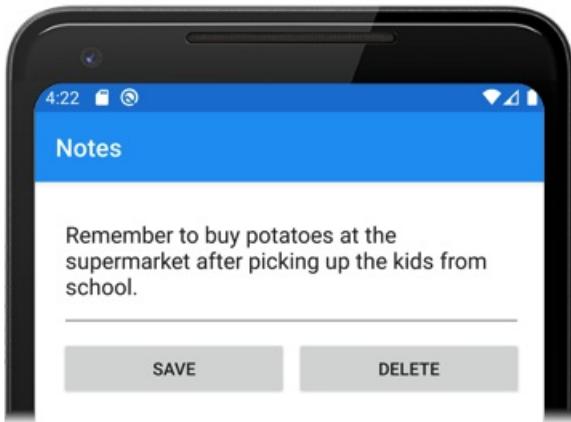
1. In Visual Studio, select the **Build > Build Solution** menu item (or press F6). The solution will build and a success message will appear in the Visual Studio status bar:



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

2. In the Visual Studio toolbar, press the **Start** button (the triangular button that resembles a Play button) to launch the application in your chosen Android emulator:





Enter a note and press the **Save** button. Then, close the application and re-launch it to ensure the note you entered is reloaded.

Press the **About** tab icon to navigate to the `AboutPage`:



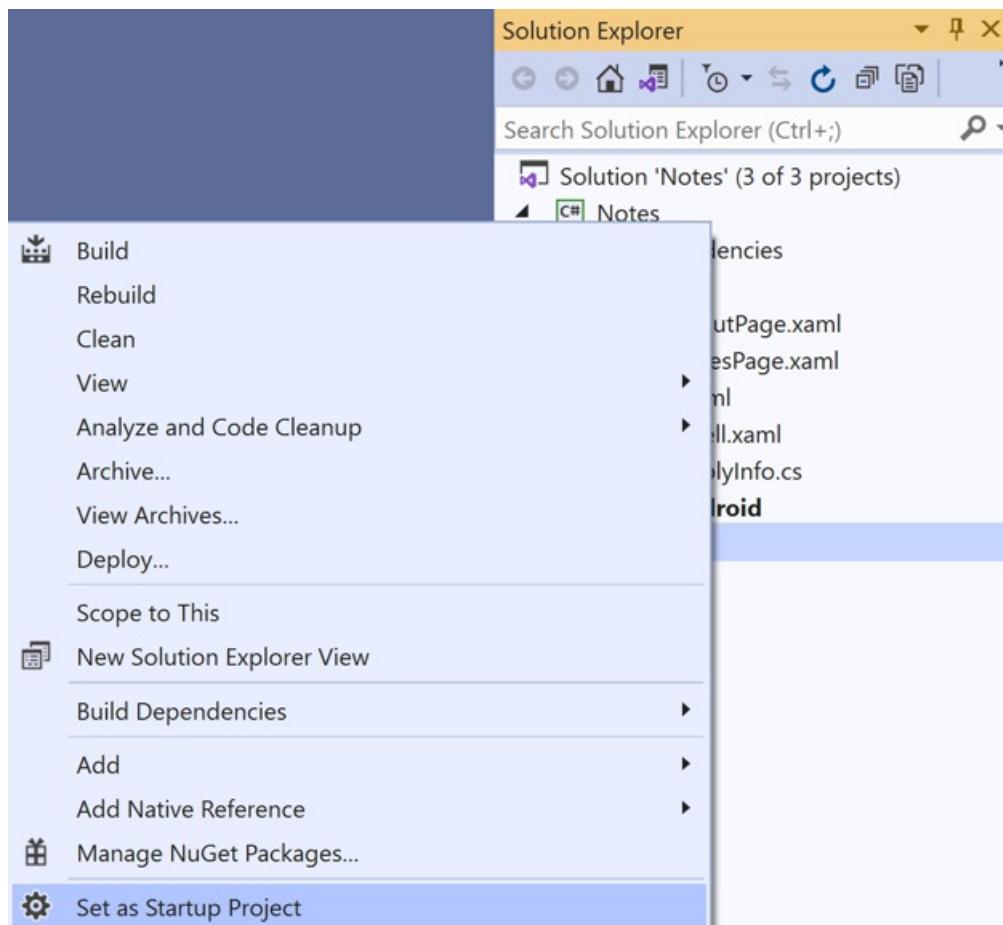
Press the **Learn more** button to launch the quickstarts web page.

For more information about how the application is launched on each platform, see [Launching the application on each platform](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

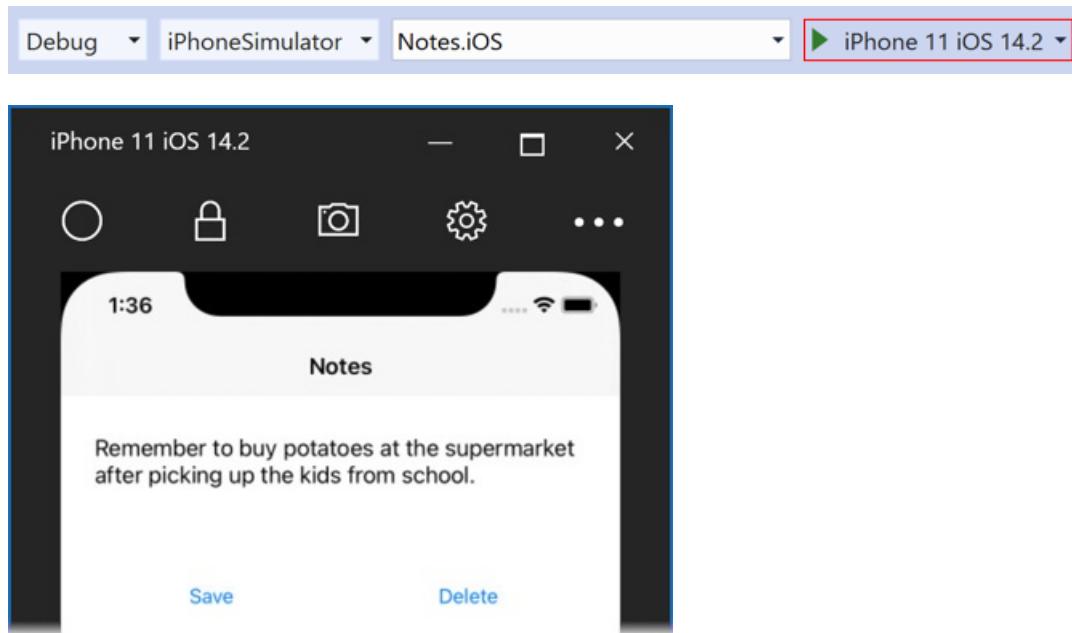
NOTE

The following steps should only be carried out if you have a [paired Mac](#) that meets the system requirements for Xamarin.Forms development.

3. In the Visual Studio toolbar, right-click on the `Notes.iOS` project, and select **Set as StartUp Project**.



4. In the Visual Studio toolbar, press the **Start** button (the triangular button that resembles a Play button) to launch the application in your chosen [iOS remote simulator](#):



Enter a note and press the **Save** button. Then, close the application and re-launch it to ensure the note you entered is reloaded.

Press the **About** tab icon to navigate to the [AboutPage](#):



Press the **Learn more** button to launch the quickstarts web page.

For more information about how the application is launched on each platform, see [Launching the application on each platform](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

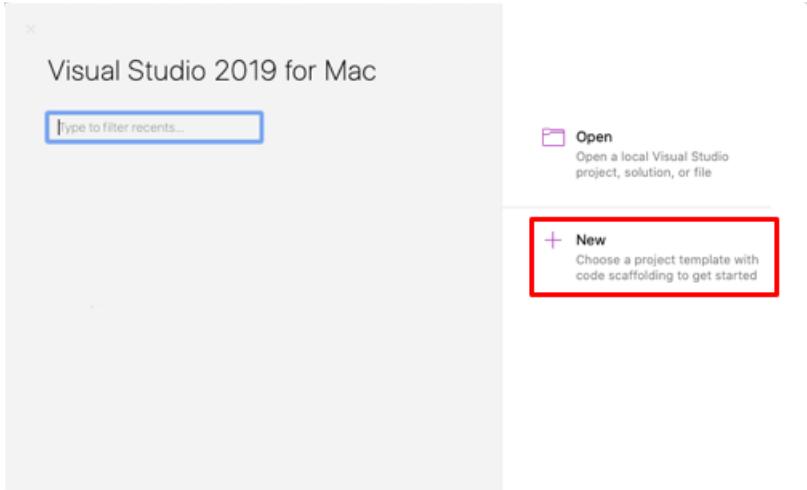
Prerequisites

- Visual Studio for Mac (latest release), with iOS and Android platform support installed.
- Xcode (latest release).
- Knowledge of C#.

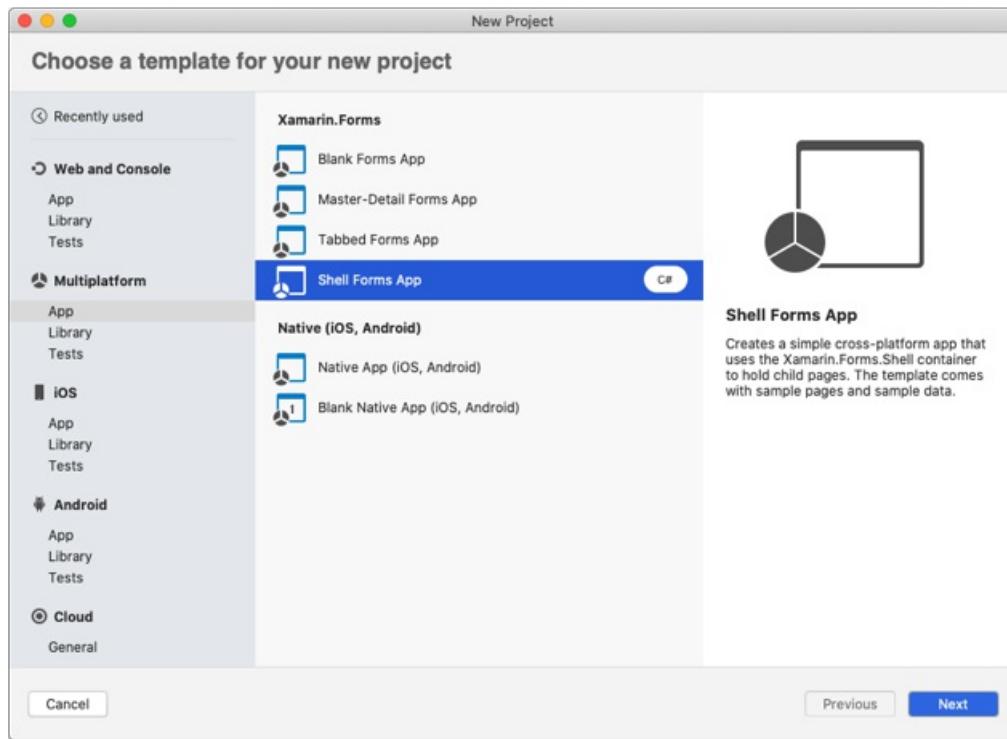
For more information about these prerequisites, see [Installing Xamarin](#).

Get started with Visual Studio for Mac

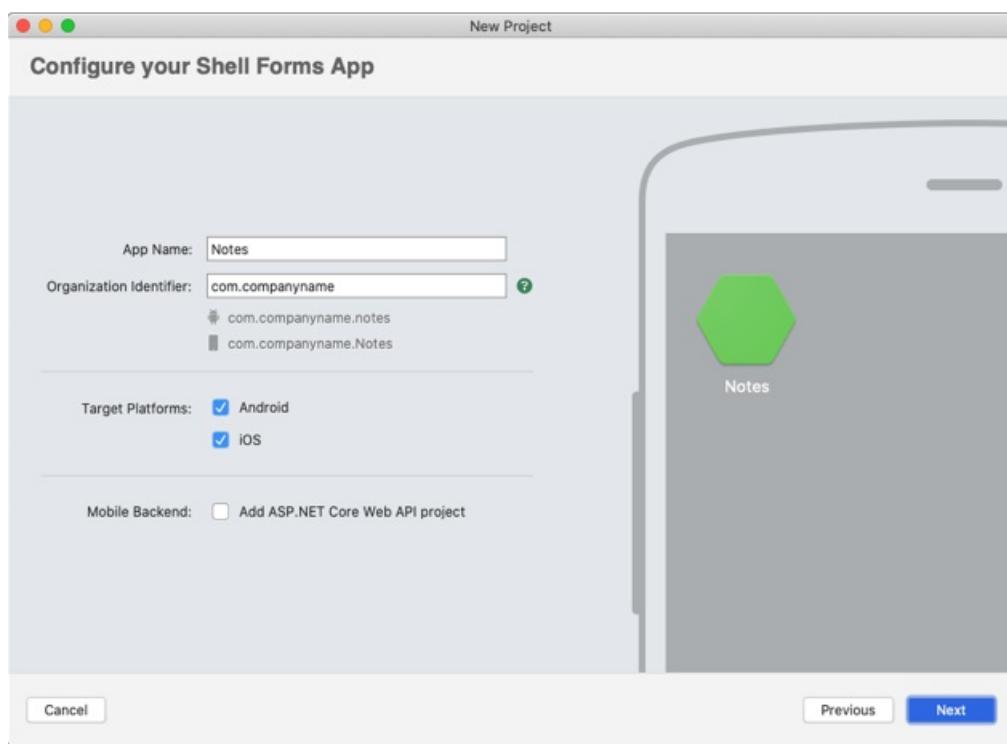
1. Launch Visual Studio for Mac, and in the start window click **New** to create a new project:



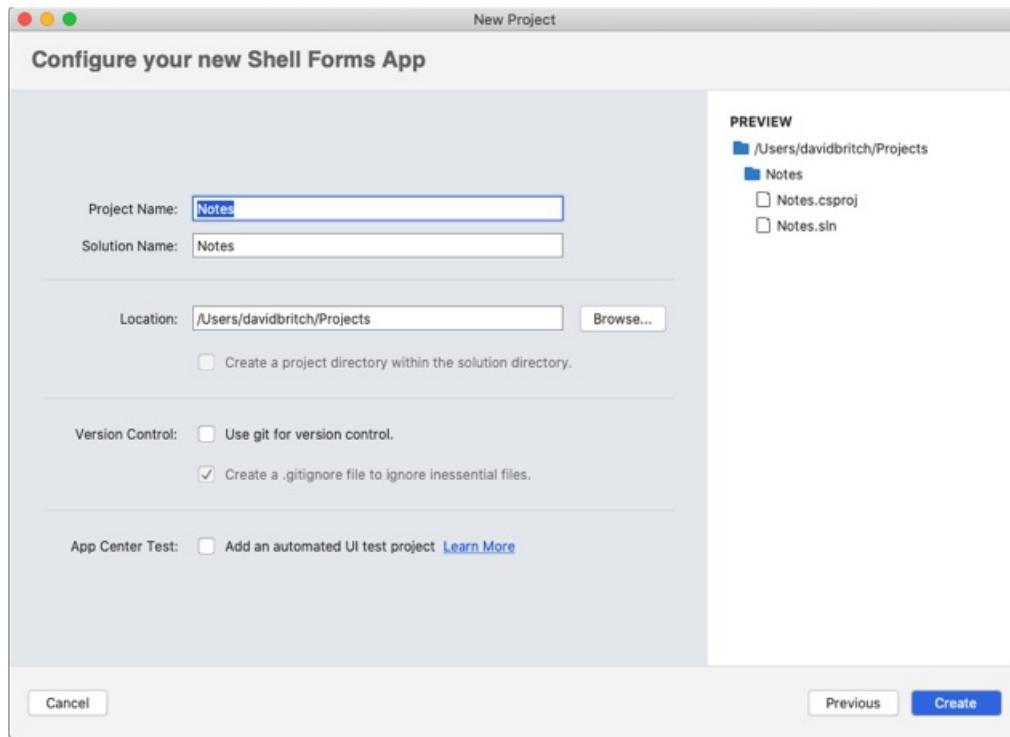
2. In the **Choose a template for your new project** dialog, click **Multiplatform > App**, select the **Shell Forms App** template, and click the **Next** button:



3. In the **Configure your Shell Forms app** dialog, name the new app **Notes**, and click the **Next** button:



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

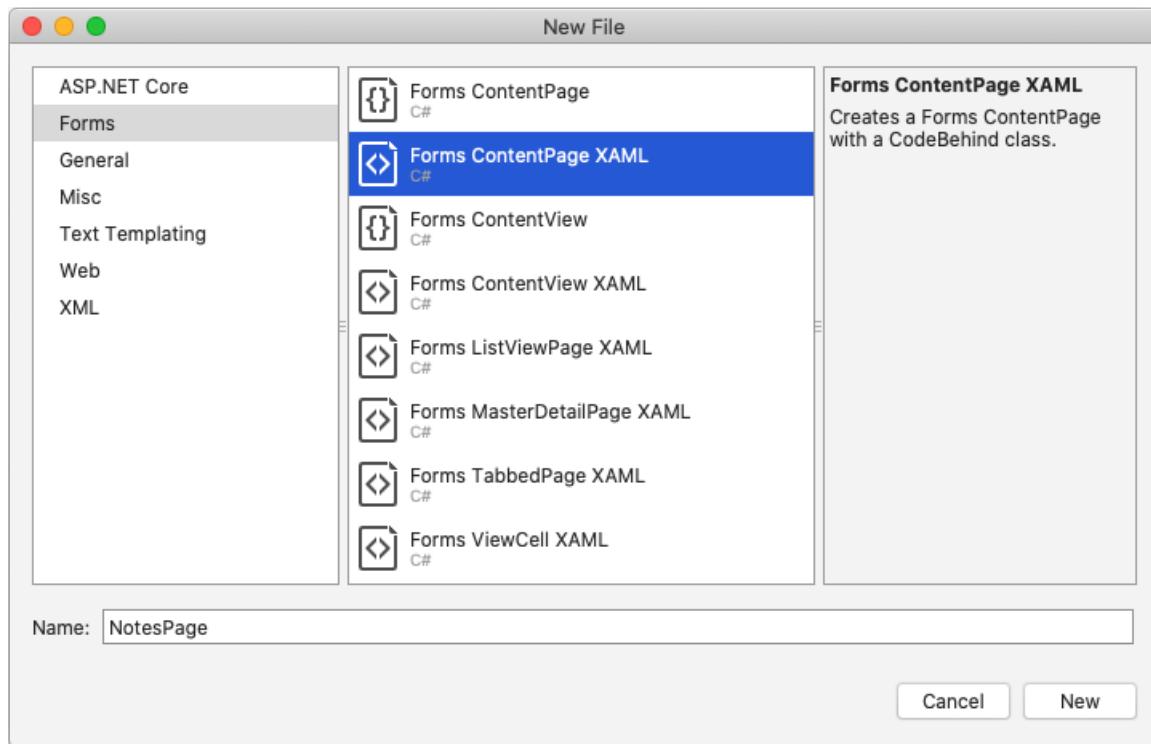


IMPORTANT

The C# and XAML snippets in this quickstart requires that the solution and project are both named **Notes**. Using a different name will result in build errors when you copy code from this quickstart into the project.

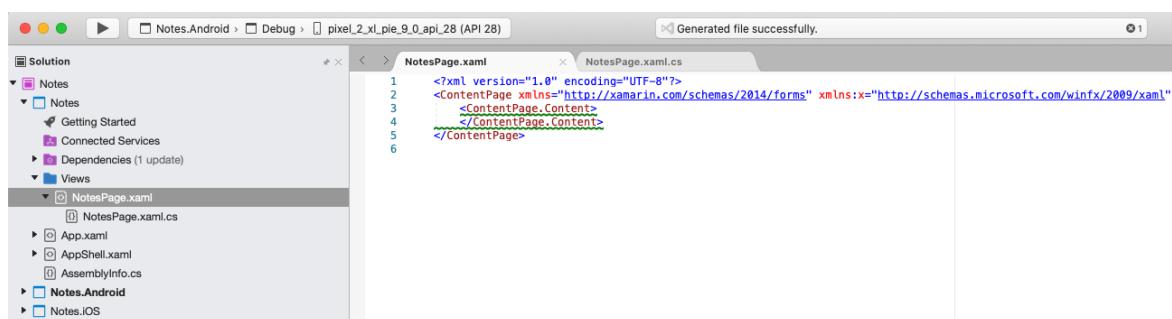
For more information about the .NET Standard library that gets created, see [Anatomy of a Xamarin.Forms Shell application](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

5. In the **Solution Pad**, in the **Notes** project, delete the following folders (and their contents):
 - **Models**
 - **Services**
 - **ViewModels**
 - **Views**
6. In the **Solution Pad**, in the **Notes** project, delete **GettingStarted.txt**.
7. In the **Solution Pad**, in the **Notes** project, add a new folder named **Views**.
8. In the **Solution Pad**, in the **Notes** project, select the **Views** folder, right-click, and select **Add > New File....** In the **New File** dialog, select **Forms > Forms ContentPage XAML**, name the new file **NotesPage**, and click the **New** button:



This will add a new page named **NotesPage** to the **Views** folder. This page will be the main page in the application.

- In the **Solution Pad**, in the **Notes** project, double-click **NotesPage.xaml** to open it:



- In **NotesPage.xaml**, remove all of the template code and replace it with the following code:

```

<?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="Notes.Views.NotesPage"
              Title="Notes">
    <!-- Layout children vertically -->
    <StackLayout Margin="20">
        <Editor x:Name="editor"
               Placeholder="Enter your note"
               HeightRequest="100" />
        <!-- Layout children in two columns -->
        <Grid ColumnDefinitions="*,*">
            <Button Text="Save"
                   Clicked="OnSaveButtonClicked" />
            <Button Grid.Column="1"
                   Text="Delete"
                   Clicked="OnDeleteButtonClicked"/>
        </Grid>
    </StackLayout>
</ContentPage>

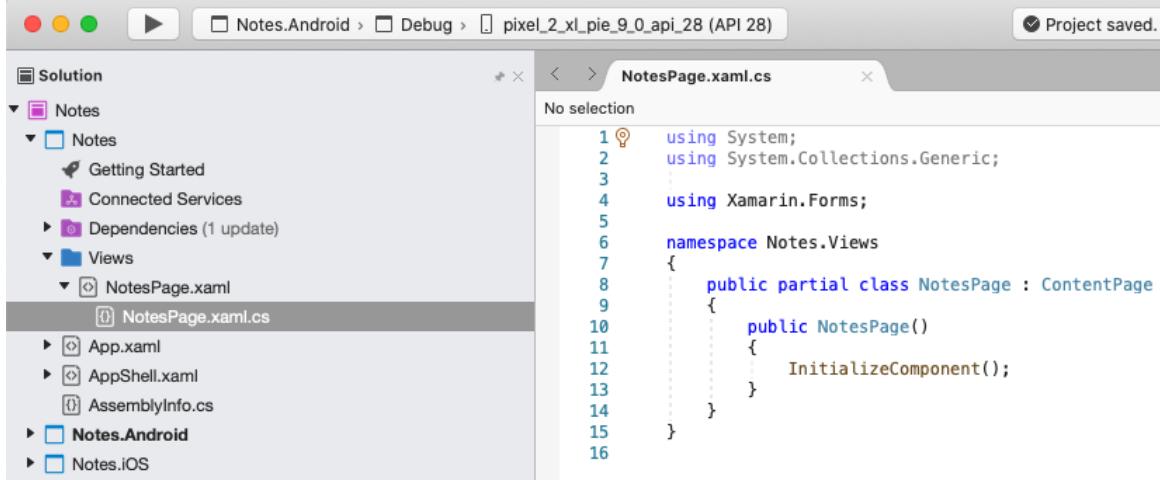
```

This code declaratively defines the user interface for the page, which consists of an **Editor** for text input,

and two `Button` objects that direct the application to save or delete a file. The two `Button` objects are horizontally laid out in a `Grid`, with the `Editor` and `Grid` being vertically laid out in a `StackLayout`. For more information about creating the user interface, see [User interface](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `NotesPage.xaml` by choosing **File > Save** (or by pressing **⌘ + S**).

11. In the **Solution Pad**, in the **Notes** project, double-click `NotesPage.xaml.cs` to open it:



The screenshot shows the Xamarin Studio interface. The title bar indicates the project is "Notes.Android" and the build configuration is "Debug". A "Project saved." message is shown in the top right. The left pane is the "Solution Pad" with a tree view of the project structure:

- Notes (selected)
- Notes
- Getting Started
- Connected Services
- Dependencies (1 update)
- Views
- NotesPage.xaml (selected)
- NotesPage.xaml.cs (highlighted)
- App.xaml
- AppShell.xaml
- AssemblyInfo.cs
- Notes.Android
- Notes.iOS

The right pane is the "NotesPage.xaml.cs" code editor. It contains the following C# code:

```
1 using System;
2 using System.Collections.Generic;
3 
4 using Xamarin.Forms;
5 
6 namespace Notes.Views
7 {
8     public partial class NotesPage : ContentPage
9     {
10         public NotesPage()
11         {
12             InitializeComponent();
13         }
14     }
15 }
16 
```

12. In `NotesPage.xaml.cs`, remove all of the template code and replace it with the following code:

```

using System;
using System.IO;
using Xamarin.Forms;

namespace Notes.Views
{
    public partial class NotesPage : ContentPage
    {
        string _fileName =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "notes.txt");

        public NotesPage()
        {
            InitializeComponent();

            // Read the file.
            if (File.Exists(_fileName))
            {
                editor.Text = File.ReadAllText(_fileName);
            }
        }

        void OnSaveButtonClicked(object sender, EventArgs e)
        {
            // Save the file.
            File.WriteAllText(_fileName, editor.Text);
        }

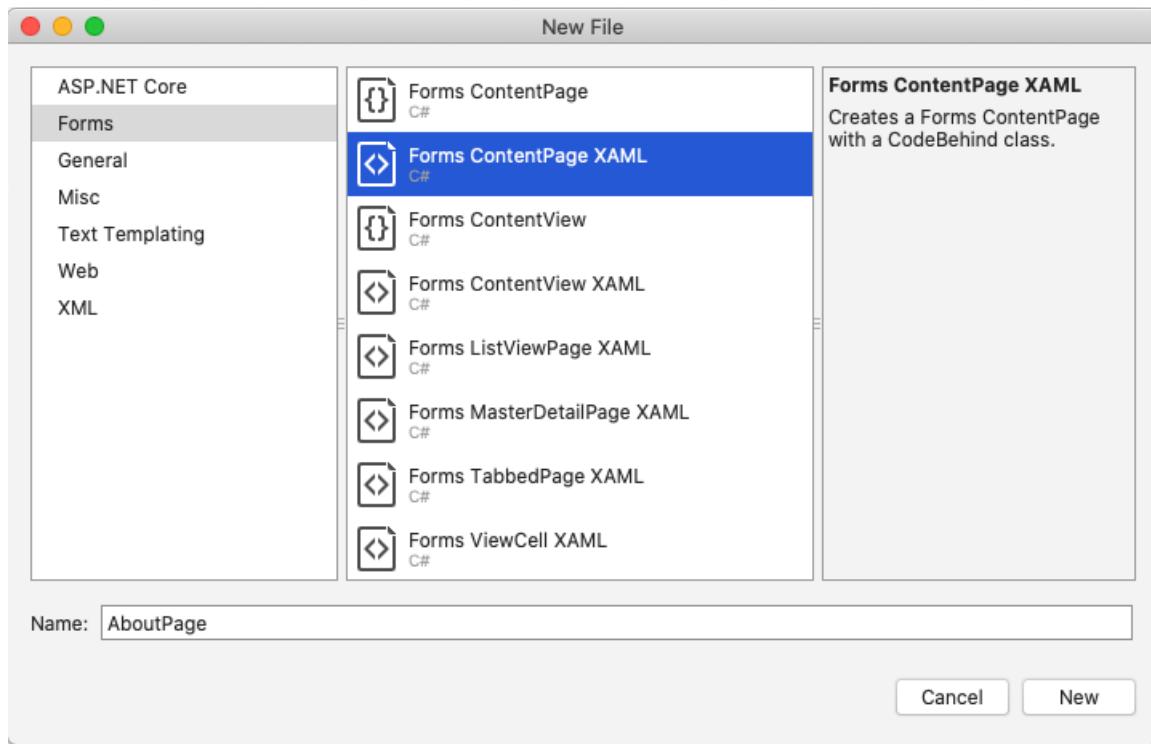
        void OnDeleteButtonClicked(object sender, EventArgs e)
        {
            // Delete the file.
            if (File.Exists(_fileName))
            {
                File.Delete(_fileName);
            }
            editor.Text = string.Empty;
        }
    }
}

```

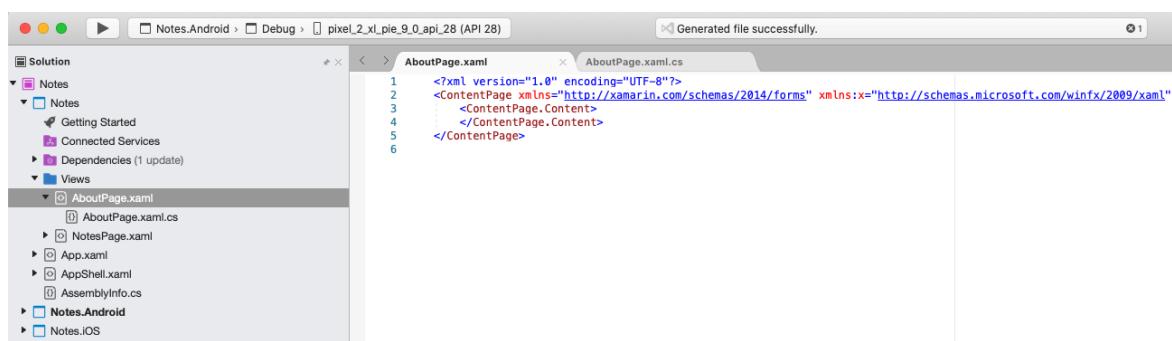
This code defines a `_fileName` field, which references a file named `notes.txt` that will store note data in the local application data folder for the application. When the page constructor is executed the file is read, if it exists, and displayed in the `Editor`. When the `Save` `Button` is pressed the `OnSaveButtonClicked` event handler is executed, which saves the content of the `Editor` to the file. When the `Delete` `Button` is pressed the `OnDeleteButtonClicked` event handler is executed, which deletes the file, provided that it exists, and removes any text from the `Editor`. For more information about user interaction, see [Responding to user interaction](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `NotesPage.xaml.cs` by choosing `File > Save` (or by pressing `⌘ + S`).

13. In the **Solution Pad**, in the **Notes** project, select the **Views** folder, right-click, and select **Add > New File....** In the **New File** dialog, select **Forms > Forms ContentPage XAML**, name the new file **AboutPage**, and click the **New** button:



14. In the Solution Pad, in the Notes project, double-click **AboutPage.xaml** to open it:



This will add a new page named **AboutPage** to the **Views** folder.

15. In **AboutPage.xaml**, remove all of the template code and replace it with the following code:

```

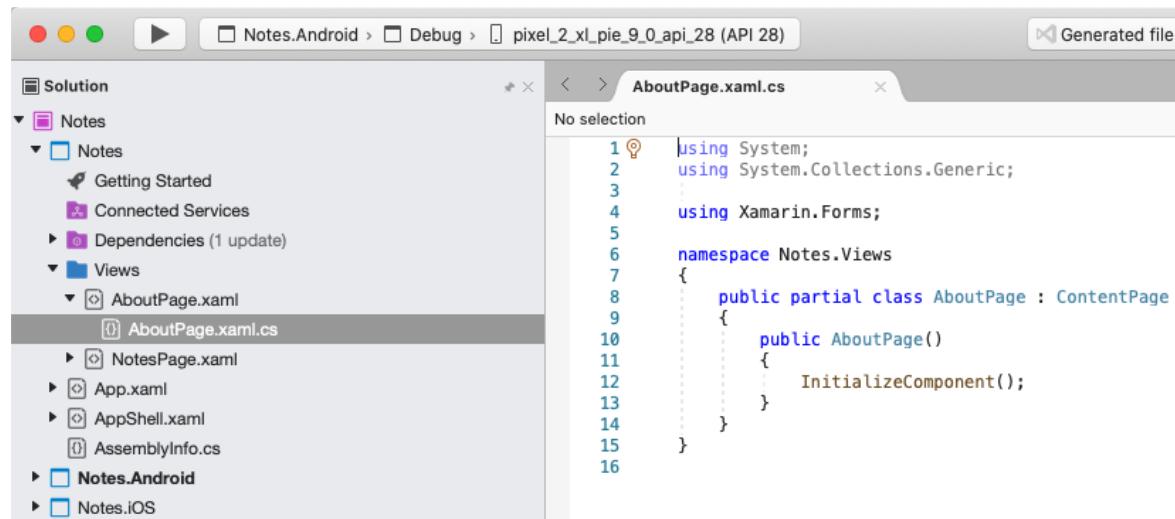
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Notes.Views.AboutPage"
    Title="About">
    <!-- Layout children in two rows -->
    <Grid RowDefinitions="Auto,*">
        <Image Source="xamarin_logo.png"
            BackgroundColor="{OnPlatform iOS=LightSlateGray, Android=#2196F3}"
            VerticalOptions="Center"
            HeightRequest="64" />
        <!-- Layout children vertically -->
        <StackLayout Grid.Row="1"
            Margin="20"
            Spacing="20">
            <Label FontSize="22">
                <Label.FormattedText>
                    <FormattedString>
                        <FormattedString.Spans>
                            <Span Text="Notes"
                                FontAttributes="Bold"
                                FontSize="22" />
                            <Span Text=" v1.0" />
                        </FormattedString.Spans>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
            <Label Text="This app is written in XAML and C# with the Xamarin Platform." />
            <Button Text="Learn more"
                Clicked="OnButtonClicked" />
        </StackLayout>
    </Grid>
</ContentPage>

```

This code declaratively defines the user interface for the page, which consists of an `Image`, two `Label` objects that display text, and a `Button`. The two `Label` objects and `Button` are vertically laid out in a `StackLayout`, with the `Image` and `StackLayout` being vertically laid out in a `Grid`. For more information about creating the user interface, see [User interface](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `AboutPage.xaml` by choosing **File > Save** (or by pressing **⌘ + S**).

- In the **Solution Pad**, in the **Notes** project, double-click `AboutPage.xaml.cs` to open it:



- In `AboutPage.xaml.cs`, remove all of the template code and replace it with the following code:

```

using System;
using Xamarin.Essentials;
using Xamarin.Forms;

namespace Notes.Views
{
    public partial class AboutPage : ContentPage
    {
        public AboutPage()
        {
            InitializeComponent();
        }

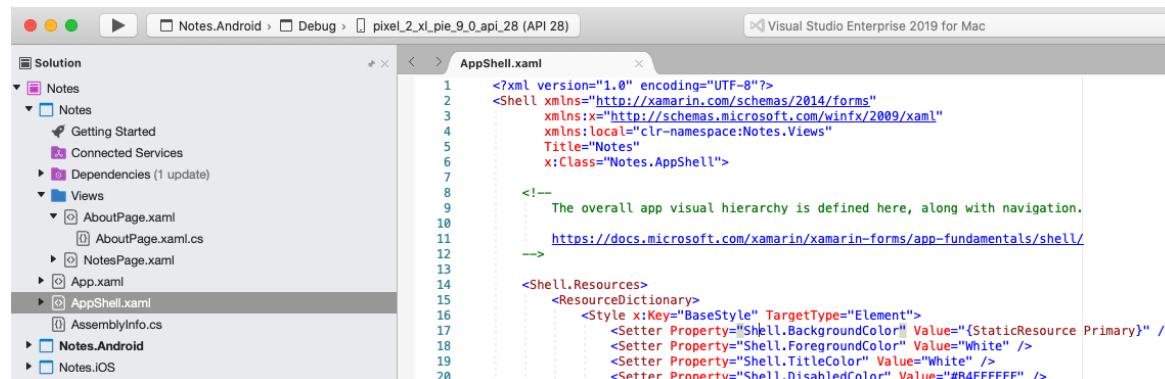
        async void OnButtonClicked(object sender, EventArgs e)
        {
            // Launch the specified URL in the system browser.
            await Launcher.OpenAsync("https://aka.ms/xamarin-quickstart");
        }
    }
}

```

This code defines the `OnButtonClicked` event handler, which is executed when the **Learn more** [Button](#) is pressed. When the button is pressed, a web browser is launched and the page represented by the URI argument to the `OpenAsync` method is displayed. For more information about user interaction, see [Responding to user interaction](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `AboutPage.xaml.cs` by choosing **File > Save** (or by pressing **⌘ + S**).

- In the Solution Pad, in the **Notes** project, double-click `AppShell.xaml` to open it:



- In `AppShell.xaml`, remove all of the template code and replace it with the following code:

```

<?xml version="1.0" encoding="UTF-8"?>
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Notes.Views"
       x:Class="Notes.AppShell">
    <!-- Display a bottom tab bar containing two tabs -->
    <TabBar>
        <ShellContent Title="Notes"
                      Icon="icon_feed.png"
                      ContentTemplate="{DataTemplate views:NotesPage}" />
        <ShellContent Title="About"
                      Icon="icon_about.png"
                      ContentTemplate="{DataTemplate views:AboutPage}" />
    </TabBar>
</Shell>

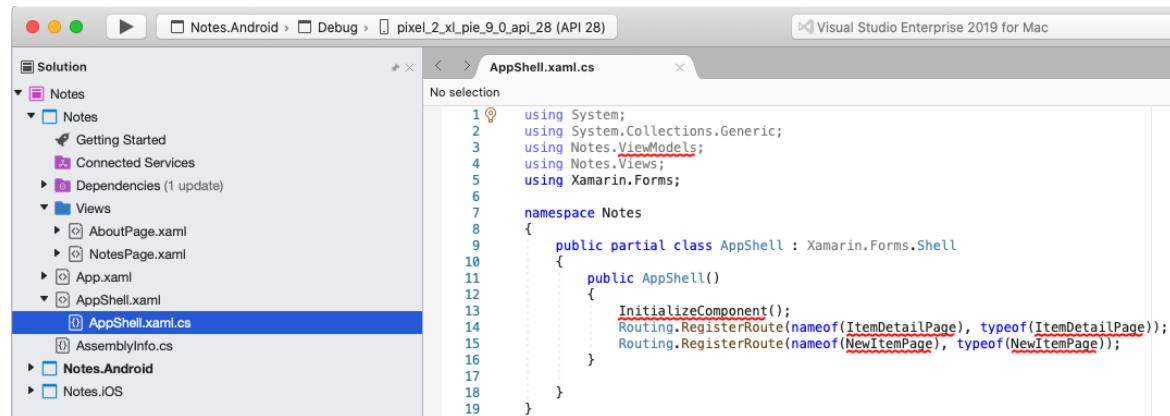
```

This code declaratively defines the visual hierarchy of the application, which consists of a [TabBar](#)

containing two `ShellContent` objects. These objects don't represent any user interface elements, but rather the organization of the application's visual hierarchy. Shell will take these objects and produce the user interface for the content. For more information about creating the user interface, see [User interface](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `AppShell.xaml` by choosing **File > Save** (or by pressing **⌘ + S**).

20. In the **Solution Pad**, in the **Notes** project, expand `AppShell.xaml`, and double-click `AppShell.xaml.cs` to open it:



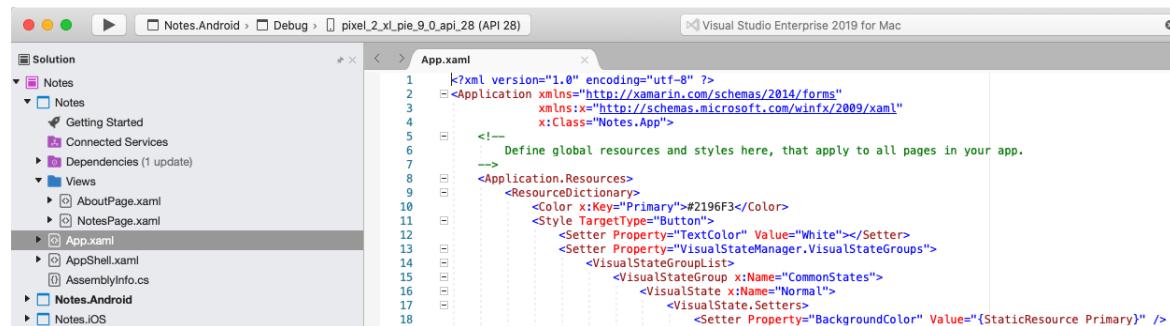
21. In `AppShell.xaml.cs`, remove all of the template code and replace it with the following code:

```
using Xamarin.Forms;

namespace Notes
{
    public partial class AppShell : Shell
    {
        public AppShell()
        {
            InitializeComponent();
        }
    }
}
```

Save the changes to `AppShell.xaml.cs` by choosing **File > Save** (or by pressing **⌘ + S**).

22. In the **Solution Pad**, in the **Notes** project, double-click `App.xaml` to open it:



23. In `App.xaml`, remove all of the template code and replace it with the following code:

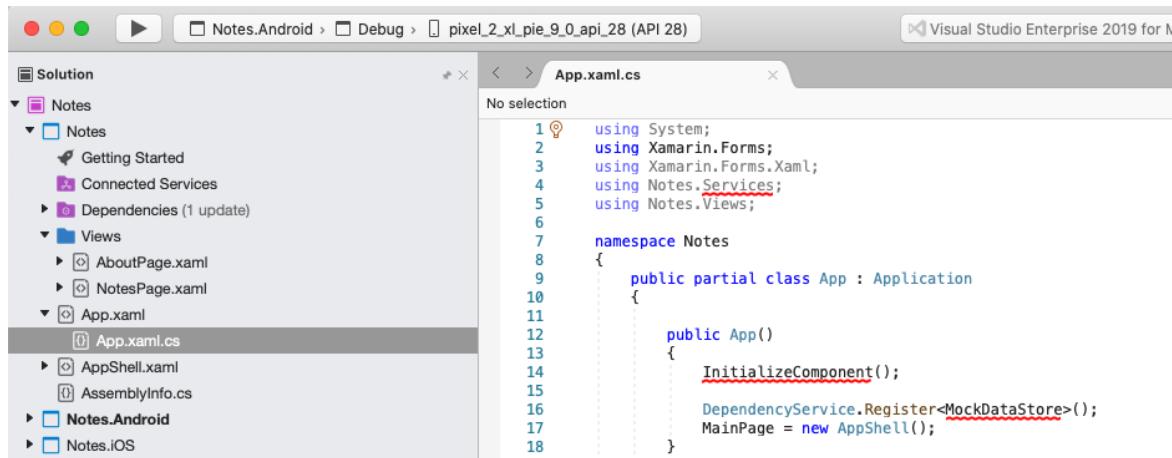
```
<?xml version="1.0" encoding="utf-8" ?>
<Application xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Notes.App">

</Application>
```

This code declaratively defines an `App` class, which is responsible for instantiating the application.

Save the changes to `App.xaml` by choosing **File > Save** (or by pressing **⌘ + S**).

24. In the **Solution Pad**, in the **Notes** project, expand `App.xaml`, and double-click `App.xaml.cs` to open it:



25. In `App.xaml.cs`, remove all of the template code and replace it with the following code:

```
using Xamarin.Forms;

namespace Notes
{
    public partial class App : Application
    {

        public App()
        {
            InitializeComponent();
            MainPage = new AppShell();
        }

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

        protected override void OnResume()
        {
        }
    }
}
```

This code defines the code-behind for the `App` class, that is responsible for instantiating the application.

It initializes the `MainPage` property to the subclassed `Shell` object.

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

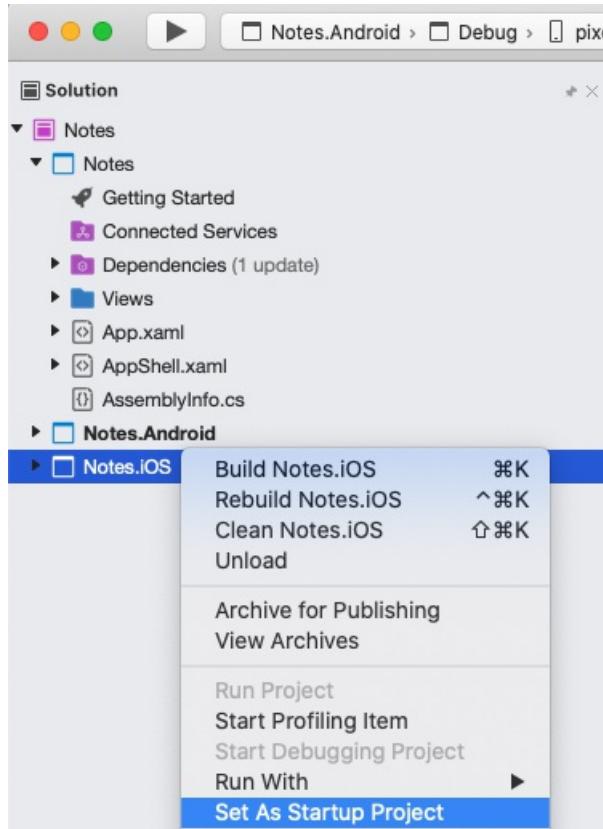
Building the quickstart

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

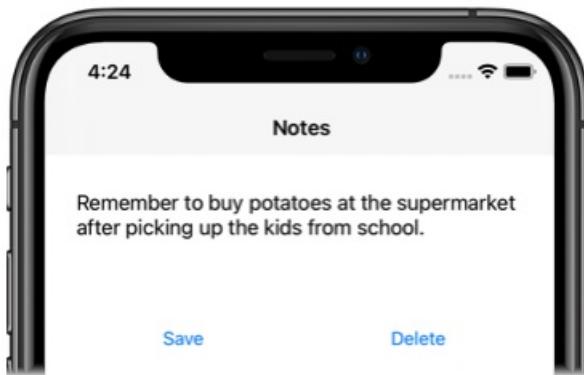


If there are errors, repeat the previous steps and correct any mistakes until the projects build successfully.

2. In the Solution Pad, select the Notes.iOS project, right-click, and select Set As Startup Project:



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



Enter a note and press the **Save** button. Then, close the application and re-launch it to ensure the note you entered is reloaded.

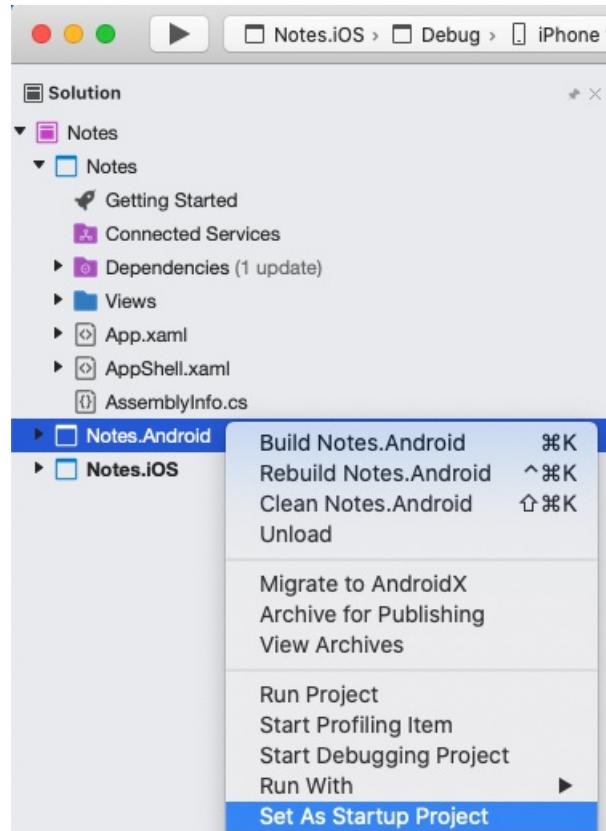
Press the **About** tab icon to navigate to the `AboutPage`:



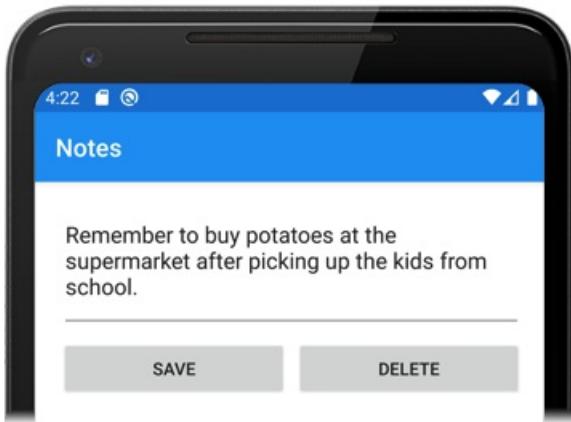
Press the **Learn more** button to launch the quickstarts web page.

For more information about how the application is launched on each platform, see [Launching the application on each platform](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

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



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



Enter a note and press the **Save** button. Then, close the application and re-launch it to ensure the note you entered is reloaded.

Press the **About** tab icon to navigate to the `AboutPage`:



Press the **Learn more** button to launch the quickstarts web page.

For more information about how the application is launched on each platform, see [Launching the application on each platform](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

Next steps

In this quickstart, you learned how to:

- Create a Xamarin.Forms Shell application.
- Define the user interface for a page using eXtensible Application Markup Language (XAML), and interact with XAML elements from code.
- Describe the visual hierarchy of a Shell application by subclassing the `Shell` class.

Continue to the next quickstart to add additional pages to this Xamarin.Forms Shell application.

[Next](#)

Related links

- [Notes \(sample\)](#)
- [Xamarin.Forms Shell quickstart deep dive](#)

Perform navigation in a Xamarin.Forms application

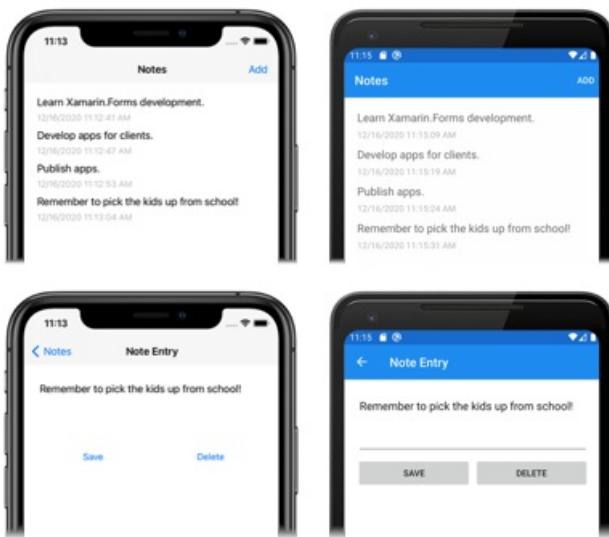
8/4/2022 • 17 minutes to read • [Edit Online](#)

 [Download the sample](#)

In this quickstart, you will learn how to:

- Add additional pages to a Xamarin.Forms Shell application.
- Perform navigation between pages.
- Use data binding to synchronize data between user interface elements and their data source.

The quickstart walks through how to turn a cross-platform Xamarin.Forms Shell application, capable of storing a single note, into an application capable of storing multiple notes. The final application is shown below:

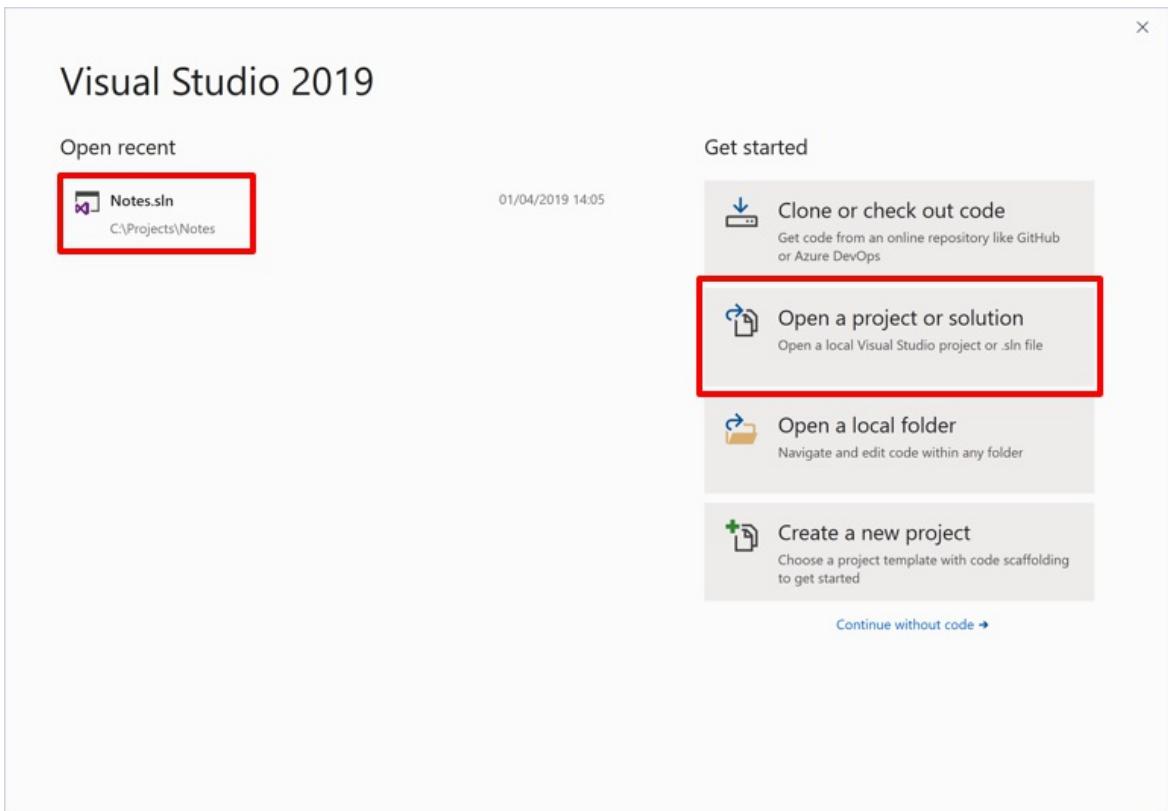


Prerequisites

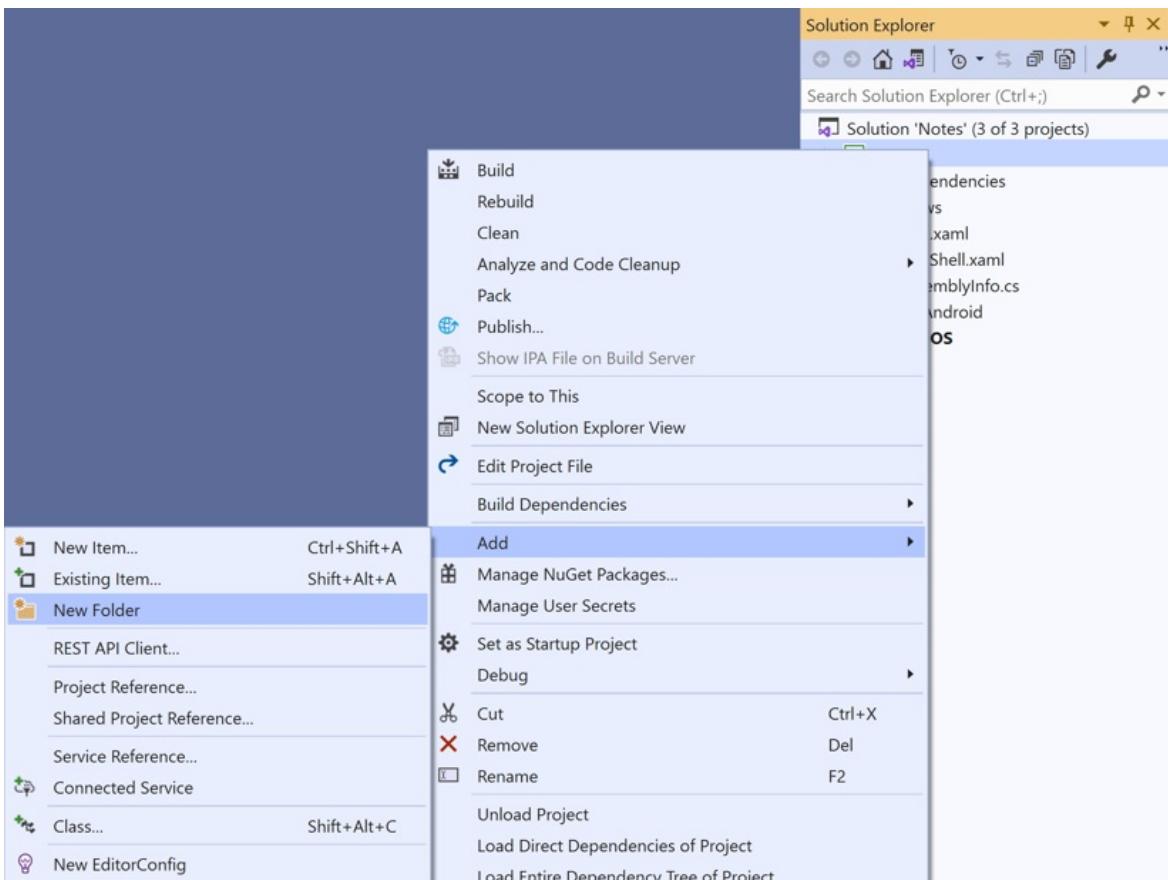
You should successfully complete the [previous quickstart](#) before attempting this quickstart. Alternatively, download the [previous quickstart sample](#) and use it as the starting point for this quickstart.

Update the app with Visual Studio

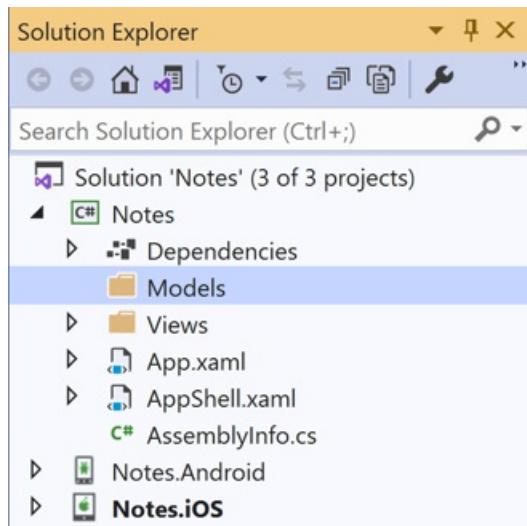
1. Launch Visual Studio. In the start window, click the **Notes** solution in the recent projects/solutions list, or click **Open a project or solution**, and in the **Open Project/Solution** dialog select the solution file for the Notes project:



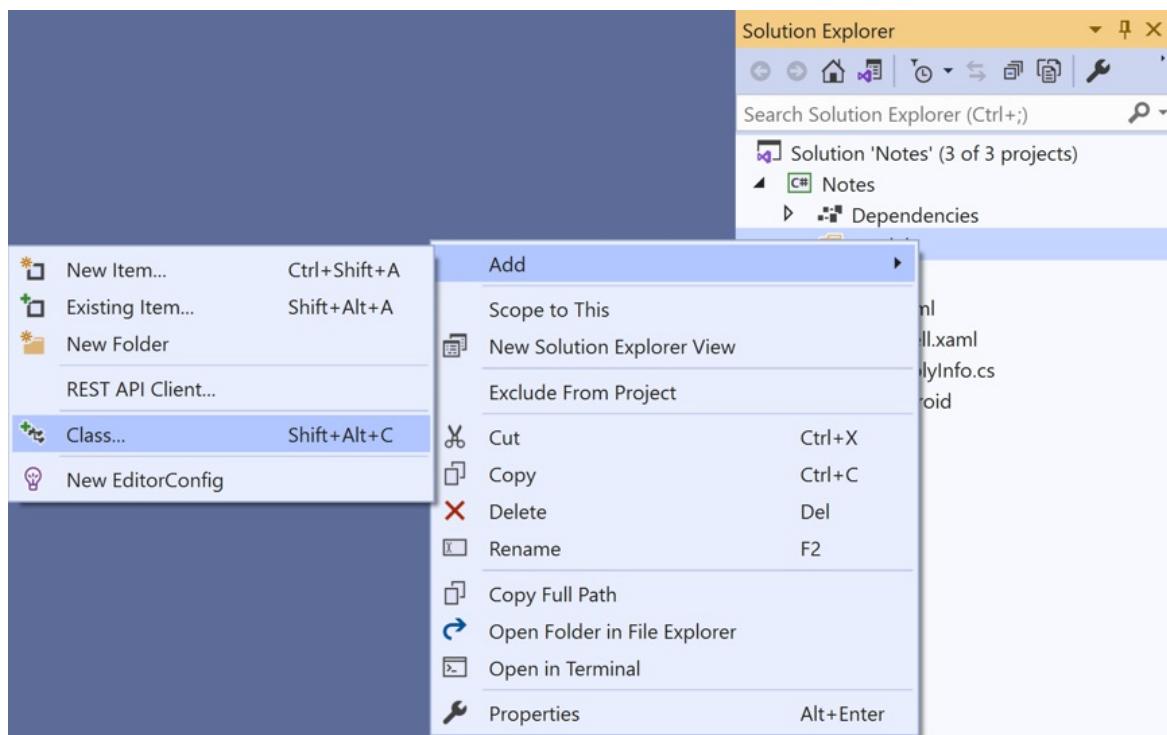
2. In Solution Explorer, right-click on the **Notes** project, and select **Add > New Folder**:



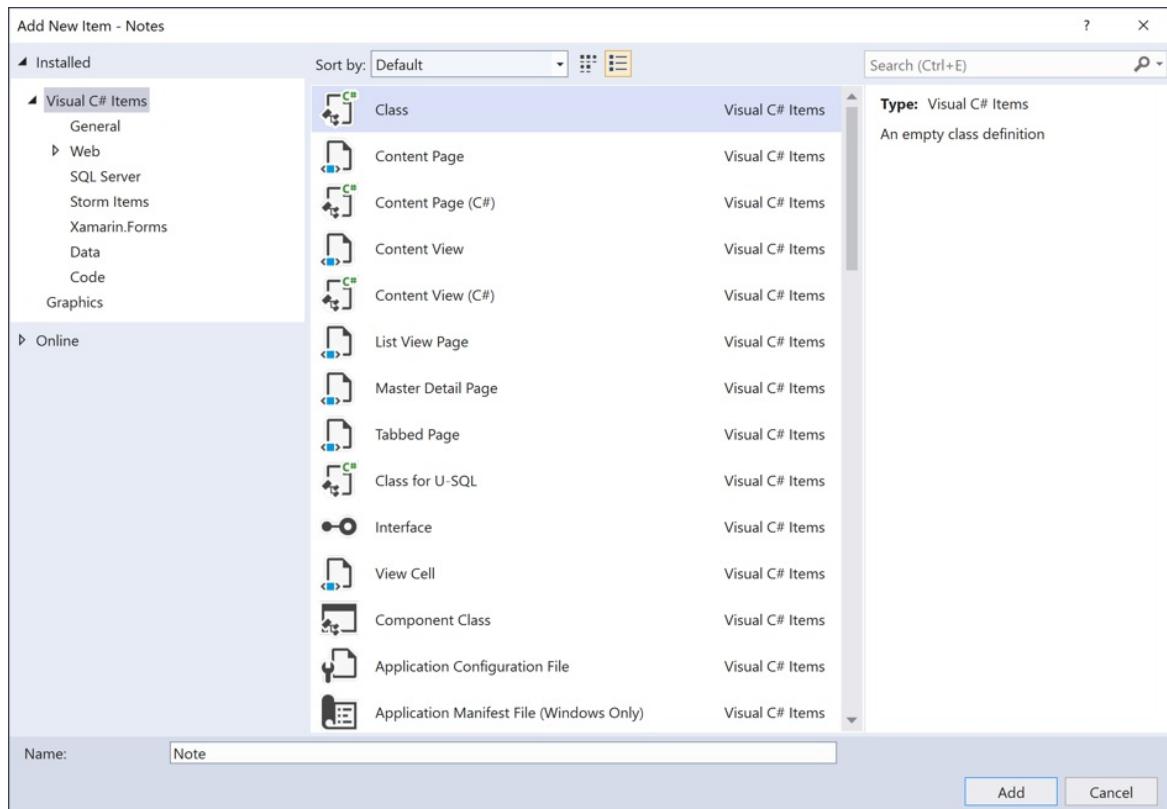
3. In Solution Explorer, name the new folder **Models**:



4. In Solution Explorer, select the Models folder, right-click, and select Add > Class...:



5. In the Add New Item dialog, select Visual C# Items > Class, name the new file Note, and click the Add button:



This will add a class named **Note** to the **Models** folder of the **Notes** project.

6. In **Note.cs**, remove all of the template code and replace it with the following code:

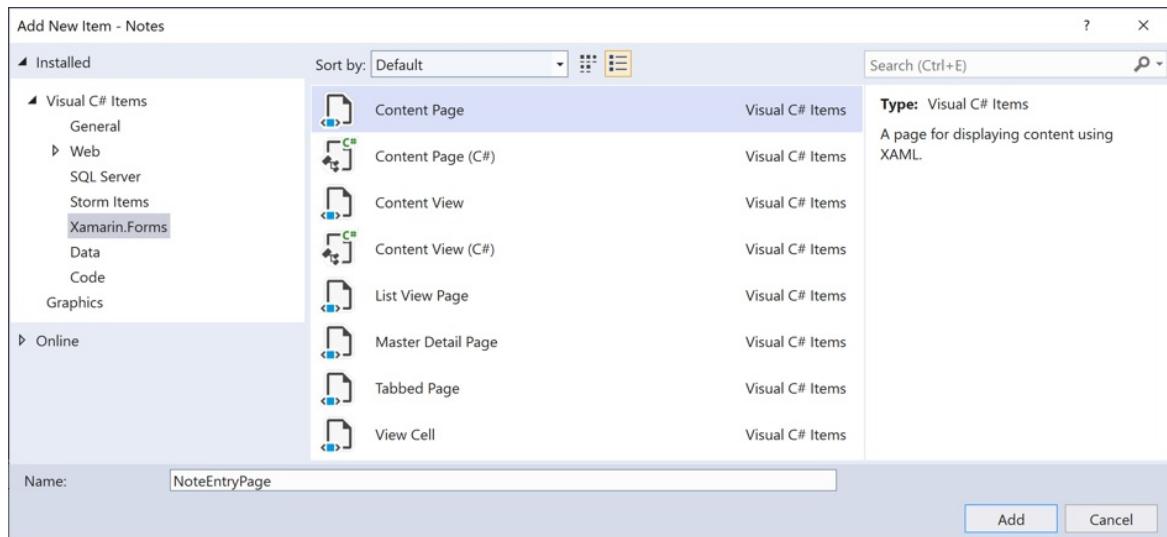
```
using System;

namespace Notes.Models
{
    public class Note
    {
        public string Filename { get; set; }
        public string Text { get; set; }
        public DateTime Date { get; set; }
    }
}
```

This class defines a **Note** model that will store data about each note in the application.

Save the changes to **Note.cs** by pressing **CTRL+S**.

7. In **Solution Explorer**, in the **Notes** project, select the **Views** folder, right-click, and select **Add > New Item....**. In the **Add New Item** dialog, select **Visual C# Items > Xamarin.Forms > Content Page**, name the new file **NoteEntryPage**, and click the **Add** button:



This will add a new page named **NoteEntryPage** to the **Views** folder of the project. This page will be used for note entry.

- In **NoteEntryPage.xaml**, remove all of the template code and replace it with the following code:

```
<?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="Notes.Views.NoteEntryPage"
    Title="Note Entry">
    <!-- Layout children vertically -->
    <StackLayout Margin="20">
        <Editor Placeholder="Enter your note"
            Text="{Binding Text}"
            HeightRequest="100" />
        <!-- Layout children in two columns -->
        <Grid ColumnDefinitions="*,*">
            <Button Text="Save"
                Clicked="OnSaveButtonClicked" />
            <Button Grid.Column="1"
                Text="Delete"
                Clicked="OnDeleteButtonClicked"/>
        </Grid>
    </StackLayout>
</ContentPage>
```

This code declaratively defines the user interface for the page, which consists of an **Editor** for text input, and two **Button** objects that direct the application to save or delete a file. The two **Button** instances are horizontally laid out in a **Grid**, with the **Editor** and **Grid** being vertically laid out in a **StackLayout**. In addition, the **Editor** uses data binding to bind to the **Text** property of the **Note** model. For more information about data binding, see [Data binding](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

Save the changes to **NoteEntryPage.xaml** by pressing **CTRL+S**.

- In **NoteEntryPage.xaml.cs**, remove all of the template code and replace it with the following code:

```
using System;
using System.IO;
using Notes.Models;
using Xamarin.Forms;

namespace Notes.Views
{
    [QueryProperty(nameof(ItemId), nameof(ItemId))]
    public partial class NoteEntryPage : ContentPage
    {
```

```

    {
        public string ItemId
        {
            set
            {
                LoadNote(value);
            }
        }

        public NoteEntryPage()
        {
            InitializeComponent();

            // Set the BindingContext of the page to a new Note.
            BindingContext = new Note();
        }

        void LoadNote(string filename)
        {
            try
            {
                // Retrieve the note and set it as the BindingContext of the page.
                Note note = new Note
                {
                    Filename = filename,
                    Text = File.ReadAllText(filename),
                    Date = File.GetCreationTime(filename)
                };
                BindingContext = note;
            }
            catch (Exception)
            {
                Console.WriteLine("Failed to load note.");
            }
        }
    }

    async void OnSaveButtonClicked(object sender, EventArgs e)
    {
        var note = (Note)BindingContext;

        if (string.IsNullOrWhiteSpace(note.Filename))
        {
            // Save the file.
            var filename = Path.Combine(AppFolderPath, $"{Path.GetFileNameWithoutExtension(note.Filename)}.notes.txt");
            File.WriteAllText(filename, note.Text);
        }
        else
        {
            // Update the file.
            File.WriteAllText(note.Filename, note.Text);
        }

        // Navigate backwards
        await Shell.Current.GoToAsync("../");
    }

    async void onDeleteButtonClicked(object sender, EventArgs e)
    {
        var note = (Note)BindingContext;

        // Delete the file.
        if (File.Exists(note.Filename))
        {
            File.Delete(note.Filename);
        }

        // Navigate backwards
        await Shell.Current.GoToAsync("../");
    }
}

```

```
    }  
}
```

This code stores a `Note` instance, which represents a single note, in the `BindingContext` of the page. The class is decorated with a `QueryPropertyAttribute` that enables data to be passed into the page, during navigation, via query parameters. The first argument for the `QueryPropertyAttribute` specifies the name of the property that will receive the data, with the second argument specifying the query parameter id. Therefore, the `QueryParameterAttribute` in the above code specifies that the `ItemId` property will receive the data passed in the `ItemId` query parameter from the URI specified in a `GoToAsync` method call. The `ItemId` property then calls the `LoadNote` method to create a `Note` object from the file on the device, and sets the `BindingContext` of the page to the `Note` object.

When the `Save` `Button` is pressed the `OnSaveButtonClicked` event handler is executed, which either saves the content of the `Editor` to a new file with a randomly generated filename, or to an existing file if a note is being updated. In both cases, the file is stored in the local application data folder for the application. Then the method navigates back to the previous page. When the `Delete` `Button` is pressed the `OnDeleteButtonClicked` event handler is executed, which deletes the file, provided that it exists, and navigates back to the previous page. For more information about navigation, see [Navigation](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `NoteEntryPage.xaml.cs` by pressing **CTRL+S**.

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

10. In **Solution Explorer**, in the **Notes** project, open **NotesPage.xaml** in the **Views** folder.
11. In **NotesPage.xaml**, remove all of the template code and replace it with the following code:

```

<?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="Notes.Views.NotesPage"
    Title="Notes">
    <!-- Add an item to the toolbar -->
    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Add"
            Clicked="OnAddClicked" />
    </ContentPage.ToolbarItems>

    <!-- Display notes in a list -->
    <CollectionView x:Name="collectionView"
        Margin="20"
        SelectionMode="Single"
        SelectionChanged="OnSelectionChanged">
        <CollectionView.ItemsLayout>
            <LinearItemsLayout Orientation="Vertical"
                ItemSpacing="10" />
        </CollectionView.ItemsLayout>
        <!-- Define the appearance of each item in the list -->
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <StackLayout>
                    <Label Text="{Binding Text}"
                        FontSize="Medium"/>
                    <Label Text="{Binding Date}"
                        TextColor="Silver"
                        FontSize="Small" />
                </StackLayout>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</ContentPage>

```

This code declaratively defines the user interface for the page, which consists of a `CollectionView` and a `ToolbarItem`. The `CollectionView` uses data binding to display any notes that are retrieved by the application. Selecting a note will navigate to the `NoteEntryPage` where the note can be modified. Alternatively, a new note can be created by pressing the `ToolbarItem`. For more information about data binding, see [Data binding](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

Save the changes to `NotesPage.xaml` by pressing **CTRL+S**.

12. In Solution Explorer, in the **Notes** project, expand `NotesPage.xaml` in the **Views** folder and open `NotesPage.xaml.cs`.
13. In `NotesPage.xaml.cs`, remove all of the template code and replace it with the following code:

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Notes.Models;
using Xamarin.Forms;

namespace Notes.Views
{
    public partial class NotesPage : ContentPage
    {
        public NotesPage()
        {
            InitializeComponent();
        }

        protected override void OnAppearing()
        {
            base.OnAppearing();

            var notes = new List<Note>();

            // Create a Note object from each file.
            var files = Directory.EnumerateFiles(App.FolderPath, "*.notes.txt");
            foreach (var filename in files)
            {
                notes.Add(new Note
                {
                    Filename = filename,
                    Text = File.ReadAllText(filename),
                    Date = File.GetCreationTime(filename)
                });
            }

            // Set the data source for the CollectionView to a
            // sorted collection of notes.
            collectionView.ItemsSource = notes
                .OrderBy(d => d.Date)
                .ToList();
        }

        async void OnAddClicked(object sender, EventArgs e)
        {
            // Navigate to the NoteEntryPage, without passing any data.
            await Shell.Current.GoToAsync(nameof(NoteEntryPage));
        }

        async void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            if (e.CurrentSelection != null)
            {
                // Navigate to the NoteEntryPage, passing the filename as a query parameter.
                Note note = (Note)e.CurrentSelection.FirstOrDefault();
                await Shell.Current.GoToAsync($"{{nameof(NoteEntryPage)}}?
{{nameof(NoteEntryPage.ItemId)}}={{note.Filename}}");
            }
        }
    }
}

```

This code defines the functionality for the `NotesPage`. When the page appears, the `OnAppearing` method is executed, which populates the `CollectionView` with any notes that have been retrieved from the local application data folder. When the `ToolbarItem` is pressed the `OnAddClicked` event handler is executed. This method navigates to the `NoteEntryPage`. When an item in the `CollectionView` is selected the `OnSelectionChanged` event handler is executed. This method navigates to the `NoteEntryPage`, provided

that an item in the `CollectionView` is selected, passing the `Filename` property of the selected `Note` as a query parameter to the page. For more information about navigation, see [Navigation](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

Save the changes to `NotesPage.xaml.cs` by pressing **CTRL+S**.

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

14. In **Solution Explorer**, in the **Notes** project, expand `AppShell.xaml` and open `AppShell.xaml.cs`. Then replace the existing code with the following code:

```
using Notes.Views;
using Xamarin.Forms;

namespace Notes
{
    public partial class AppShell : Shell
    {
        public AppShell()
        {
            InitializeComponent();
            Routing.RegisterRoute(nameof(NoteEntryPage), typeof(NoteEntryPage));
        }
    }
}
```

This code registers a route for the `NoteEntryPage`, which isn't represented in the Shell visual hierarchy (`AppShell.xaml`). This page can then be navigated to using URI-based navigation, with the `GoToAsync` method.

Save the changes to `AppShell.xaml.cs` by pressing **CTRL+S**.

15. In **Solution Explorer**, in the **Notes** project, expand `App.xaml` and open `App.xaml.cs`. Then replace the existing code with the following code:

```

using System;
using System.IO;
using Xamarin.Forms;

namespace Notes
{
    public partial class App : Application
    {
        public static string FolderPath { get; private set; }

        public App()
        {
            InitializeComponent();
            FolderPath =
                Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData));
            MainPage = new AppShell();
        }

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

        protected override void OnResume()
        {
        }
    }
}

```

This code adds a namespace declaration for the `System.IO` namespace, and adds a declaration for a static `FolderPath` property of type `string`. The `FolderPath` property is used to store the path on the device where note data will be stored. In addition, the code initializes the `FolderPath` property in the `App` constructor, and initializes the `MainPage` property to the subclassed `Shell` object.

Save the changes to `App.xaml.cs` by pressing **CTRL+S**.

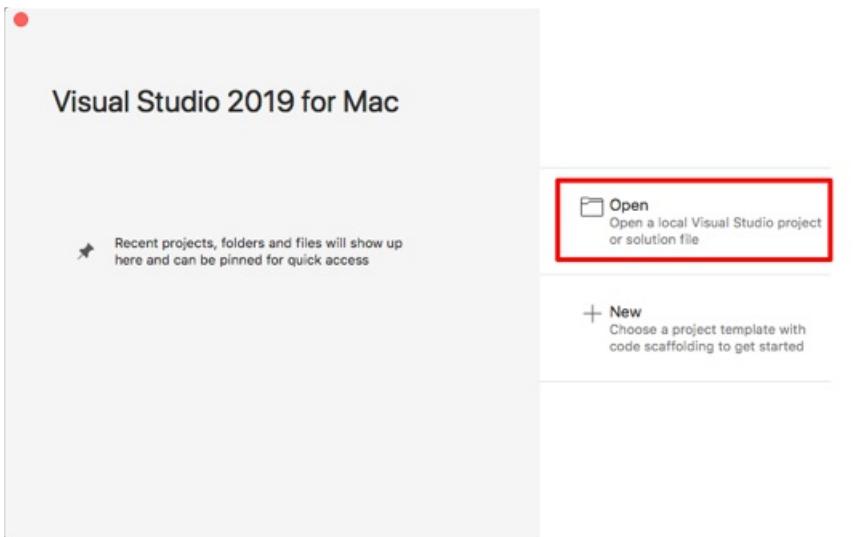
16. Build and run the project on each platform. For more information, see [Building the quickstart](#).

On the **NotesPage** press the **Add** button to navigate to the **NoteEntryPage** and enter a note. After saving the note the application will navigate back to the **NotesPage**.

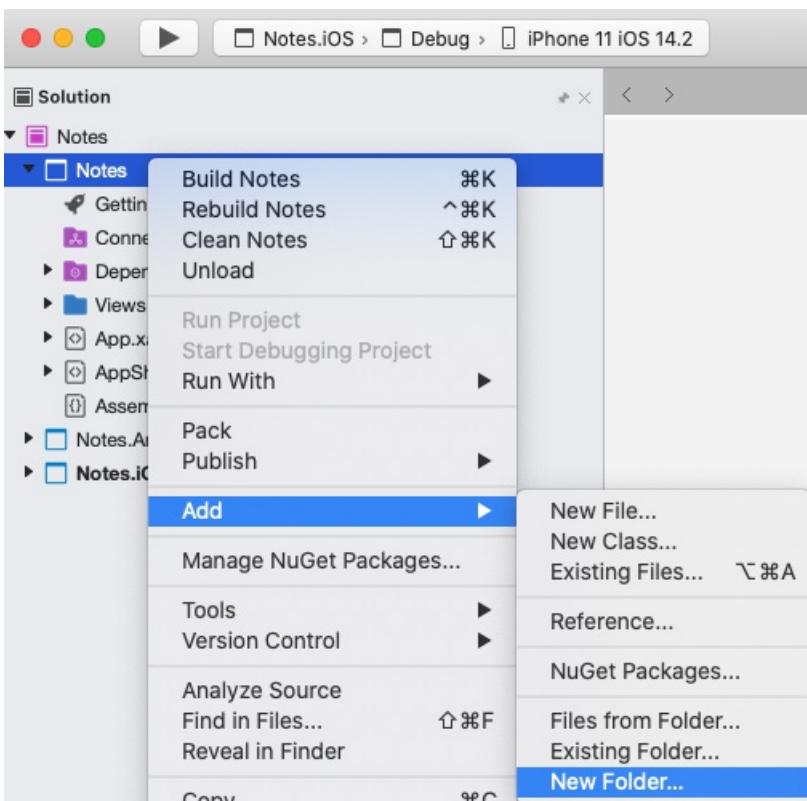
Enter several notes, of varying length, to observe the application behavior. Close the application and re-launch it to ensure that the notes you entered were saved to the device.

Update the app with Visual Studio for Mac

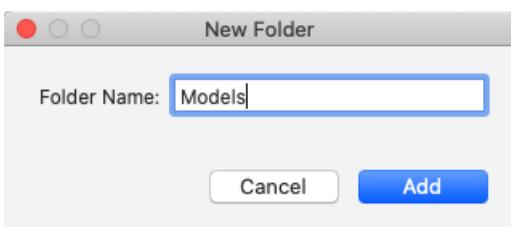
1. Launch Visual Studio for Mac. In the start window click **Open**, and in the dialog select the solution file for the Notes project:



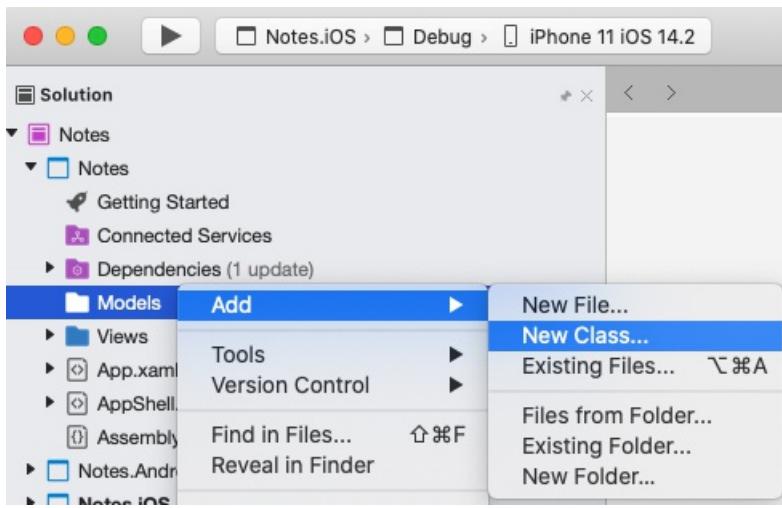
2. In the **Solution Pad**, right-click on the **Notes** project, and select **Add > New Folder**:



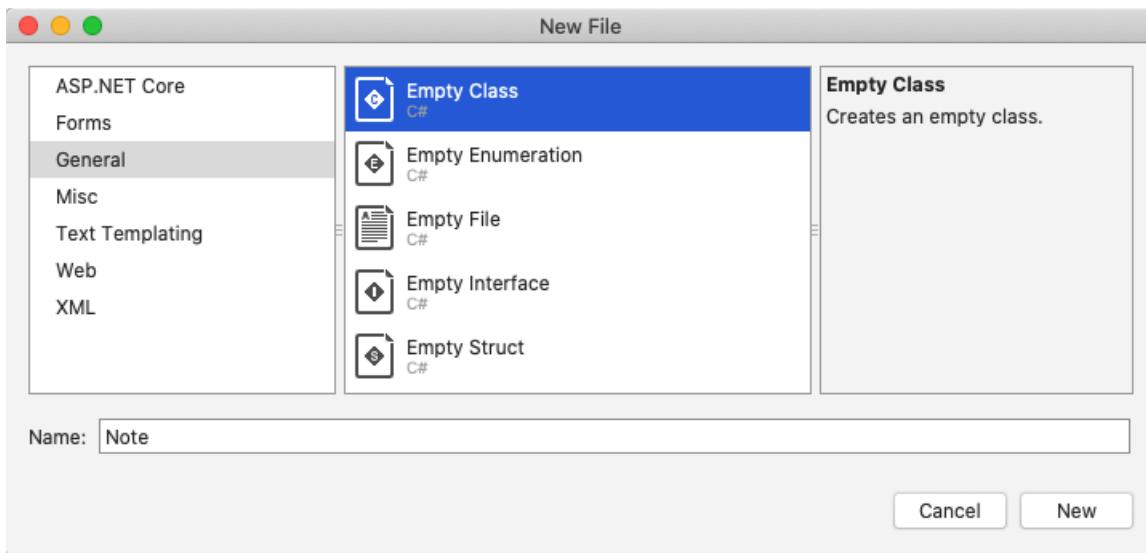
3. In the **New Folder** dialog, name the new folder **Models**:



4. In the **Solution Pad**, select the **Models** folder, right-click, and select **Add > New Class...**:



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



This will add a class named **Note** to the **Models** folder of the **Notes** project.

6. In **Note.cs**, remove all of the template code and replace it with the following code:

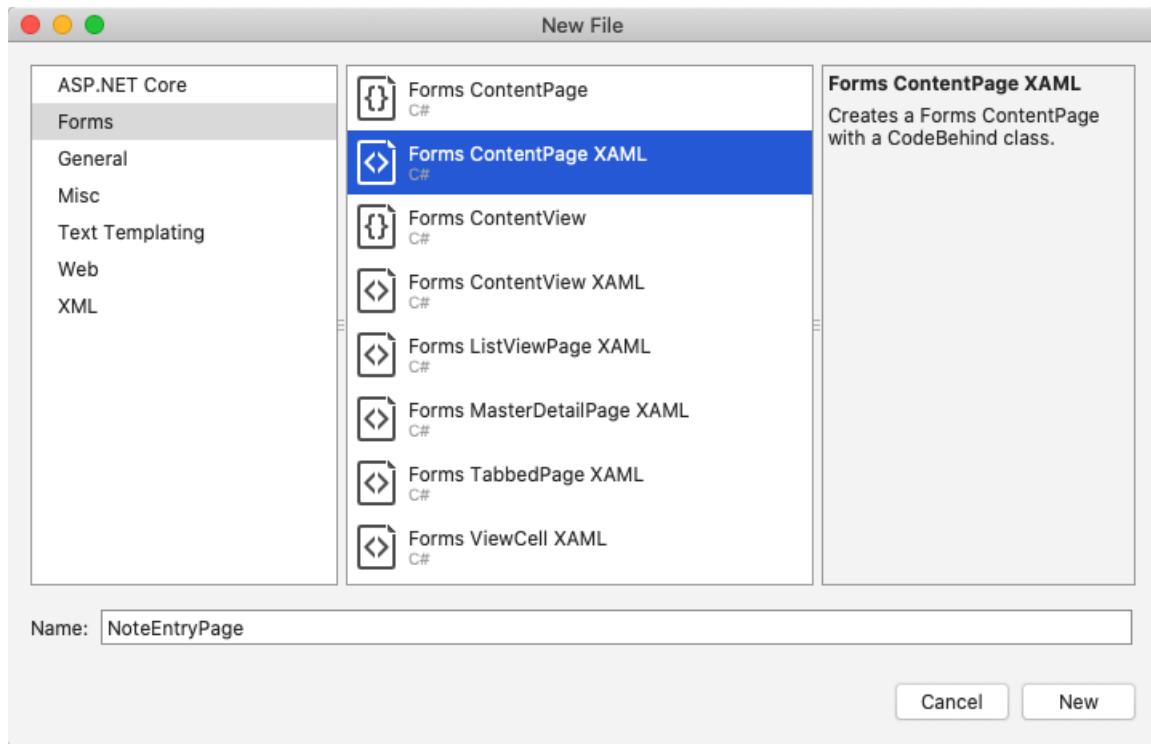
```
using System;

namespace Notes.Models
{
    public class Note
    {
        public string Filename { get; set; }
        public string Text { get; set; }
        public DateTime Date { get; set; }
    }
}
```

This class defines a **Note** model that will store data about each note in the application.

Save the changes to **Note.cs** by choosing **File > Save** (or by pressing **⌘ + S**).

7. In the **Solution Pad**, select the **Notes** project, right-click, and select **Add > New File....** In the **New File** dialog, select **Forms > Forms ContentPage XAML**, name the new file **NoteEntryPage**, and click the **New** button:



This will add a new page named **NoteEntryPage** to the **Views** folder of the project. This page will be used for note entry.

8. In **NoteEntryPage.xaml**, remove all of the template code and replace it with the following code:

```
<?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="Notes.Views.NoteEntryPage"
    Title="Note Entry">
    <!-- Layout children vertically -->
    <StackLayout Margin="20">
        <Editor Placeholder="Enter your note"
            Text="{Binding Text}"
            HeightRequest="100" />
        <!-- Layout children in two columns -->
        <Grid ColumnDefinitions="*,*">
            <Button Text="Save"
                Clicked="OnSaveButtonClicked" />
            <Button Grid.Column="1"
                Text="Delete"
                Clicked="OnDeleteButtonClicked"/>
        </Grid>
    </StackLayout>
</ContentPage>
```

This code declaratively defines the user interface for the page, which consists of an **Editor** for text input, and two **Button** objects that direct the application to save or delete a file. The two **Button** instances are horizontally laid out in a **Grid**, with the **Editor** and **Grid** being vertically laid out in a **StackLayout**. In addition, the **Editor** uses data binding to bind to the **Text** property of the **Note** model. For more information about data binding, see [Data binding](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

Save the changes to **NoteEntryPage.xaml** by choosing **File > Save** (or by pressing **⌘ + S**).

9. In **NoteEntryPage.xaml.cs**, remove all of the template code and replace it with the following code:

```
using System;
using System.IO;
using Notes.Models;
```

```
using Notes.Models;
using Xamarin.Forms;

namespace Notes.Views
{
    [QueryProperty(nameof(ItemId), nameof(ItemId))]
    public partial class NoteEntryPage : ContentPage
    {
        public string ItemId
        {
            set
            {
                LoadNote(value);
            }
        }

        public NoteEntryPage()
        {
            InitializeComponent();

            // Set the BindingContext of the page to a new Note.
            BindingContext = new Note();
        }

        void LoadNote(string filename)
        {
            try
            {
                // Retrieve the note and set it as the BindingContext of the page.
                Note note = new Note
                {
                    Filename = filename,
                    Text = File.ReadAllText(filename),
                    Date = File.GetCreationTime(filename)
                };
                BindingContext = note;
            }
            catch (Exception)
            {
                Console.WriteLine("Failed to load note.");
            }
        }

        async void OnSaveButtonClicked(object sender, EventArgs e)
        {
            var note = (Note)BindingContext;

            if (string.IsNullOrWhiteSpace(note.Filename))
            {
                // Save the file.
                var filename = Path.Combine(App.FolderPath, $"{Path.GetRandomFileName()}.notes.txt");
                File.WriteAllText(filename, note.Text);
            }
            else
            {
                // Update the file.
                File.WriteAllText(note.Filename, note.Text);
            }

            // Navigate backwards
            await Shell.Current.GoToAsync("..");
        }

        async void OnDeleteButtonClicked(object sender, EventArgs e)
        {
            var note = (Note)BindingContext;

            // Delete the file.
            if (File.Exists(note.Filename))
            {

```

```

        {
            File.Delete(note.Filename);
        }

        // Navigate backwards
        await Shell.Current.GoToAsync("../");
    }
}

```

This code stores a `Note` instance, which represents a single note, in the `BindingContext` of the page. The class is decorated with a `QueryPropertyAttribute` that enables data to be passed into the page, during navigation, via query parameters. The first argument for the `QueryPropertyAttribute` specifies the name of the property that will receive the data, with the second argument specifying the query parameter id. Therefore, the `QueryParameterAttribute` in the above code specifies that the `ItemId` property will receive the data passed in the `ItemId` query parameter from the URI specified in a `GoToAsync` method call. The `ItemId` property then calls the `LoadNote` method to create a `Note` object from the file on the device, and sets the `BindingContext` of the page to the `Note` object.

When the `Save` `Button` is pressed the `OnSaveButtonClicked` event handler is executed, which either saves the content of the `Editor` to a new file with a randomly generated filename, or to an existing file if a note is being updated. In both cases, the file is stored in the local application data folder for the application. Then the method navigates back to the previous page. When the `Delete` `Button` is pressed the `OnDeleteButtonClicked` event handler is executed, which deletes the file, provided that it exists, and navigates back to the previous page. For more information about navigation, see [Navigation](#) in the [Xamarin.Forms Shell Quickstart Deep Dive](#).

Save the changes to `NoteEntryPage.xaml.cs` by choosing `File > Save` (or by pressing `⌘ + S`).

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

10. In the **Solution Pad**, in the **Notes** project, open **NotesPage.xaml** in the **Views** folder.
11. In **NotesPage.xaml**, remove all of the template code and replace it with the following code:

```

<?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="Notes.Views.NotesPage"
    Title="Notes">
    <!-- Add an item to the toolbar -->
    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Add"
            Clicked="OnAddClicked" />
    </ContentPage.ToolbarItems>

    <!-- Display notes in a list -->
    <CollectionView x:Name="collectionView"
        Margin="20"
       SelectionMode="Single"
        SelectionChanged="OnSelectionChanged">
        <CollectionView.ItemsLayout>
            <LinearItemsLayout Orientation="Vertical"
                ItemSpacing="10" />
        </CollectionView.ItemsLayout>
        <!-- Define the appearance of each item in the list -->
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <StackLayout>
                    <Label Text="{Binding Text}"
                        FontSize="Medium"/>
                    <Label Text="{Binding Date}"
                        TextColor="Silver"
                        FontSize="Small" />
                </StackLayout>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</ContentPage>

```

This code declaratively defines the user interface for the page, which consists of a `CollectionView` and a `ToolbarItem`. The `CollectionView` uses data binding to display any notes that are retrieved by the application. Selecting a note will navigate to the `NoteEntryPage` where the note can be modified. Alternatively, a new note can be created by pressing the `ToolbarItem`. For more information about data binding, see [Data binding](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

Save the changes to `NotesPage.xaml` by choosing **File > Save** (or by pressing **⌘ + S**).

12. In the **Solution Pad**, in the **Notes** project, expand `NotesPage.xaml` in the **Views** folder and open `NotesPage.xaml.cs`.
13. In `NotesPage.xaml.cs`, remove all of the template code and replace it with the following code:

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Notes.Models;
using Xamarin.Forms;

namespace Notes.Views
{
    public partial class NotesPage : ContentPage
    {
        public NotesPage()
        {
            InitializeComponent();
        }

        protected override void OnAppearing()
        {
            base.OnAppearing();

            var notes = new List<Note>();

            // Create a Note object from each file.
            var files = Directory.EnumerateFiles(App.FolderPath, "*.notes.txt");
            foreach (var filename in files)
            {
                notes.Add(new Note
                {
                    Filename = filename,
                    Text = File.ReadAllText(filename),
                    Date = File.GetCreationTime(filename)
                });
            }

            // Set the data source for the CollectionView to a
            // sorted collection of notes.
            collectionView.ItemsSource = notes
                .OrderBy(d => d.Date)
                .ToList();
        }

        async void OnAddClicked(object sender, EventArgs e)
        {
            // Navigate to the NoteEntryPage, without passing any data.
            await Shell.Current.GoToAsync(nameof(NoteEntryPage));
        }

        async void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            if (e.CurrentSelection != null)
            {
                // Navigate to the NoteEntryPage, passing the filename as a query parameter.
                Note note = (Note)e.CurrentSelection.FirstOrDefault();
                await Shell.Current.GoToAsync($"{{nameof(NoteEntryPage)}}?
{{nameof(NoteEntryPage.ItemId)}}={{note.Filename}}");
            }
        }
    }
}

```

This code defines the functionality for the `NotesPage`. When the page appears, the `OnAppearing` method is executed, which populates the `CollectionView` with any notes that have been retrieved from the local application data folder. When the `ToolbarItem` is pressed the `OnAddClicked` event handler is executed. This method navigates to the `NoteEntryPage`. When an item in the `CollectionView` is selected the `OnSelectionChanged` event handler is executed. This method navigates to the `NoteEntryPage`, provided

that an item in the `CollectionView` is selected, passing the `Filename` property of the selected `Note` as a query parameter to the page. For more information about navigation, see [Navigation](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

Save the changes to `NotesPage.xaml.cs` by choosing **File > Save** (or by pressing **⌘ + S**).

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

14. In the **Solution Pad**, in the **Notes** project, expand `AppShell.xaml` and open `AppShell.xaml.cs`. Then replace the existing code with the following code:

```
using Notes.Views;
using Xamarin.Forms;

namespace Notes
{
    public partial class AppShell : Shell
    {
        public AppShell()
        {
            InitializeComponent();
            Routing.RegisterRoute(nameof(NoteEntryPage), typeof(NoteEntryPage));
        }
    }
}
```

This code registers a route for the `NoteEntryPage`, which isn't represented in the Shell visual hierarchy. This page can then be navigated to using URI-based navigation, with the `GoToAsync` method.

Save the changes to `AppShell.xaml.cs` by choosing **File > Save** (or by pressing **⌘ + S**).

15. In the **Solution Pad**, in the **Notes** project, expand `App.xaml` and open `App.xaml.cs`. Then replace the existing code with the following code:

```

using System;
using System.IO;
using Xamarin.Forms;

namespace Notes
{
    public partial class App : Application
    {
        public static string FolderPath { get; private set; }

        public App()
        {
            InitializeComponent();
            FolderPath =
                Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData));
            MainPage = new AppShell();
        }

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

        protected override void OnResume()
        {
        }
    }
}

```

This code adds a namespace declaration for the `System.IO` namespace, and adds a declaration for a static `FolderPath` property of type `string`. The `FolderPath` property is used to store the path on the device where note data will be stored. In addition, the code initializes the `FolderPath` property in the `App` constructor, and initializes the `MainPage` property to the subclassed `Shell` object.

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

16. Build and run the project on each platform. For more information, see [Building the quickstart](#).

On the **NotesPage** press the **Add** button to navigate to the **NoteEntryPage** and enter a note. After saving the note the application will navigate back to the **NotesPage**.

Enter several notes, of varying length, to observe the application behavior. Close the application and re-launch it to ensure that the notes you entered were saved to the device.

Next steps

In this quickstart, you learned how to:

- Add additional pages to a `Xamarin.Forms` Shell application.
- Perform navigation between pages.
- Use data binding to synchronize data between user interface elements and their data source.

Continue to the next quickstart to modify the application so that it stores its data in a local SQLite.NET database.

[Next](#)

Related links

- [Notes \(sample\)](#)
- [Xamarin.Forms Shell quickstart deep dive](#)

Store data in a local SQLite.NET database

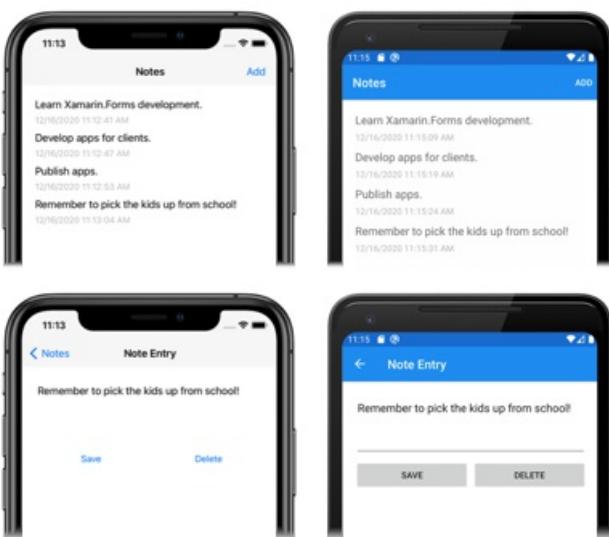
8/4/2022 • 11 minutes to read • [Edit Online](#)

 [Download the sample](#)

In this quickstart, you will learn how to:

- Store data locally in a SQLite.NET database.

The quickstart walks through how to store data in a local SQLite.NET database, from a Xamarin.Forms Shell application. The final application is shown below:

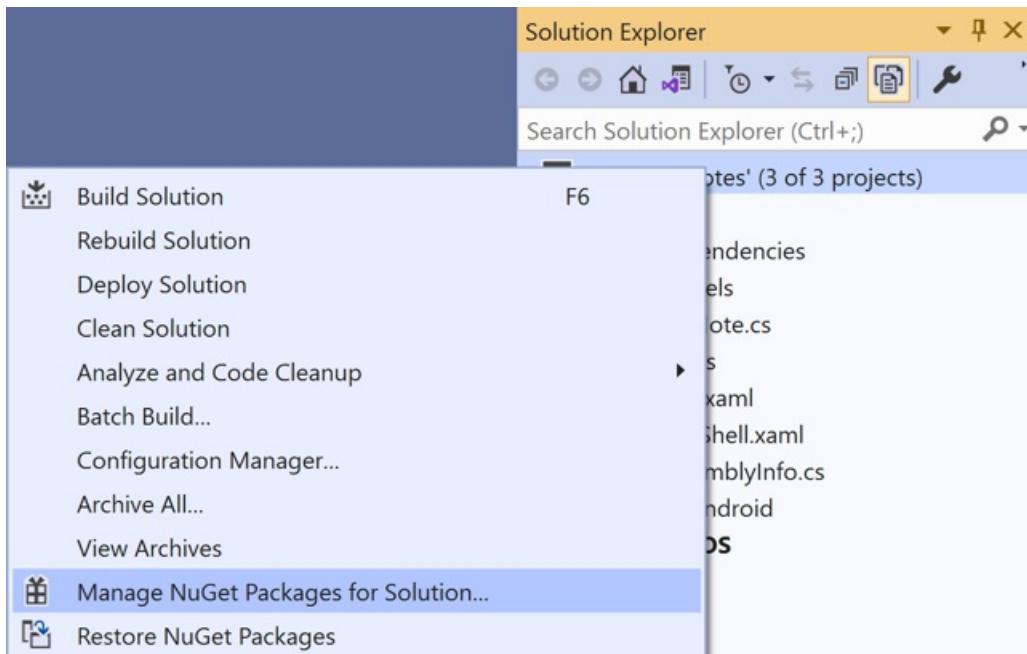


Prerequisites

You should successfully complete the [previous quickstart](#) before attempting this quickstart. Alternatively, download the [previous quickstart sample](#) and use it as the starting point for this quickstart.

Update the app with Visual Studio

1. Launch Visual Studio and open the Notes solution.
2. In **Solution Explorer**, right-click the **Notes** solution and select **Manage NuGet Packages for Solution...:**



3. In the **NuGet Package Manager**, select the **Browse** tab, and search for the **sqlite-net-pcl** NuGet package.

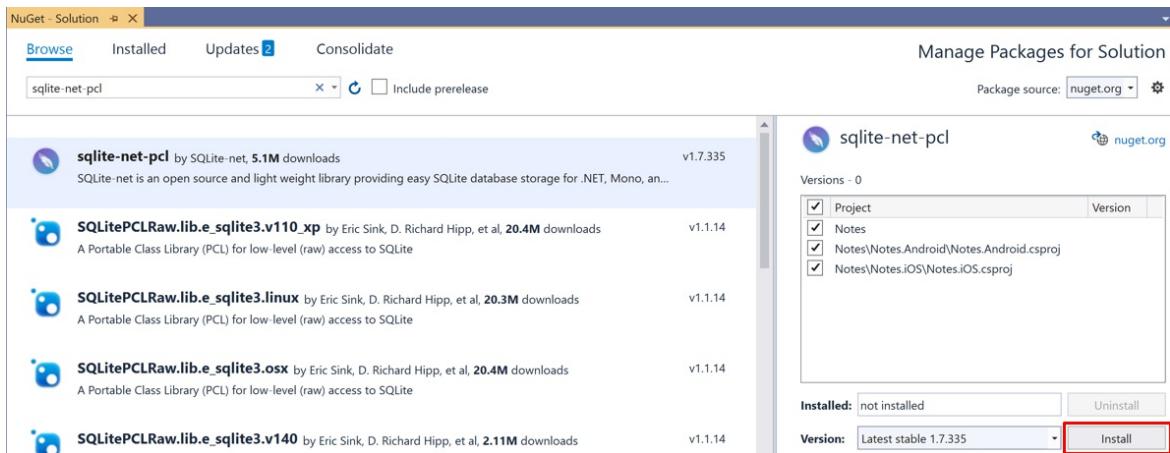
WARNING

There are many NuGet packages with similar names. The correct package has these attributes:

- **Authors:** SQLite-net
- **NuGet link:** [sqlite-net-pcl](https://www.nuget.org/packages/sqlite-net-pcl/)

Despite the package name, this NuGet package can be used in .NET Standard projects.

In the **NuGet Package Manager**, select the correct **sqlite-net-pcl** package, check the **Project** checkbox, and click the **Install** button to add it to the solution:



This package will be used to incorporate database operations into the application, and will be added to every project in the solution.

Close the **NuGet Package Manager**.

4. In **Solution Explorer**, in the **Notes** project, open **Note.cs** in the **Models** folder and replace the existing code with the following code:

```
using System;
using SQLite;

namespace Notes.Models
{
    public class Note
    {
        [PrimaryKey, AutoIncrement]
        public int ID { get; set; }
        public string Text { get; set; }
        public DateTime Date { get; set; }
    }
}
```

This class defines a `Note` model that will store data about each note in the application. The `ID` property is marked with `PrimaryKey` and `AutoIncrement` attributes to ensure that each `Note` instance in the SQLite.NET database will have a unique id provided by SQLite.NET.

Save the changes to `Note.cs` by pressing **CTRL+S**.

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

5. In **Solution Explorer**, add a new folder named **Data** to the **Notes** project.
6. In **Solution Explorer**, in the **Notes** project, add a new class named **NoteDatabase** to the **Data** folder.
7. In `NoteDatabase.cs`, replace the existing code with the following code:

```

using System.Collections.Generic;
using System.Threading.Tasks;
using SQLite;
using Notes.Models;

namespace Notes.Data
{
    public class NoteDatabase
    {
        readonly SQLiteAsyncConnection database;

        public NoteDatabase(string dbPath)
        {
            database = new SQLiteAsyncConnection(dbPath);
            database.CreateTableAsync<Note>().Wait();
        }

        public Task<List<Note>> GetNotesAsync()
        {
            //Get all notes.
            return database.Table<Note>().ToListAsync();
        }

        public Task<Note> GetNoteAsync(int id)
        {
            // Get a specific note.
            return database.Table<Note>()
                .Where(i => i.ID == id)
                .FirstOrDefaultAsync();
        }

        public Task<int> SaveNoteAsync(Note note)
        {
            if (note.ID != 0)
            {
                // Update an existing note.
                return database.UpdateAsync(note);
            }
            else
            {
                // Save a new note.
                return database.InsertAsync(note);
            }
        }

        public Task<int> DeleteNoteAsync(Note note)
        {
            // Delete a note.
            return database.DeleteAsync(note);
        }
    }
}

```

This class contains code to create the database, read data from it, write data to it, and delete data from it. The code uses asynchronous SQLite.NET APIs that move database operations to background threads. In addition, the `NoteDatabase` constructor takes the path of the database file as an argument. This path will be provided by the `App` class in the next step.

Save the changes to `NoteDatabase.cs` by pressing **CTRL+S**.

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

8. In Solution Explorer, in the **Notes** project, expand **App.xaml** and double-click **App.xaml.cs** to open it. Then replace the existing code with the following code:

```
using System;
using System.IO;
using Notes.Data;
using Xamarin.Forms;

namespace Notes
{
    public partial class App : Application
    {
        static NoteDatabase database;

        // Create the database connection as a singleton.
        public static NoteDatabase Database
        {
            get
            {
                if (database == null)
                {
                    database = new
NoteDatabase(Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
"Notes.db3"));
                }
                return database;
            }
        }

        public App()
        {
            InitializeComponent();
            MainPage = new AppShell();
        }

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

        protected override void OnResume()
        {
        }
    }
}
```

This code defines a `Database` property that creates a new `NoteDatabase` instance as a singleton, passing in the filename of the database as the argument to the `NoteDatabase` constructor. 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.

Save the changes to **App.xaml.cs** by pressing **CTRL+S**.

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

9. In Solution Explorer, in the **Notes** project, expand **NotesPage.xaml** in the **Views** folder and open

NotesPage.xaml.cs. Then replace the `OnAppearing` and `OnSelectionChanged` methods with the following code:

```
protected override async void OnAppearing()
{
    base.OnAppearing();

    // Retrieve all the notes from the database, and set them as the
    // data source for the CollectionView.
    collectionView.ItemsSource = await App.Database.GetNotesAsync();
}

async void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (e.CurrentSelection != null)
    {
        // Navigate to the NoteEntryPage, passing the ID as a query parameter.
        Note note = (Note)e.CurrentSelection.FirstOrDefault();
        await Shell.Current.GoToAsync($"{nameof(NoteEntryPage)}?{nameof(NoteEntryPage.ItemId)}={note.ID.ToString()}");
    }
}
```

The `OnAppearing` method populates the `CollectionView` with any notes stored in the database. The `OnSelectionChanged` method navigates to the `NoteEntryPage`, passing the `ID` property of the selected `Note` object as a query parameter.

Save the changes to **NotesPage.xaml.cs** by pressing **CTRL+S**.

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

10. In **Solution Explorer**, expand **NoteEntryPage.xaml** in the **Views** folder and open **NoteEntryPage.xaml.cs**. Then replace the `LoadNote`, `OnSaveButtonClicked`, and `OnDeleteButtonClicked` methods with the following code:

```

async void LoadNote(string itemId)
{
    try
    {
        int id = Convert.ToInt32(itemId);
        // Retrieve the note and set it as the BindingContext of the page.
        Note note = await App.Database.GetNoteAsync(id);
        BindingContext = note;
    }
    catch (Exception)
    {
        Console.WriteLine("Failed to load note.");
    }
}

async void OnSaveButtonClicked(object sender, EventArgs e)
{
    var note = (Note)BindingContext;
    note.Date = DateTime.UtcNow;
    if (!string.IsNullOrWhiteSpace(note.Text))
    {
        await App.Database.SaveNoteAsync(note);
    }

    // Navigate backwards
    await Shell.Current.GoToAsync("../");
}

async void onDeleteButtonClicked(object sender, EventArgs e)
{
    var note = (Note)BindingContext;
    await App.Database.DeleteNoteAsync(note);

    // Navigate backwards
    await Shell.Current.GoToAsync("../");
}

```

The `NoteEntryPage` uses the `LoadNote` method to retrieve the note from the database, whose ID was passed as a query parameter to the page, and store it as a `Note` object in the `BindingContext` of the page. When the `OnSaveButtonClicked` event handler is executed, the `Note` instance is saved to the database and the application navigates back to the previous page. When the `onDeleteButtonClicked` event handler is executed, the `Note` instance is deleted from the database and the application navigates back to the previous page.

Save the changes to `NoteEntryPage.xaml.cs` by pressing **CTRL+S**.

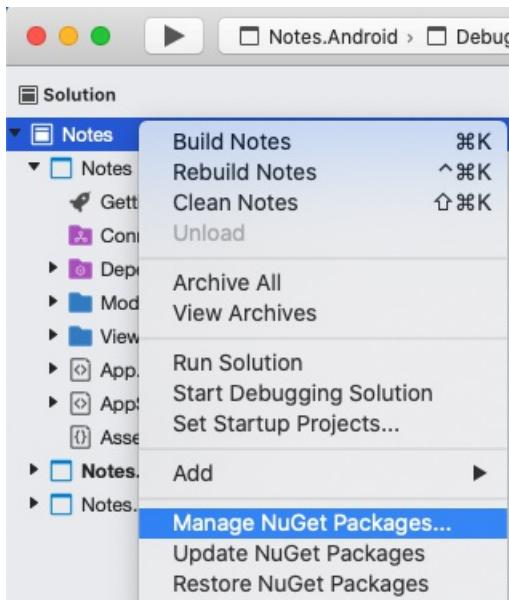
11. Build and run the project on each platform. For more information, see [Building the quickstart](#).

On the **NotesPage** press the **Add** button to navigate to the **NoteEntryPage** and enter a note. After saving the note the application will navigate back to the **NotesPage**.

Enter several notes, of varying length, to observe the application behavior. Close the application and re-launch it to ensure that the notes you entered were saved to the database.

Update the app with Visual Studio for Mac

1. Launch Visual Studio for Mac and open the Notes solution.
2. In the **Solution Pad**, right-click the **Notes** solution and select **Manage NuGet Packages...**:



3. In the **Manage NuGet Packages** dialog, select the **Browse** tab, and search for the **sqlite-net-pcl** NuGet package.

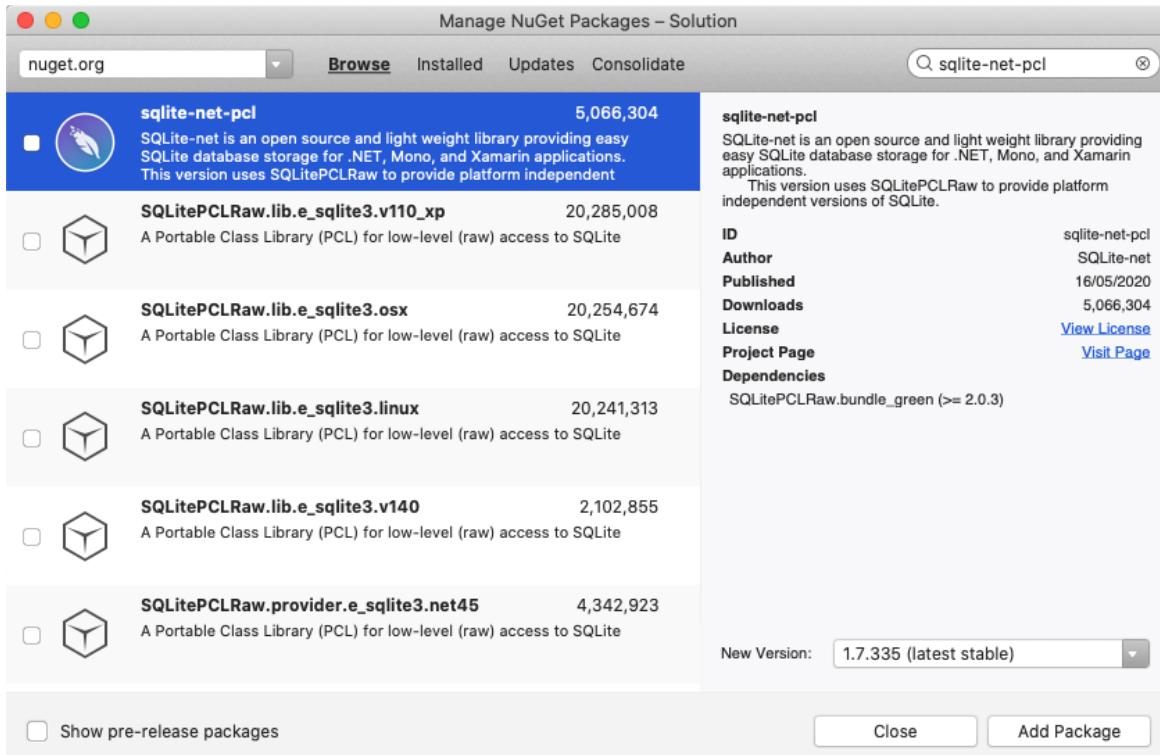
WARNING

There are many NuGet packages with similar names. The correct package has these attributes:

- **Authors:** SQLite-net
- **NuGet link:** [sqlite-net-pcl](https://www.nuget.org/packages/sqlite-net-pcl/)

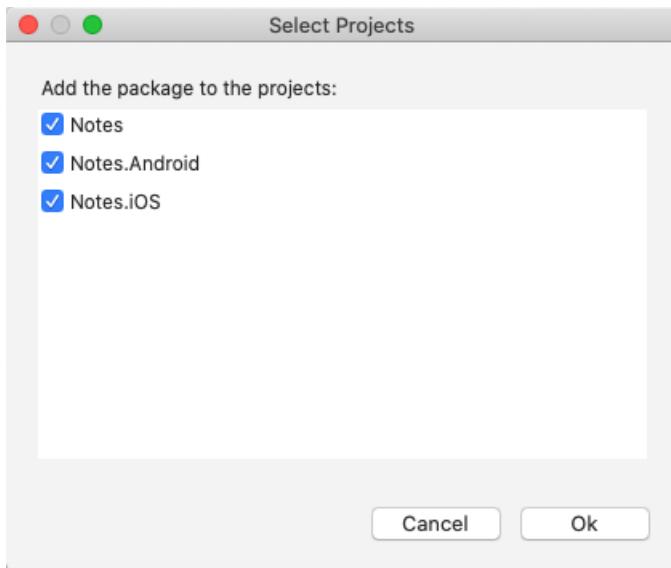
Despite the package name, this NuGet package can be used in .NET Standard projects.

In the **Manage NuGet Packages** dialog, select the **sqlite-net-pcl** package, and click the **Add Package** button to add it to the solution:



This package will be used to incorporate database operations into the application.

4. In the **Select Projects** dialog, ensure that every checkbox is checked and press the **Ok** button:



This will add the NuGet package to every project in the solution.

5. In the **Solution Pad**, in the **Notes** project, open **Note.cs** in the **Models** folder and replace the existing code with the following code:

```
using System;
using SQLite;

namespace Notes.Models
{
    public class Note
    {
        [PrimaryKey, AutoIncrement]
        public int ID { get; set; }
        public string Text { get; set; }
        public DateTime Date { get; set; }
    }
}
```

This class defines a `Note` model that will store data about each note in the application. The `ID` property is marked with `PrimaryKey` and `AutoIncrement` attributes to ensure that each `Note` instance in the SQLite.NET database will have a unique id provided by SQLite.NET.

Save the changes to **Note.cs** by choosing **File > Save** (or by pressing **⌘ + S**).

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

6. In the **Solution Pad**, add a new folder named **Data** to the **Notes** project.
7. In the **Solution Pad**, in the **Notes** project, add a new class named **NoteDatabase** to the **Data** folder.
8. In **NoteDatabase.cs**, replace the existing code with the following code:

```

using System.Collections.Generic;
using System.Threading.Tasks;
using SQLite;
using Notes.Models;

namespace Notes.Data
{
    public class NoteDatabase
    {
        readonly SQLiteAsyncConnection database;

        public NoteDatabase(string dbPath)
        {
            database = new SQLiteAsyncConnection(dbPath);
            database.CreateTableAsync<Note>().Wait();
        }

        public Task<List<Note>> GetNotesAsync()
        {
            //Get all notes.
            return database.Table<Note>().ToListAsync();
        }

        public Task<Note> GetNoteAsync(int id)
        {
            // Get a specific note.
            return database.Table<Note>()
                .Where(i => i.ID == id)
                .FirstOrDefaultAsync();
        }

        public Task<int> SaveNoteAsync(Note note)
        {
            if (note.ID != 0)
            {
                // Update an existing note.
                return database.UpdateAsync(note);
            }
            else
            {
                // Save a new note.
                return database.InsertAsync(note);
            }
        }

        public Task<int> DeleteNoteAsync(Note note)
        {
            // Delete a note.
            return database.DeleteAsync(note);
        }
    }
}

```

This class contains code to create the database, read data from it, write data to it, and delete data from it. The code uses asynchronous SQLite.NET APIs that move database operations to background threads. In addition, the `NoteDatabase` constructor takes the path of the database file as an argument. This path will be provided by the `App` class in the next step.

Save the changes to `NoteDatabase.cs` by choosing **File > Save** (or by pressing **⌘ + S**).

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

9. In the **Solution Pad**, in the **Notes** project, expand **App.xaml** and double-click **App.xaml.cs** to open it. Then replace the existing code with the following code:

```
using System;
using System.IO;
using Notes.Data;
using Xamarin.Forms;

namespace Notes
{
    public partial class App : Application
    {
        static NoteDatabase database;

        // Create the database connection as a singleton.
        public static NoteDatabase Database
        {
            get
            {
                if (database == null)
                {
                    database = new
NoteDatabase(Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
"Notes.db3"));
                }
                return database;
            }
        }

        public App()
        {
            InitializeComponent();
            MainPage = new AppShell();
        }

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

        protected override void OnResume()
        {
        }
    }
}
```

This code defines a `Database` property that creates a new `NoteDatabase` instance as a singleton, passing in the filename of the database as the argument to the `NoteDatabase` constructor. 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.

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

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

10. In the **Solution Pad**, in the **Notes** project, expand **NotesPage.xaml** in the **Views** folder and open

NotesPage.xaml.cs. Then replace the `OnAppearing` and `OnSelectionChanged` methods with the following code:

```
protected override async void OnAppearing()
{
    base.OnAppearing();

    // Retrieve all the notes from the database, and set them as the
    // data source for the CollectionView.
    collectionView.ItemsSource = await App.Database.GetNotesAsync();
}

async void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (e.CurrentSelection != null)
    {
        // Navigate to the NoteEntryPage, passing the ID as a query parameter.
        Note note = (Note)e.CurrentSelection.FirstOrDefault();
        await Shell.Current.GoToAsync($"{nameof(NoteEntryPage)}?{nameof(NoteEntryPage.ItemId)}={note.ID.ToString()}");
    }
}
```

The `OnAppearing` method populates the `CollectionView` with any notes stored in the database. The `OnSelectionChanged` method navigates to the `NoteEntryPage`, passing the `ID` property of the selected `Note` object as a query parameter.

Save the changes to **NotesPage.xaml.cs** by choosing **File > Save** (or by pressing **⌘ + S**).

WARNING

The application will not currently build due to errors that will be fixed in subsequent steps.

11. In the **Solution Pad**, expand **NoteEntryPage.xaml** in the **Views** folder and open **NoteEntryPage.xaml.cs**. Then replace the `LoadNote`, `OnSaveButtonClicked`, and `OnDeleteButtonClicked` methods with the following code:

```

async void LoadNote(string itemId)
{
    try
    {
        int id = Convert.ToInt32(itemId);
        // Retrieve the note and set it as the BindingContext of the page.
        Note note = await App.Database.GetNoteAsync(id);
        BindingContext = note;
    }
    catch (Exception)
    {
        Console.WriteLine("Failed to load note.");
    }
}

async void OnSaveButtonClicked(object sender, EventArgs e)
{
    var note = (Note)BindingContext;
    note.Date = DateTime.UtcNow;
    if (!string.IsNullOrWhiteSpace(note.Text))
    {
        await App.Database.SaveNoteAsync(note);
    }

    // Navigate backwards
    await Shell.Current.GoToAsync("../");
}

async void onDeleteButtonClicked(object sender, EventArgs e)
{
    var note = (Note)BindingContext;
    await App.Database.DeleteNoteAsync(note);

    // Navigate backwards
    await Shell.Current.GoToAsync("../");
}

```

The `NoteEntryPage` uses the `LoadNote` method to retrieve the note from the database, whose ID was passed as a query parameter to the page, and store it as a `Note` object in the `BindingContext` of the page. When the `OnSaveButtonClicked` event handler is executed, the `Note` instance is saved to the database and the application navigates back to the previous page. When the `onDeleteButtonClicked` event handler is executed, the `Note` instance is deleted from the database and the application navigates back to the previous page.

Save the changes to `NoteEntryPage.xaml.cs` by choosing **File > Save** (or by pressing **⌘ + S**).

12. Build and run the project on each platform. For more information, see [Building the quickstart](#).

On the `NotesPage` press the **Add** button to navigate to the `NoteEntryPage` and enter a note. After saving the note the application will navigate back to the `NotesPage`.

Enter several notes, of varying length, to observe the application behavior. Close the application and re-launch it to ensure that the notes you entered were saved to the database.

Next steps

In this quickstart, you learned how to:

- Store data locally in a SQLite.NET database.

Continue to the next quickstart to style the application with XAML styles.

[Next](#)

Related links

- [Notes \(sample\)](#)
- [Xamarin.Forms Shell Quickstart Deep Dive](#)

Style a cross-platform Xamarin.Forms application

8/4/2022 • 10 minutes to read • [Edit Online](#)



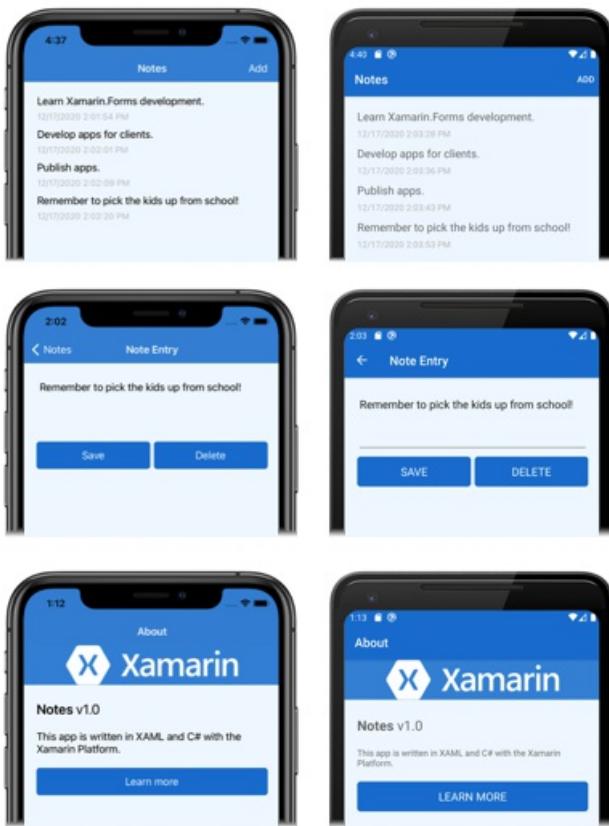
[Download the sample](#)

In this quickstart, you will learn how to:

- Style a Xamarin.Forms Shell application using XAML styles.
- Use XAML Hot Reload to see UI changes without rebuilding your application.

The quickstart walks through how to style a cross-platform Xamarin.Forms application with XAML styles. In addition, the quickstart uses XAML Hot Reload to update the UI of your running application, without having to rebuild the application. For more information about XAML Hot Reload, see [XAML Hot Reload for Xamarin.Forms](#).

The final application is shown below:



Prerequisites

You should successfully complete the [previous quickstart](#) before attempting this quickstart. Alternatively, download the [previous quickstart sample](#) and use it as the starting point for this quickstart.

Update the app with Visual Studio

1. Launch Visual Studio and open the Notes solution.
2. Build and run the project on your chosen platform. For more information, see [Building the quickstart](#).
Leave the application running and return to Visual Studio.
3. In **Solution Explorer**, in the **Notes** project, open **App.xaml**. Then replace the existing code with the following code:

```

<?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="Notes.App">

    <!-- Resources used by multiple pages in the application -->
    <Application.Resources>

        <Thickness x:Key="PageMargin">20</Thickness>

        <!-- Colors -->
        <Color x:Key="AppPrimaryColor">#1976D2</Color>
        <Color x:Key="AppBackgroundColor">AliceBlue</Color>
        <Color x:Key="PrimaryColor">Black</Color>
        <Color x:Key="SecondaryColor">White</Color>
        <Color x:Key="TertiaryColor">Silver</Color>

        <!-- Implicit styles -->
        <Style TargetType="ContentPage"
            ApplyToDerivedTypes="True">
            <Setter Property="BackgroundColor"
                Value="{StaticResource AppBackgroundColor}" />
        </Style>

        <Style TargetType="Button">
            <Setter Property="FontSize"
                Value="Medium" />
            <Setter Property="BackgroundColor"
                Value="{StaticResource AppPrimaryColor}" />
            <Setter Property="TextColor"
                Value="{StaticResource SecondaryColor}" />
            <Setter Property="CornerRadius"
                Value="5" />
        </Style>
    </Application.Resources>
</Application>

```

This code defines a `Thickness` value, a series of `Color` values, and implicit styles for the `ContentPage` and `Button` types. Note that these styles, which are in the application-level `ResourceDictionary`, can be consumed throughout the application. For more information about XAML styling, see [Styling](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

After making the changes to `App.xaml`, XAML Hot Reload will update the UI of the running app, with no need to rebuild the application. Specifically, the background color each page will change. By default Hot Reload applies changes immediately after stopping typing. However, there's a [preference setting](#) that can be changed, if you prefer, to wait until file save to apply changes.

4. In **Solution Explorer**, in the **Notes** project, open `AppShell.xaml`. Then replace the existing code with the following code:

```

<?xml version="1.0" encoding="UTF-8"?>
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Notes.Views"
       x:Class="Notes.AppShell">

    <Shell.Resources>
        <!-- Style Shell elements -->
        <Style x:Key="BaseStyle"
               TargetType="Element">
            <Setter Property="Shell.BackgroundColor"
                    Value="{StaticResource AppPrimaryColor}" />
            <Setter Property="Shell.ForegroundColor"
                    Value="{StaticResource SecondaryColor}" />
            <Setter Property="Shell.TitleColor"
                    Value="{StaticResource SecondaryColor}" />
            <Setter Property="Shell.TabBarUnselectedColor"
                    Value="#95FFFFFF"/>
        </Style>
        <Style TargetType="TabBar"
               BasedOn="{StaticResource BaseStyle}" />
    </Shell.Resources>

    <!-- Display a bottom tab bar containing two tabs -->
    <TabBar>
        <ShellContent Title="Notes"
                      Icon="icon_feed.png"
                      ContentTemplate="{DataTemplate views:NotesPage}" />
        <ShellContent Title="About"
                      Icon="icon_about.png"
                      ContentTemplate="{DataTemplate views:AboutPage}" />
    </TabBar>
</Shell>

```

This code adds two styles to the `Shell` resource dictionary, which define a series of `Color` values used by the application.

After making the `AppShell.xaml` changes, XAML Hot Reload will update the UI of the running app, without rebuilding the application. Specifically, the background color of the Shell chrome will change.

5. In **Solution Explorer**, in the **Notes** project, open **NotesPage.xaml** in the **Views** folder. Then replace the existing code with the following code:

```

<?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="Notes.Views.NotesPage"
    Title="Notes">

    <ContentPage.Resources>
        <!-- Define a visual state for the Selected state of the CollectionView -->
        <Style TargetType="StackLayout">
            <Setter Property="VisualStateManager.VisualStateGroups">
                <VisualStateGroupList>
                    <VisualStateGroup x:Name="CommonStates">
                        <VisualState x:Name="Normal" />
                        <VisualState x:Name="Selected">
                            <VisualState.Setters>
                                <Setter Property="BackgroundColor"
                                       Value="{StaticResource AppPrimaryColor}" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
                </VisualStateGroupList>
            </Setter>
        </Style>
    </ContentPage.Resources>

    <!-- Add an item to the toolbar -->
    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Add"
                     Clicked="OnAddClicked" />
    </ContentPage.ToolbarItems>

    <!-- Display notes in a list -->
    <CollectionView x:Name="collectionView"
                   Margin="{StaticResource PageMargin}"
                   SelectionMode="Single"
                   SelectionChanged="OnSelectionChanged">
        <CollectionView.ItemsLayout>
            <LinearItemsLayout Orientation="Vertical"
                               ItemSpacing="10" />
        </CollectionView.ItemsLayout>
        <!-- Define the appearance of each item in the list -->
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <StackLayout>
                    <Label Text="{Binding Text}"
                           FontSize="Medium" />
                    <Label Text="{Binding Date}"
                           TextColor="{StaticResource TertiaryColor}"
                           FontSize="Small" />
                </StackLayout>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</ContentPage>

```

This code adds an implicit style for the `StackLayout` that defines the appearance of each selected item in the `CollectionView`, to the page-level `ResourceDictionary`, and sets the `collectionView.Margin` and `Label.TextColor` property to values defined in the application-level `ResourceDictionary`. Note that the `StackLayout` implicit style was added to the page-level `ResourceDictionary`, because it is only consumed by the `NotesPage`.

After making the `NotesPage.xaml` changes, XAML Hot Reload will update the UI of the running app, without rebuilding the application. Specifically, the color of selected items in the `CollectionView` will change.

6. In Solution Explorer, in the **Notes** project, open **NoteEntryPage.xaml** in the **Views** folder. Then replace the existing code with the following code:

```
<?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="Notes.Views.NoteEntryPage"
    Title="Note Entry">
<ContentPage.Resources>
    <!-- Implicit styles -->
    <Style TargetType="{x:Type Editor}">
        <Setter Property="BackgroundColor"
            Value="{StaticResource AppBackgroundColor}" />
    </Style>
</ContentPage.Resources>

<!-- Layout children vertically -->
<StackLayout Margin="{StaticResource PageMargin}">
    <Editor Placeholder="Enter your note"
        Text="{Binding Text}"
        HeightRequest="100" />
    <Grid ColumnDefinitions="*,*">
        <!-- Layout children in two columns -->
        <Button Text="Save"
            Clicked="OnSaveButtonClicked" />
        <Button Grid.Column="1"
            Text="Delete"
            Clicked="OnDeleteButtonClicked"/>
    </Grid>
</StackLayout>
</ContentPage>
```

This code adds an implicit style for the `Editor` to the page-level `ResourceDictionary`, and sets the `StackLayout.Margin` property to a value defined in the application-level `ResourceDictionary`. Note that the `Editor` implicit styles was added to the page-level `ResourceDictionary` because it's only consumed by the `NoteEntryPage`.

7. In the running application, navigate to the `NoteEntryPage`.

XAML Hot Reload updated the UI of the application, without rebuilding it. Specifically, the background color of the `Editor` changed in the running application, as well as the appearance of the `Button` objects.

8. In Solution Explorer, in the **Notes** project, open **AboutPage.xaml** in the **Views** folder. Then replace the existing code with the following code:

```

<?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="Notes.Views.AboutPage"
    Title="About">
    <!-- Layout children in two rows -->
    <Grid RowDefinitions="Auto,*">
        <Image Source="xamarin_logo.png"
            BackgroundColor="{StaticResource AppPrimaryColor}"
            Opacity="0.85"
            VerticalOptions="Center"
            HeightRequest="64" />
        <!-- Layout children vertically -->
        <StackLayout Grid.Row="1"
            Margin="{StaticResource PageMargin}"
            Spacing="20">
            <Label FontSize="22">
                <Label.FormattedText>
                    <FormattedString>
                        <FormattedString.Spans>
                            <Span Text="Notes"
                                FontAttributes="Bold"
                                FontSize="22" />
                            <Span Text=" v1.0" />
                        </FormattedString.Spans>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
            <Label Text="This app is written in XAML and C# with the Xamarin Platform." />
            <Button Text="Learn more"
                Clicked="OnButtonClicked" />
        </StackLayout>
    </Grid>
</ContentPage>

```

This code sets the `Image.BackgroundColor` and `StackLayout.Margin` properties to values defined in the application-level [ResourceDictionary](#).

- In the running application, navigate to the [AboutPage](#).

XAML Hot Reload updated the UI of the application, without rebuilding it. Specifically, the background color of the [Image](#) changed in the running application.

Update the app with Visual Studio for Mac

- Launch Visual Studio for Mac and open the Notes project.
- Build and run the project on your chosen platform. For more information, see [Building the quickstart](#).
Leave the application running and return to Visual Studio for Mac.
- In the **Solution Pad**, in the **Notes** project, open **App.xaml**. Then replace the existing code with the following code:

```

<?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="Notes.App">

    <!-- Resources used by multiple pages in the application -->
    <Application.Resources>

        <Thickness x:Key="PageMargin">20</Thickness>

        <!-- Colors -->
        <Color x:Key="AppPrimaryColor">#1976D2</Color>
        <Color x:Key="AppBackgroundColor">AliceBlue</Color>
        <Color x:Key="PrimaryColor">Black</Color>
        <Color x:Key="SecondaryColor">White</Color>
        <Color x:Key="TertiaryColor">Silver</Color>

        <!-- Implicit styles -->
        <Style TargetType="ContentPage"
            ApplyToDerivedTypes="True">
            <Setter Property="BackgroundColor"
                Value="{StaticResource AppBackgroundColor}" />
        </Style>

        <Style TargetType="Button">
            <Setter Property="FontSize"
                Value="Medium" />
            <Setter Property="BackgroundColor"
                Value="{StaticResource AppPrimaryColor}" />
            <Setter Property="TextColor"
                Value="{StaticResource SecondaryColor}" />
            <Setter Property="CornerRadius"
                Value="5" />
        </Style>
    </Application.Resources>
</Application>

```

This code defines a `Thickness` value, a series of `Color` values, and implicit styles for the `ContentPage` and `Button` types. Note that these styles, which are in the application-level `ResourceDictionary`, can be consumed throughout the application. For more information about XAML styling, see [Styling](#) in the [Xamarin.Forms Quickstart Deep Dive](#).

After making the changes to `App.xaml`, XAML Hot Reload will update the UI of the running app, with no need to rebuild the application. Specifically, the background color each page will change. By default Hot Reload applies changes immediately after stopping typing. However, there's a [preference setting](#) that can be changed, if you prefer, to wait until file save to apply changes.

4. In the **Solution Pad**, in the **Notes** project, open `AppShell.xaml`. Then replace the existing code with the following code:

```

<?xml version="1.0" encoding="UTF-8"?>
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Notes.Views"
       x:Class="Notes.AppShell">

    <Shell.Resources>
        <!-- Style Shell elements -->
        <Style x:Key="BaseStyle"
               TargetType="Element">
            <Setter Property="Shell.BackgroundColor"
                    Value="{StaticResource AppPrimaryColor}" />
            <Setter Property="Shell.ForegroundColor"
                    Value="{StaticResource SecondaryColor}" />
            <Setter Property="Shell.TitleColor"
                    Value="{StaticResource SecondaryColor}" />
            <Setter Property="Shell.TabBarUnselectedColor"
                    Value="#95FFFFFF"/>
        </Style>
        <Style TargetType="TabBar"
               BasedOn="{StaticResource BaseStyle}" />
    </Shell.Resources>

    <!-- Display a bottom tab bar containing two tabs -->
    <TabBar>
        <ShellContent Title="Notes"
                      Icon="icon_feed.png"
                      ContentTemplate="{DataTemplate views:NotesPage}" />
        <ShellContent Title="About"
                      Icon="icon_about.png"
                      ContentTemplate="{DataTemplate views:AboutPage}" />
    </TabBar>
</Shell>

```

This code adds two styles to the `Shell` resource dictionary, which define a series of `Color` values used by the application.

After making the `AppShell.xaml` changes, XAML Hot Reload will update the UI of the running app, without rebuilding the application. Specifically, the background color of the Shell chrome will change.

5. In the **Solution Pad**, in the **Notes** project, open **NotesPage.xaml** in the **Views** folder. Then replace the existing code with the following code:

```

<?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="Notes.Views.NotesPage"
    Title="Notes">

    <ContentPage.Resources>
        <!-- Define a visual state for the Selected state of the CollectionView -->
        <Style TargetType="StackLayout">
            <Setter Property="VisualStateManager.VisualStateGroups">
                <VisualStateGroupList>
                    <VisualStateGroup x:Name="CommonStates">
                        <VisualState x:Name="Normal" />
                        <VisualState x:Name="Selected">
                            <VisualState.Setters>
                                <Setter Property="BackgroundColor"
                                       Value="{StaticResource AppPrimaryColor}" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
                </VisualStateGroupList>
            </Setter>
        </Style>
    </ContentPage.Resources>

    <!-- Add an item to the toolbar -->
    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Add"
                     Clicked="OnAddClicked" />
    </ContentPage.ToolbarItems>

    <!-- Display notes in a list -->
    <CollectionView x:Name="collectionView"
                   Margin="{StaticResource PageMargin}"
                   SelectionMode="Single"
                   SelectionChanged="OnSelectionChanged">
        <CollectionView.ItemsLayout>
            <LinearItemsLayout Orientation="Vertical"
                               ItemSpacing="10" />
        </CollectionView.ItemsLayout>
        <!-- Define the appearance of each item in the list -->
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <StackLayout>
                    <Label Text="{Binding Text}"
                           FontSize="Medium" />
                    <Label Text="{Binding Date}"
                           TextColor="{StaticResource TertiaryColor}"
                           FontSize="Small" />
                </StackLayout>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</ContentPage>

```

This code adds an implicit style for the `StackLayout` that defines the appearance of each selected item in the `CollectionView`, to the page-level `ResourceDictionary`, and sets the `collectionView.Margin` and `Label.TextColor` property to values defined in the application-level `ResourceDictionary`. Note that the `StackLayout` implicit style was added to the page-level `ResourceDictionary`, because it is only consumed by the `NotesPage`.

After making the `NotesPage.xaml` changes, XAML Hot Reload will update the UI of the running app, without rebuilding the application. Specifically, the color of selected items in the `CollectionView` will change.

6. In the **Solution Pad**, in the **Notes** project, open **NoteEntryPage.xaml** in the **Views** folder. Then replace the existing code with the following code:

```
<?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="Notes.Views.NoteEntryPage"
    Title="Note Entry">
<ContentPage.Resources>
    <!-- Implicit styles -->
    <Style TargetType="{x:Type Editor}">
        <Setter Property="BackgroundColor"
            Value="{StaticResource AppBackgroundColor}" />
    </Style>
</ContentPage.Resources>

<!-- Layout children vertically -->
<StackLayout Margin="{StaticResource PageMargin}">
    <Editor Placeholder="Enter your note"
        Text="{Binding Text}"
        HeightRequest="100" />
    <!-- Layout children in two columns -->
    <Grid ColumnDefinitions="*,*">
        <Button Text="Save"
            Clicked="OnSaveButtonClicked" />
        <Button Grid.Column="1"
            Text="Delete"
            Clicked="OnDeleteButtonClicked"/>
    </Grid>
</StackLayout>
</ContentPage>
```

This code adds implicit styles for the `Editor` to the page-level `ResourceDictionary`, and sets the `StackLayout.Margin` property to a value defined in the application-level `ResourceDictionary`. Note that the `Editor` implicit style was added to the page-level `ResourceDictionary` because it's only consumed by the `NoteEntryPage`.

7. In the running application, navigate to the `NoteEntryPage`.

XAML Hot Reload updated the UI of the application, without rebuilding it. Specifically, the background color of the `Editor` changed in the running application, as well as the appearance of the `Button` objects.

8. In the **Solution Pad**, in the **Notes** project, open **AboutPage.xaml** in the **Views** folder. Then replace the existing code with the following code:

```

<?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="Notes.Views.AboutPage"
    Title="About">
    <!-- Layout children in two rows -->
    <Grid RowDefinitions="Auto,*">
        <Image Source="xamarin_logo.png"
            BackgroundColor="{StaticResource AppPrimaryColor}"
            Opacity="0.85"
            VerticalOptions="Center"
            HeightRequest="64" />
        <!-- Layout children vertically -->
        <StackLayout Grid.Row="1"
            Margin="{StaticResource PageMargin}"
            Spacing="20">
            <Label FontSize="22">
                <Label.FormattedText>
                    <FormattedString>
                        <FormattedString.Spans>
                            <Span Text="Notes"
                                FontAttributes="Bold"
                                FontSize="22" />
                            <Span Text=" v1.0" />
                        </FormattedString.Spans>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
            <Label Text="This app is written in XAML and C# with the Xamarin Platform." />
            <Button Text="Learn more"
                Clicked="OnButtonClicked" />
        </StackLayout>
    </Grid>
</ContentPage>

```

This code sets the `Image.BackgroundColor` and `StackLayout.Margin` properties to values defined in the application-level [ResourceDictionary](#).

9. In the running application, navigate to the [AboutPage](#).

XAML Hot Reload updated the UI of the application, without rebuilding it. Specifically, the background color of the [Image](#) changed in the running application.

Next steps

In this quickstart, you learned how to:

- Style a Xamarin.Forms Shell application using XAML styles.
- Use XAML Hot Reload to see UI changes without rebuilding your application.

To learn more about the fundamentals of application development using Xamarin.Forms Shell, continue to the quickstart deep dive.

[Next](#)

Related links

- [Notes \(sample\)](#)
- [XAML Hot Reload for Xamarin.Forms](#)
- [Xamarin.Forms Quickstart Deep Dive](#)

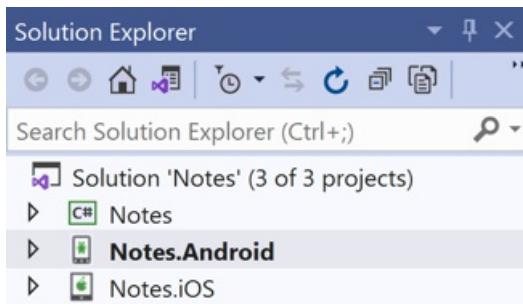
Xamarin.Forms Quickstart Deep Dive

8/4/2022 • 18 minutes to read • [Edit Online](#)

In the [Xamarin.Forms Quickstart](#), the Notes application was built. This article reviews what has been built to gain an understanding of the fundamentals of how Xamarin.Forms Shell 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 Notes application consists of one solution containing three projects, as shown in the following screenshot:

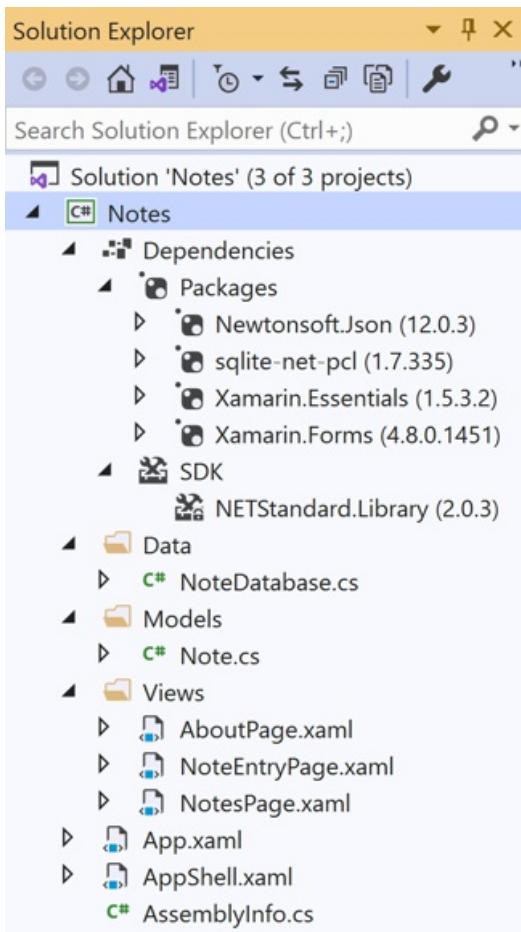


The projects are:

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

Anatomy of a Xamarin.Forms application

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



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

- **NuGet** – the Xamarin.Forms, Xamarin.Essentials, Newtonsoft.Json, and sqlite-net-pcl NuGet packages that have 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 Notes application consists of one solution containing three projects, as shown in the following screenshot:

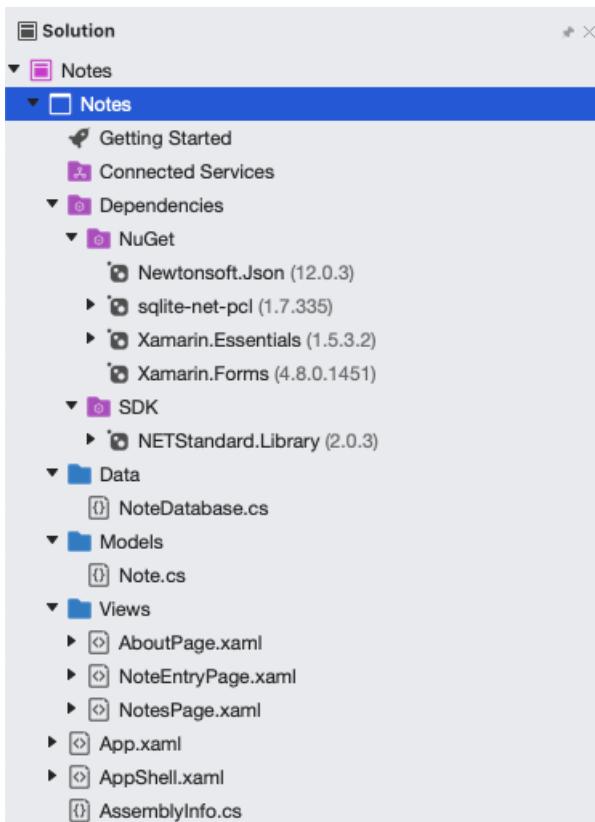


The projects are:

- Notes – This project is the .NET Standard library project that holds all of the shared code and shared UI.
- Notes.Android – This project holds Android-specific code and is the entry point for Android applications.
- Notes.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 Notes .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, Xamarin.Essentials, Newtonsoft.Json, and sqlite-net-pcl NuGet packages that have 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 multiple files:

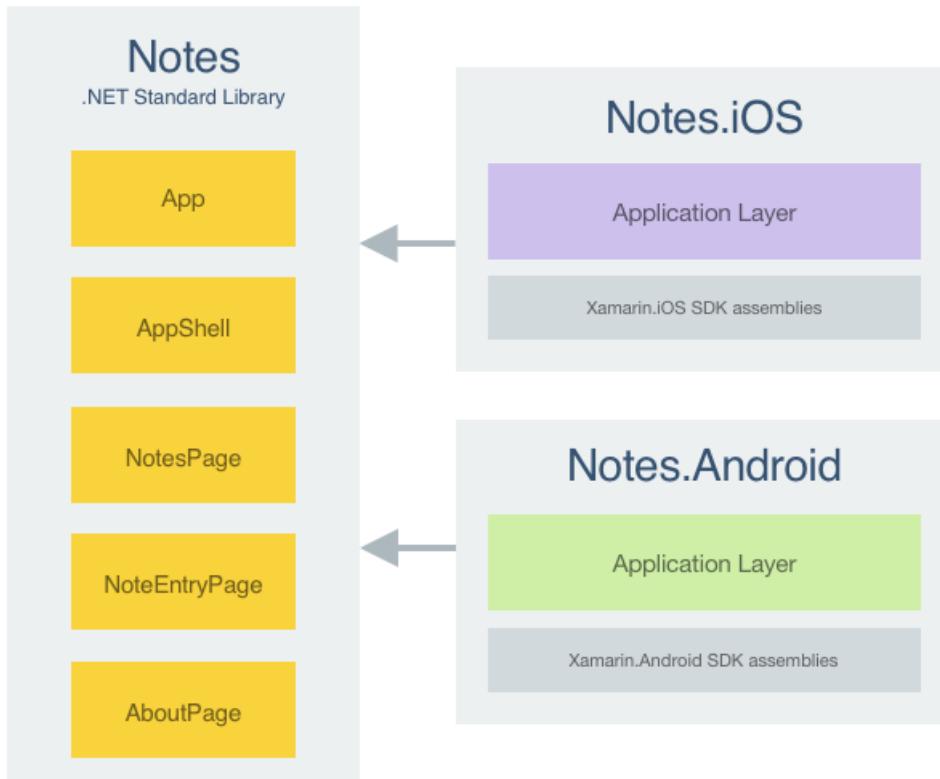
- **Data\NoteDatabase.cs** – This class contains code to create the database, read data from it, write data to it, and delete data from it.
- **Models\Note.cs** – This class defines a `Note` model whose instances store data about each note in the application.
- **Views\AboutPage.xaml** – The XAML markup for the `AboutPage` class, which defines the UI for the about page.
- **Views\AboutPage.xaml.cs** – The code-behind for the `AboutPage` class, which contains the business logic that is executed when the user interacts with the page.
- **Views\NotesPage.xaml** – The XAML markup for the `NotesPage` class, which defines the UI for the page shown when the application launches.
- **Views\NotesPage.xaml.cs** – The code-behind for the `NotesPage` class, which contains the business logic that is executed when the user interacts with the page.
- **Views\NoteEntryPage.xaml** – The XAML markup for the `NoteEntryPage` class, which defines the UI for the page shown when the user enters a note.
- **Views\NoteEntryPage.xaml.cs** – The code-behind for the `NoteEntryPage` class, which contains the business logic that is executed when the user interacts with the page.
- **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 Shell application, and for handling application lifecycle events.
- **AppShell.xaml** – The XAML markup for the `AppShell` class, which defines the visual hierarchy of the application.

- **AppShell.xaml.cs** – The code-behind for the `AppShell` class, which creates a route for the `NoteEntryPage` so that it can be navigated to programmatically.
- **AssemblyInfo.cs** – This file contains an application attribute about the project, that is applied at the assembly level.

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 Notes application:



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

```

using Xamarin.Forms;

namespace Notes
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
            MainPage = new AppShell();
        }
        // ...
    }
}
  
```

This code sets the `MainPage` property of the `App` class to the `AppShell` object. The `AppShell` class defines the

visual hierarchy of the application. Shell takes this visual hierarchy and produces the user interface for it. For more information about defining the visual hierarchy of the application, see [Application visual hierarchy](#).

In addition, the `AssemblyInfo.cs` file contains a single application attribute, that is applied at the assembly level:

```
using Xamarin.Forms.Xaml;

[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
```

The `XamlCompilation` attribute turns on the XAML compiler, so that XAML is compiled directly into intermediate language. For more information, see [XAML Compilation](#).

Launch the application on each platform

How the application is launched on each platform is specific to the platform.

iOS

To launch the initial Xamarin.Forms page in iOS, the Notes.iOS project defines the `AppDelegate` class that inherits from the `FormsApplicationDelegate` class:

```
namespace Notes.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 Notes.Android project includes code that creates an `Activity` with the `MainLauncher` attribute, with the activity inheriting from the `FormsAppCompatActivity` class:

```

namespace Notes.Droid
{
    [Activity(Label = "Notes",
        Icon = "@mipmap/icon",
        Theme = "@style/MainTheme",
        MainLauncher = true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
    {
        protected override void OnCreate(Bundle savedInstanceState)
        {
            TabLayoutResource = Resource.Layout.Tabbar;
            ToolbarResource = Resource.Layout.Toolbar;

            base.OnCreate(savedInstanceState);
            global::Xamarin.Forms.Forms.Init(this, savedInstanceState);
            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.

Application visual hierarchy

Xamarin.Forms Shell applications define the visual hierarchy of the application in a class that subclasses the `Shell` class. In the Notes application this is the `AppShell` class:

```

<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Notes.Views"
       x:Class="Notes.AppShell">
    <TabBar>
        <ShellContent Title="Notes"
                      Icon="icon_feed.png"
                      ContentTemplate="{DataTemplate views:NotesPage}" />
        <ShellContent Title="About"
                      Icon="icon_about.png"
                      ContentTemplate="{DataTemplate views:AboutPage}" />
    </TabBar>
</Shell>

```

This XAML consists of two main objects:

- `TabBar`. The `TabBar` represents the bottom tab bar, and should be used when the navigation pattern for the application uses bottom tabs. The `TabBar` object is a child of the `Shell` object.
- `ShellContent`, which represents the `ContentPage` objects for each tab in the `TabBar`. Each `ShellContent` object is a child of the `TabBar` object.

These objects don't represent any user interface, but rather the organization of the application's visual hierarchy. Shell will take these objects and produce the navigation user interface for the content. Therefore, the `AppShell` class defines two pages that are navigable from bottom tabs. The pages are created on demand, in response to navigation.

For more information about Shell applications, see [Xamarin.Forms Shell](#).

User interface

There are several 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 Notes application uses the `ContentPage` class to display single screens. For more information about pages, see [Xamarin.Forms Pages](#).
2. **Views** – Xamarin.Forms views are the controls displayed on the user interface, such as labels, buttons, and text entry boxes. The finished Notes application uses the `CollectionView`, `Editor`, and `Button` views. For more information about views, see [Xamarin.Forms Views](#).
3. **Layouts** – Xamarin.Forms layouts are containers used to compose views into logical structures. The Notes application uses the `StackLayout` class to arrange views in a vertical stack, and the `Grid` class to arrange buttons horizontally. For more information about layouts, see [Xamarin.Forms Layouts](#).

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

Layout

The Notes application uses the `StackLayout` to simplify cross-platform application development by automatically arranging views 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 layout the `NoteEntryPage`:

```
<?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="Notes.Views.NoteEntryPage"
    Title="Note Entry">
    ...
    <StackLayout Margin="{StaticResource PageMargin}">
        <Editor Placeholder="Enter your note"
            Text="{Binding Text}"
            HeightRequest="100" />
        <Grid>
            ...
        </Grid>
    </StackLayout>
</ContentPage>
```

By default the `StackLayout` assumes a vertical orientation. However, it can be changed to a horizontal orientation by setting the `StackLayout.Orientation` property to the `StackOrientation.Horizontal` enumeration member.

NOTE

The size of views can be set through the `HeightRequest` and `WidthRequest` properties.

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

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 `OnSaveButtonClicked` method in the code-behind for the `NoteEntryPage` class, which is executed in response to the `Clicked` event firing on the *Save* button.

```

async void OnSaveButtonClicked(object sender, EventArgs e)
{
    var note = (Note)BindingContext;
    note.Date = DateTime.UtcNow;
    if (!string.IsNullOrWhiteSpace(note.Text))
    {
        await App.Database.SaveNoteAsync(note);
    }
    await Shell.Current.GoToAsync("../");
}

```

The `OnSaveButtonClicked` method saves the note in the database, and navigates back to the previous page. For more information about navigation, see [Navigation](#).

NOTE

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 save button to the `OnSaveButtonClicked` method occurs in the XAML markup for the `NoteEntryPage` class:

```

<Button Text="Save"
        Clicked="OnSaveButtonClicked" />

```

Lists

The `CollectionView` is responsible for displaying a collection of items in a list. By default, list items are displayed vertically and each item is displayed in a single row.

The following code example shows the `CollectionView` from the `NotesPage`:

```

<CollectionView x:Name="collectionView"
               Margin="{StaticResource PageMargin}"
               SelectionMode="Single"
               SelectionChanged="OnSelectionChanged">
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical"
                           ItemSpacing="10" />
    </CollectionView.ItemsLayout>
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <StackLayout>
                <Label Text="{Binding Text}"
                      FontSize="Medium" />
                <Label Text="{Binding Date}"
                      TextColor="{StaticResource TertiaryColor}"
                      FontSize="Small" />
            </StackLayout>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>

```

The layout of each row in the `CollectionView` is defined within the `CollectionView.ItemTemplate` element, and uses data binding to display any notes that are retrieved by the application. The `CollectionView.ItemsSource` property is set to the data source, in `NotesPage.xaml.cs`:

```
protected override async void OnAppearing()
{
    base.OnAppearing();

    collectionView.ItemsSource = await App.Database.GetNotesAsync();
}
```

This code populates the `CollectionView` with any notes stored in the database, and is executed when the page appears.

When an item is selected in the `CollectionView`, the `SelectionChanged` event fires. An event handler, named `OnSelectionChanged`, is executed when the event fires:

```
async void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (e.CurrentSelection != null)
    {
        // ...
    }
}
```

The `SelectionChanged` event can access the object that was associated with the item through the `e.CurrentSelection` property.

For more information about the `CollectionView` class, see [Xamarin.Forms CollectionView](#).

Navigation

Navigation is performed in a Shell application by specifying a URI to navigate to. Navigation URIs have three components:

- A *route*, which defines the path to content that exists as part of the Shell visual hierarchy.
- A *page*. Pages that don't exist in the Shell visual hierarchy can be pushed onto the navigation stack from anywhere within a Shell application. For example, the `NoteEntryPage` isn't defined in the Shell visual hierarchy, but can be pushed onto the navigation stack as required.
- One or more *query parameters*. Query parameters are parameters that can be passed to the destination page while navigating.

A navigation URI doesn't have to include all three components, but when it does the structure is: `//route/page?queryParameters`

NOTE

Routes can be defined on elements in the Shell visual hierarchy via the `Route` property. However, if the `Route` property isn't set, such as in the Notes application, a route is generated at runtime.

For more information about Shell navigation, see [Xamarin.Forms Shell navigation](#).

Register routes

To navigate to a page that doesn't exist in the Shell visual hierarchy requires it to first be registered with the Shell routing system, using the `Routing.RegisterRoute` method. In the Notes application, this occurs in the `AppShell` constructor:

```
public partial class AppShell : Shell
{
    public AppShell()
    {
        // ...
        Routing.RegisterRoute(nameof(NoteEntryPage), typeof(NoteEntryPage));
    }
}
```

In this example, a route named `NoteEntryPage` is registered against the `NoteEntryPage` type. This page can then be navigated to using URI-based navigation, from anywhere in the application.

Perform navigation

Navigation is performed by the `GoToAsync` method, which accepts an argument that represents the route to navigate to:

```
await Shell.Current.GoToAsync("NoteEntryPage");
```

In this example, the `NoteEntryPage` is navigated to.

IMPORTANT

A navigation stack is created when a page that's not in the Shell visual hierarchy is navigated to.

When navigating to a page, data can be passed to the page as a query parameter:

```
async void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (e.CurrentSelection != null)
    {
        // Navigate to the NoteEntryPage, passing the ID as a query parameter.
        Note note = (Note)e.CurrentSelection.FirstOrDefault();
        await Shell.Current.GoToAsync($"{nameof(NoteEntryPage)}?{nameof(NoteEntryPage.ItemId)}={note.ID.ToString()}");
    }
}
```

This example retrieves the currently selected item in the `CollectionView` and navigates to the `NoteEntryPage`, with the value of `ID` property of the `Note` object being passed as a query parameter to the `NoteEntryPage.ItemId` property.

To receive the passed data, the `NoteEntryPage` class is decorated with the `QueryPropertyAttribute`

```
[QueryProperty(nameof(ItemId), nameof(ItemId))]
public partial class NoteEntryPage : ContentPage
{
    public string ItemId
    {
        set
        {
            LoadNote(value);
        }
    }
    // ...
}
```

The first argument for the `QueryPropertyAttribute` specifies that the `ItemId` property will receive the passed data, with the second argument specifying the query parameter id. Therefore, the `QueryPropertyAttribute` in the above example specifies that the `ItemId` property will receive the data passed in the `ItemId` query parameter from the URI in the `GoToAsync` method call. The `ItemId` property then calls the `LoadNote` method to retrieve the note from the device.

Backwards navigation is performed by specifying ".." as the argument to the `GoToAsync` method:

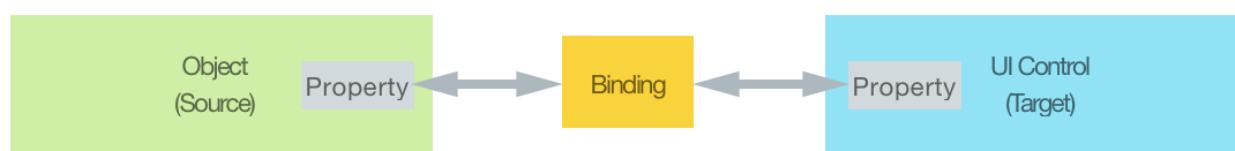
```
await Shell.Current.GoToAsync(..);
```

For more information about backwards navigation, see [Backwards navigation](#).

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 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, an `Editor` (*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 the Notes application, the binding target is the `Editor` that displays a note, while the `Note` instance set as the `BindingContext` of `NoteEntryPage` is the binding source. Initially, the `BindingContext` of the `NoteEntryPage` is set when the page constructor executes:

```
public NoteEntryPage()
{
    // ...
    BindingContext = new Note();
}
```

In this example, the page's `BindingContext` is set to a new `Note` when the `NoteEntryPage` is created. This handles the scenario of adding a new note to the application.

In addition, the page's `BindingContext` can also be set when navigation to the `NoteEntryPage` occurs, provided that an existing note was selected on the `NotesPage`:

```
[QueryProperty(nameof(ItemId), nameof(ItemId))]
public partial class NoteEntryPage : ContentPage
{
    public string ItemId
    {
        set
        {
            LoadNote(value);
        }
    }

    async void LoadNote(string itemId)
    {
        try
        {
            int id = Convert.ToInt32(itemId);
            // Retrieve the note and set it as the BindingContext of the page.
            Note note = await App.Database.GetNoteAsync(id);
            BindingContext = note;
        }
        catch (Exception)
        {
            Console.WriteLine("Failed to load note.");
        }
    }
    // ...
}
```

In this example, when page navigation occurs the page's `BindingContext` is set to the selected `Note` object after it's been retrieved from the database.

IMPORTANT

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 a `Note` instance, all of the children of the `ContentPage` have the same `BindingContext`, and can bind to public properties of the `Note` object.

The `Editor` in `NoteEntryPage` then binds to the `Text` property of the `Note` object:

```
<Editor Placeholder="Enter your note"
        Text="{Binding Text}" />
```

A binding between the `Editor.Text` property and the `Text` property of the *source* object is established. Changes made in the `Editor` will automatically be propagated to the `Note` object. Similarly, if changes are made to the `Note.Text` property, the Xamarin.Forms binding engine will also update the contents of the `Editor`. This is known as a *two-way binding*.

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

Styling

Xamarin.Forms applications often contain multiple visual elements that have an identical appearance. Setting the appearance of each visual element can be repetitive and error prone. Instead, styles can be created that define the appearance, and then applied to the required visual elements.

The `Style` class groups a collection of property values into one object that can then be applied to multiple visual element instances. Styles are stored in a `ResourceDictionary`, either at the application level, the page level,

or the view level. Choosing where to define a `Style` impacts where it can be used:

- `Style` instances defined at the application level can be applied throughout the application.
- `Style` instances defined at the page level can be applied to the page and to its children.
- `Style` instances defined at the view level can be applied to the view and to its children.

IMPORTANT

Any styles that are used throughout the application are stored in the application's resource dictionary to avoid duplication. However, XAML that's specific to a page shouldn't be included in the application's resource dictionary, as the 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](#).

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. The following code example shows a style from `NoteEntryPage`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Notes.Views.NoteEntryPage"
    Title="Note Entry">
    <ContentPage.Resources>
        <!-- Implicit styles -->
        <Style TargetType="{x:Type Editor}">
            <Setter Property="BackgroundColor"
                Value="{StaticResource AppBackgroundColor}" />
        </Style>
        ...
    </ContentPage.Resources>
    ...
</ContentPage>
```

This style is applied to any `Editor` instances on the page.

When creating a `Style`, the `TargetType` property is always required.

NOTE

Styling a Xamarin.Forms application is traditionally accomplished by using XAML styles. However, Xamarin.Forms also supports styling visual elements using Cascading Style Sheets (CSS). For more information, see [Styling Xamarin.Forms apps using Cascading Style Sheets \(CSS\)](#).

For more information about XAML styles, see [Styling Xamarin.Forms Apps using XAML Styles](#).

Test 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](#).

Next steps

This deep dive has examined the fundamentals of application development using Xamarin.Forms Shell.

Suggested next steps include reading about the following functionality:

- Xamarin.Forms Shell reduces the complexity of mobile application development by providing the fundamental features that most mobile applications require. For more information, see [Xamarin.Forms Shell](#).
- There are several control groups used to create the user interface of a Xamarin.Forms application. For more information, see [Controls Reference](#).
- Data binding is a technique for linking properties of two objects so that changes in one property are automatically reflected in the other property. For more information, see [Data Binding](#).
- Xamarin.Forms provides multiple page navigation experiences, depending upon the page type being used. For more information, see [Navigation](#).
- Styles help to reduce repetitive markup, and allow an applications appearance to be more easily changed. For more information, see [Styling Xamarin.Forms Apps](#).
- Data templates provide the ability to define the presentation of data on supported views. For more information, see [Data Templates](#).
- Effects also allow the native controls on each platform to be customized. Effects are created in platform-specific projects by subclassing the `PlatformEffect` class, and are consumed by attaching them to an appropriate Xamarin.Forms control. For more information, see [Effects](#).
- Each page, layout, and view 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](#).
- Shared code can access native functionality through the `DependencyService` class. For more information, see [Accessing Native Features with DependencyService](#).

Related links

- [Xamarin.Forms Shell](#)
- [eXtensible Application Markup Language \(XAML\)](#)
- [Data Binding](#)
- [Controls Reference](#)
- [Get Started Samples](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API reference](#)

Related video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Cross-Platform for Desktop Developers

8/4/2022 • 2 minutes to read • [Edit Online](#)

This section contains information to help WPF and Windows Forms developers to learn mobile app development with Xamarin, by cross-referencing their existing knowledge and experience to mobile idioms, and providing examples of porting desktop apps to mobile.

App Lifecycle Comparison

Understanding the differences between WPF and Xamarin.Forms app startup and background states.

UI Controls Comparison

Quick reference to find equivalent controls in Windows Forms, WPF, and Xamarin.Forms, including additional guidance on the differences between WPF and Xamarin.Forms.

Porting Guidance

Using the Portability Analyzer to help migrate desktop application code (excluding the user interface) to Xamarin.Forms.

Samples

Reference samples demonstrating enterprise application architecture and porting code from WPF to Xamarin.Forms.

Learn More



Xamarin for Java developers

8/4/2022 • 24 minutes to read • [Edit Online](#)

If you are a Java developer, you are well on your way to leveraging your skills and existing code on the Xamarin platform while reaping the code reuse benefits of C#. You will find that C# syntax is very similar to Java syntax, and that both languages provide very similar features. In addition, you'll discover features unique to C# that will make your development life easier.

Overview

This article provides an introduction to C# programming for Java developers, focusing primarily on the C# language features that you will encounter while developing Xamarin.Android applications. Also, this article explains how these features differ from their Java counterparts, and it introduces important C# features (relevant to Xamarin.Android) that are not available in Java. Links to additional reference material are included, so you can use this article as a "jumping off" point for further study of C# and .NET.

If you are familiar with Java, you will feel instantly at home with the syntax of C#. C# syntax is very similar to Java syntax – C# is a "curly brace" language like Java, C, and C++. In many ways, C# syntax reads like a superset of Java syntax, but with a few renamed and added keywords.

Many key characteristics of Java can be found in C#:

- Class-based object-oriented programming
- Strong typing
- Support for interfaces
- Generics
- Garbage collection
- Runtime compilation

Both Java and C# are compiled to an intermediate language that is run in a managed execution environment. Both C# and Java are statically-typed, and both languages treat strings as immutable types. Both languages use a single-rooted class hierarchy. Like Java, C# supports only single inheritance and does not allow for global methods. In both languages, objects are created on the heap using the `new` keyword, and objects are garbage-collected when they are no longer used. Both languages provide formal exception handling support with `try / catch` semantics. Both provide thread management and synchronization support.

However, there are many differences between Java and C#. For example:

- Java (as used on Android) does not support implicitly-typed local variables (C# supports the `var` keyword).
- In Java, you can pass parameters only by value, while in C# you can pass by reference as well as by value. (C# provides the `ref` and `out` keywords for passing parameters by reference; there is no equivalent to these in Java).
- Java does not support preprocessor directives like `#define`.
- Java does not support unsigned integer types, while C# provides unsigned integer types such as `ulong`, `uint`, `ushort` and `byte`.

- Java does not support operator overloading; in C# you can overload operators and conversions.
- In a Java `switch` statement, code can fall through into the next switch section, but in C# the end of every `switch` section must terminate the switch (the end of each section must close with a `break` statement).
- In Java, you specify the exceptions thrown by a method with the `throws` keyword, but C# has no concept of checked exceptions – the `throws` keyword is not supported in C#.
- C# supports Language-Integrated Query (LINQ), which lets you use the reserved words `from`, `select`, and `where` to write queries against collections in a way that is similar to database queries.

Of course, there are many more differences between C# and Java than can be covered in this article. Also, both Java and C# continue to evolve (for example, Java 8, which is not yet in the Android toolchain, supports C#-style lambda expressions) so these differences will change over time. Only the most important differences currently encountered by Java developers new to Xamarin.Android are outlined here.

- [Going from Java to C# Development](#) provides an introduction to the fundamental differences between C# and Java.
- [Object-Oriented Programming Features](#) outlines the most important object-oriented feature differences between the two languages.
- [Keyword Differences](#) provides a table of useful keyword equivalents, C#-only keywords, and links to C# keyword definitions.

C# brings many key features to Xamarin.Android that are not currently readily available to Java developers on Android. These features can help you to write better code in less time:

- [Properties](#) – With C#'s property system, you can access member variables safely and directly without having to write setter and getter methods.
- [Lambda Expressions](#) – In C# you can use anonymous methods (also called *lambdas*) to express your functionality more succinctly and more efficiently. You can avoid the overhead of having to write one-time-use objects, and you can pass local state to a method without having to add parameters.
- [Event Handling](#) – C# provides language-level support for *event-driven programming*, where an object can register to be notified when an event of interest occurs. The `event` keyword defines a multicast broadcast mechanism that a publisher class can use to notify event subscribers.
- [Asynchronous Programming](#) – The asynchronous programming features of C# (`async` / `await`) keep apps responsive. The language-level support of this feature makes async programming easy to implement and less error-prone.

Finally, Xamarin allows you to [leverage existing Java assets](#) via a technology known as *binding*. You can call your existing Java code, frameworks, and libraries from C# by making use of Xamarin's automatic binding generators. To do this, you simply create a static library in Java and expose it to C# via a binding.

NOTE

Android programming uses a specific version of the Java language that supports all Java 7 features [and a subset of Java 8](#).

Some features mentioned on this page (such as the `var` keyword in C#) are available in newer versions of Java (e.g. [var in Java 10](#)), but are still not available to Android developers.

Going from Java to C# development

The following sections outline the basic "getting started" differences between C# and Java; a later section describes the object-oriented differences between these languages.

Libraries vs. assemblies

Java typically packages related classes in `.jar` files. In C# and .NET, however, reusable bits of precompiled code are packaged into *assemblies*, which are typically packaged as `.dll` files. An assembly is a unit of deployment for C#/NET code, and each assembly is typically associated with a C# project. Assemblies contain intermediate code (IL) that is just-in-time compiled at runtime.

For more information about assemblies, see the [Assemblies and the Global Assembly Cache](#) topic.

Packages vs. namespaces

C# uses the `namespace` keyword to group related types together; this is similar to Java's `package` keyword.

Typically, a Xamarin.Android app will reside in a namespace created for that app. For example, the following C# code declares the `WeatherApp` namespace wrapper for a weather-reporting app:

```
namespace WeatherApp
{
    ...
}
```

Importing types

When you make use of types defined in external namespaces, you import these types with a `using` statement (which is very similar to the Java `import` statement). In Java, you might import a single type with a statement like the following:

```
import javax.swing.JButton
```

You might import an entire Java package with a statement like this:

```
import javax.swing.*
```

The C# `using` statement works in a very similar way, but it allows you to import an entire package without specifying a wildcard. For example, you will often see a series of `using` statements at the beginning of Xamarin.Android source files, as seen in this example:

```
using System;
using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;
using System.Net;
using System.IO;
using System.Json;
using System.Threading.Tasks;
```

These statements import functionality from the `System`, `Android.App`, `Android.Content`, etc. namespaces.

Generics

Both Java and C# support *generics*, which are placeholders that let you plug in different types at compile time. However, generics work slightly differently in C#. In Java, [type erasure](#) makes type information available only at compile time, but not at run time. By contrast, the .NET common language runtime (CLR) provides explicit support for generic types, which means that C# has access to type information at runtime. In day-to-day

Xamarin.Android development, the importance of this distinction is not often apparent, but if you are using [reflection](#), you will depend on this feature to access type information at run time.

In Xamarin.Android, you will often see the generic method `FindViewById<TextView>` used to get a reference to a layout control. This method accepts a generic type parameter that specifies the type of control to look up. For example:

```
TextView label = FindViewById<TextView> (Resource.Id.Label);
```

In this code example, `FindViewById` gets a reference to the `TextView` control that is defined in the layout as `Label`, then returns it as a `TextView` type.

For more information about generics, see the [Generics](#) topic. Note that there are some limitations in Xamarin.Android support for generic C# classes; for more information, see [Limitations](#).

Object-oriented programming features

Both Java and C# use very similar object-oriented programming idioms:

- All classes are ultimately derived from a single root object – all Java objects derive from `java.lang.Object`, while all C# objects derive from `System.Object`.
- Instances of classes are reference types.
- When you access the properties and methods of an instance, you use the "`.`" operator.
- All class instances are created on the heap via the `new` operator.
- Because both languages use garbage collection, there is no way to explicitly release unused objects (i.e., there is not a `delete` keyword as there is in C++).
- You can extend classes through inheritance, and both languages only allow a single base class per type.
- You can define interfaces, and a class can inherit from (i.e., implement) multiple interface definitions.

However, there are also some important differences:

- Java has two powerful features that C# does not support: anonymous classes and inner classes. (However, C# does allow nesting of class definitions – C#'s nested classes are like Java's static nested classes.)
- C# supports C-style structure types (`struct`) while Java does not.
- In C#, you can implement a class definition in separate source files by using the `partial` keyword.
- C# interfaces cannot declare fields.
- C# uses C++-style destructor syntax to express finalizers. The syntax is different from Java's `finalize` method, but the semantics are nearly the same. (Note that in C#, destructors automatically call the base-class destructor – in contrast to Java where an explicit call to `super.finalize` is used.)

Class inheritance

To extend a class in Java, you use the `extends` keyword. To extend a class in C#, you use a colon (`:`) to indicate derivation. For example, in Xamarin.Android apps, you will often see class derivations that resemble the following code fragment:

```
public class MainActivity : Activity
{
    ...
}
```

In this example, `MainActivity` inherits from the `Activity` class.

To declare support for an interface in Java, you use the `implements` keyword. However, in C#, you simply add interface names to the list of classes to inherit from, as shown in this code fragment:

```
public class SensorsActivity : Activity, ISensorEventListener
{
    ...
}
```

In this example, `SensorsActivity` inherits from `Activity` and implements the functionality declared in the `ISensorEventListener` interface. Note that the list of interfaces must come after the base class (or you will get a compile-time error). By convention, C# interface names are prepended with an upper-case "I"; this makes it possible to determine which classes are interfaces without requiring an `implements` keyword.

When you want to prevent a class from being further subclassed in C#, you precede the class name with `sealed` – in Java, you precede the class name with `final`.

For more about C# class definitions, see the [Classes](#) and [Inheritance](#) topics.

Properties

In Java, mutator methods (setters) and inspector methods (getters) are often used to control how changes are made to class members while hiding and protecting these members from outside code. For example, the Android `TextView` class provides `getText` and `setText` methods. C# provides a similar but more direct mechanism known as *properties*. Users of a C# class can access a property in the same way as they would access a field, but each access actually results in a method call that is transparent to the caller. This "under the covers" method can implement side effects such as setting other values, performing conversions, or changing object state.

Properties are often used for accessing and modifying UI (user interface) object members. For example:

```
int width = rulerView.MeasuredWidth;
int height = rulerView.MeasuredHeight;
...
rulerView.DrawingCacheEnabled = true;
```

In this example, `width` and `height` values are read from the `rulerView` object by accessing its `MeasuredWidth` and `MeasuredHeight` properties. When these properties are read, values from their associated (but hidden) field values are fetched behind the scenes and returned to the caller. The `rulerView` object may store width and height values in one unit of measurement (say, pixels) and convert these values on-the-fly to a different unit of measurement (say, millimeters) when the `MeasuredWidth` and `MeasuredHeight` properties are accessed.

The `rulerView` object also has a property called `DrawingCacheEnabled` – the example code sets this property to `true` to enable the drawing cache in `rulerView`. Behind the scenes, an associated hidden field is updated with the new value, and possibly other aspects of `rulerView` state are modified. For example, when `DrawingCacheEnabled` is set to `false`, `rulerView` may also erase any drawing cache information already accumulated in the object.

Access to properties can be read/write, read-only, or write-only. Also, you can use different access modifiers for reading and writing. For example, you can define a property that has public read access but private write access.

For more information about C# properties, see the [Properties](#) topic.

Calling base class methods

To call a base-class constructor in C#, you use a colon (`:`) followed by the `base` keyword and an initializer list; this `base` constructor call is placed immediately after the derived constructor parameter list. The base-class constructor is called on entry to the derived constructor; the compiler inserts the call to the base constructor at the start of the method body. The following code fragment illustrates a base constructor called from a derived constructor in a Xamarin.Android app:

```
public class PictureLayout : ViewGroup
{
    ...
    public PictureLayout (Context context)
        : base (context)
    {
        ...
    }
    ...
}
```

In this example, the `PictureLayout` class is derived from the `ViewGroup` class. The `PictureLayout` constructor shown in this example accepts a `context` argument and passes it to the `ViewGroup` constructor via the `base(context)` call.

To call a base-class method in C#, use the `base` keyword. For example, Xamarin.Android apps often make calls to base methods as shown here:

```
public class MainActivity : Activity
{
    ...
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
```

In this case, the `OnCreate` method defined by the derived class (`MainActivity`) calls the `OnCreate` method of the base class (`Activity`).

Access modifiers

Java and C# both support the `public`, `private`, and `protected` access modifiers. However, C# supports two additional access modifiers:

- `internal` – The class member is accessible only within the current assembly.
- `protected internal` – The class member is accessible within the defining assembly, the defining class, and derived classes (derived classes both inside and outside the assembly have access).

For more information about C# access modifiers, see the [Access Modifiers](#) topic.

Virtual and override methods

Both Java and C# support *polymorphism*, the ability to treat related objects in the same manner. In both languages, you can use a base-class reference to refer to a derived-class object, and the methods of a derived class can override the methods of its base classes. Both languages have the concept of a *virtual* method, a method in a base class that is designed to be replaced by a method in a derived class. Like Java, C# supports `abstract` classes and methods.

However, there are some differences between Java and C# in how you declare virtual methods and override them:

- In C#, methods are non-virtual by default. Parent classes must explicitly label which methods are to be overridden by using the `virtual` keyword. By contrast, all methods in Java are virtual methods by default.
- To prevent a method from being overridden in C#, you simply leave off the `virtual` keyword. By contrast, Java uses the `final` keyword to mark a method with "override is not allowed."
- C# derived classes must use the `override` keyword to explicitly indicate that a virtual base-class method is being overridden.

For more information about C#'s support for polymorphism, see the [Polymorphism](#) topic.

Lambda expressions

C# makes it possible to create *closures*: inline, anonymous methods that can access the state of the method in which they are enclosed. Using lambda expressions, you can write fewer lines of code to implement the same functionality that you might have implemented in Java with many more lines of code.

Lambda expressions make it possible for you to skip the extra ceremony involved in creating a one-time-use class or anonymous class as you would in Java – instead, you can just write the business logic of your method code inline. Also, because lambdas have access to the variables in the surrounding method, you don't have to create a long parameter list to pass state to your method code.

In C#, lambda expressions are created with the `=>` operator as shown here:

```
(arg1, arg2, ...) => {
    // implementation code
};
```

In Xamarin.Android, lambda expressions are often used for defining event handlers. For example:

```
button.Click += (sender, args) => {
    clickCount += 1;      // access variable in surrounding code
    button.Text = string.Format ("Clicked {0} times.", clickCount);
};
```

In this example, the lambda expression code (the code within the curly braces) increments a click count and updates the `button` text to display the click count. This lambda expression is registered with the `button` object as a click event handler to be called whenever the button is tapped. (Event handlers are explained in more detail below.) In this simple example, the `sender` and `args` parameters are not used by the lambda expression code, but they are required in the lambda expression to meet the method signature requirements for event registration. Under the hood, the C# compiler translates the lambda expression into an anonymous method that is called whenever button click events take place.

For more information about C# and lambda expressions, see the [Lambda Expressions](#) topic.

Event handling

An *event* is a way for an object to notify registered subscribers when something interesting happens to that object. Unlike in Java, where a subscriber typically implements a `Listener` interface that contains a callback method, C# provides language-level support for event handling through *delegates*. A *delegate* is like an object-oriented type-safe function pointer – it encapsulates an object reference and a method token. If a client object wants to subscribe to an event, it creates a delegate and passes the delegate to the notifying object. When the event occurs, the notifying object invokes the method represented by the delegate object, notifying the

subscribing client object of the event. In C#, event handlers are essentially nothing more than methods that are invoked through delegates.

For more information about delegates, see the [Delegates](#) topic.

In C#, events are *multicast*; that is, more than one listener can be notified when an event takes place. This difference is observed when you consider the syntactical differences between Java and C# event registration. In Java you call `SetXXXListener` to register for event notifications; in C# you use the `+=` operator to register for event notifications by "adding" your delegate to the list of event listeners. In Java, you call `SetXXXListener` to unregister, while in C# you use the `-=` to "subtract" your delegate from the list of listeners.

In Xamarin.Android, events are often used for notifying objects when a user does something to a UI control. Normally, a UI control will have members that are defined using the `event` keyword; you attach your delegates to these members to subscribe to events from that UI control.

To subscribe to an event:

1. Create a delegate object that refers to the method that you want to be invoked when the event occurs.
2. Use the `+=` operator to attach your delegate to the event you are subscribing to.

The following example defines a delegate (with an explicit use of the `delegate` keyword) to subscribe to button clicks. This button-click handler starts a new activity:

```
startActivityButton.Click += delegate {
    Intent intent = new Intent (this, typeof (MyActivity));
    StartActivity (intent);
};
```

However, you also can use a lambda expression to register for events, skipping the `delegate` keyword altogether. For example:

```
startActivityButton.Click += (sender, e) => {
    Intent intent = new Intent (this, typeof (MyActivity));
    StartActivity (intent);
};
```

In this example, the `startActivityButton` object has an event that expects a delegate with a certain method signature: one that accepts sender and event arguments and returns void. However, because we don't want to go to the trouble to explicitly define such a delegate or its method, we declare the signature of the method with `(sender, e)` and use a lambda expression to implement the body of the event handler. Note that we have to declare this parameter list even though we aren't using the `sender` and `e` parameters.

It is important to remember that you can unsubscribe a delegate (via the `-=` operator), but you cannot unsubscribe a lambda expression – attempting to do so can cause memory leaks. Use the lambda form of event registration only when your handler will not unsubscribe from the event.

Typically, lambda expressions are used to declare event handlers in Xamarin.Android code. This shorthand way to declare event handlers may seem cryptic at first, but it saves an enormous amount of time when you are writing and reading code. With increasing familiarity, you become accustomed to recognizing this pattern (which occurs frequently in Xamarin.Android code), and you spend more time thinking about the business logic of your application and less time wading through syntactical overhead.

Asynchronous programming

Asynchronous programming is a way to improve the overall responsiveness of your application. Asynchronous programming features make it possible for the rest of your app code to continue running while some part of your app is blocked by a lengthy operation. Accessing the web, processing images, and reading/writing files are examples of operations that can cause an entire app to appear to freeze up if it is not written asynchronously.

C# includes language-level support for asynchronous programming via the `async` and `await` keywords. These language features make it very easy to write code that performs long-running tasks without blocking the main thread of your application. Briefly, you use the `async` keyword on a method to indicate that the code in the method is to run asynchronously and not block the caller's thread. You use the `await` keyword when you call methods that are marked with `async`. The compiler interprets the `await` as the point where your method execution is to be moved to a background thread (a task is returned to the caller). When this task completes, execution of the code resumes on the caller's thread at the `await` point in your code, returning the results of the `async` call. By convention, methods that run asynchronously have `Async` suffixed to their names.

In Xamarin.Android applications, `async` and `await` are typically used to free up the UI thread so that it can respond to user input (such as the tapping of a **Cancel** button) while a long-running operation takes place in a background task.

In the following example, a button click event handler causes an asynchronous operation to download an image from the web:

```
downloadButton.Click += downloadAsync;
...
async void downloadAsync(object sender, System.EventArgs e)
{
    WebClient = new WebClient ();
    var url = new Uri ("http://photojournal.jpl.nasa.gov/jpeg/PIA15416.jpg");
    byte[] bytes = null;

    bytes = await WebClient.DownloadDataTaskAsync(url);

    // display the downloaded image ...
}
```

In this example, when the user clicks the `downloadButton` control, the `downloadAsync` event handler creates a `WebClient` object and a `Uri` object to fetch an image from the specified URL. Next, it calls the `WebClient` object's `DownloadDataTaskAsync` method with this URL to retrieve the image.

Notice that the method declaration of `downloadAsync` is prefaced by the `async` keyword to indicate that it will run asynchronously and return a task. Also note that the call to `DownloadDataTaskAsync` is preceded by the `await` keyword. The app moves the execution of the event handler (starting at the point where `await` appears) to a background thread until `DownloadDataTaskAsync` completes and returns. Meanwhile, the app's UI thread can still respond to user input and fire event handlers for the other controls. When `DownloadDataTaskAsync` completes (which may take several seconds), execution resumes where the `bytes` variable is set to the result of the call to `DownloadDataTaskAsync`, and the remainder of the event handler code displays the downloaded image on the caller's (UI) thread.

For an introduction to `async / await` in C#, see the [Asynchronous Programming with Async and Await](#) topic. For more information about Xamarin support of asynchronous programming features, see [Async Support Overview](#).

Keyword differences

Many language keywords used in Java are also used in C#. There are also a number of Java keywords that have an equivalent but differently-named counterpart in C#, as listed in this table:

JAVA	C#	DESCRIPTION
<code>boolean</code>	<code>bool</code>	Used for declaring the boolean values true and false.
<code>extends</code>	<code>:</code>	Precedes the class and interfaces to inherit from.
<code>implements</code>	<code>:</code>	Precedes the class and interfaces to inherit from.
<code>import</code>	<code>using</code>	Imports types from a namespace, also used for creating a namespace alias.
<code>final</code>	<code>sealed</code>	Prevents class derivation; prevents methods and properties from being overridden in derived classes.
<code>instanceof</code>	<code>is</code>	Evaluates whether an object is compatible with a given type.
<code>native</code>	<code>extern</code>	Declares a method that is implemented externally.
<code>package</code>	<code>namespace</code>	Declares a scope for a related set of objects.
<code>T...</code>	<code>params T</code>	Specifies a method parameter that takes a variable number of arguments.
<code>super</code>	<code>base</code>	Used to access members of the parent class from within a derived class.
<code>synchronized</code>	<code>lock</code>	Wraps a critical section of code with lock acquisition and release.

Also, there are many keywords that are unique to C# and have no counterpart in the Java used on Android. Xamarin.Android code often makes use of the following C# keywords (this table is useful to refer to when you are reading through Xamarin.Android [sample code](#)):

C#	DESCRIPTION
<code>as</code>	Performs conversions between compatible reference types or nullable types.
<code>async</code>	Specifies that a method or lambda expression is asynchronous.
<code>await</code>	Suspends the execution of a method until a task completes.
<code>byte</code>	Unsigned 8-bit integer type.
<code>delegate</code>	Used to encapsulate a method or anonymous method.
<code>enum</code>	Declares an enumeration, a set of named constants.

C#	DESCRIPTION
event	Declares an event in a publisher class.
fixed	Prevents a variable from being relocated.
get	Defines an accessor method that retrieves the value of a property.
in	Enables a parameter to accept a less derived type in a generic interface.
object	An alias for the Object type in the .NET framework.
out	Parameter modifier or generic type parameter declaration.
override	Extends or modifies the implementation of an inherited member.
partial	Declares a definition to be split into multiple files, or splits a method definition from its implementation.
readonly	Declares that a class member can be assigned only at declaration time or by the class constructor.
ref	Causes an argument to be passed by reference rather than by value.
set	Defines an accessor method that sets the value of a property.
string	Alias for the String type in the .NET framework.
struct	A value type that encapsulates a group of related variables.
typeof	Obtains the type of an object.
var	Declares an implicitly-typed local variable.
value	References the value that client code wants to assign to a property.
virtual	Allows a method to be overridden in a derived class.

Interoperating with existing java code

If you have existing Java functionality that you do not want to convert to C#, you can reuse your existing Java libraries in Xamarin.Android applications via two techniques:

- **Create a Java Bindings Library** – Using this approach, you use Xamarin tools to generate C# wrappers around Java types. These wrappers are called *bindings*. As a result, your Xamarin.Android application can use your *.jar* file by calling into these wrappers.
- **Java Native Interface** – The *Java Native Interface* (JNI) is a framework that makes it possible for C#

apps to call or be called by Java code.

For more information about these techniques, see [Java Integration Overview](#).

Further reading

The MSDN [C# Programming Guide](#) is a great way to get started in learning the C# programming language, and you can use the [C# Reference](#) to look up particular C# language features.

In the same way that Java knowledge is at least as much about familiarity with the Java class libraries as knowing the Java language, practical knowledge of C# requires some familiarity with the .NET framework.

Microsoft's [Moving to C# and the .NET Framework, for Java Developers](#) learning packet is a good way to learn more about the .NET framework from a Java perspective (while gaining a deeper understanding of C#).

When you are ready to tackle your first Xamarin.Android project in C#, our [Hello, Android](#) series can help you build your first Xamarin.Android application and further advance your understanding of the fundamentals of Android application development with Xamarin.

Summary

This article provided an introduction to the Xamarin.Android C# programming environment from a Java developer's perspective. It pointed out the similarities between C# and Java while explaining their practical differences. It introduced assemblies and namespaces, explained how to import external types, and provided an overview of the differences in access modifiers, generics, class derivation, calling base-class methods, method overriding, and event handling. It introduced C# features that are not available in Java, such as properties, `async / await` asynchronous programming, lambdas, C# delegates, and the C# event handling system. It included tables of important C# keywords, explained how to interoperate with existing Java libraries, and provided links to related documentation for further study.

Related links

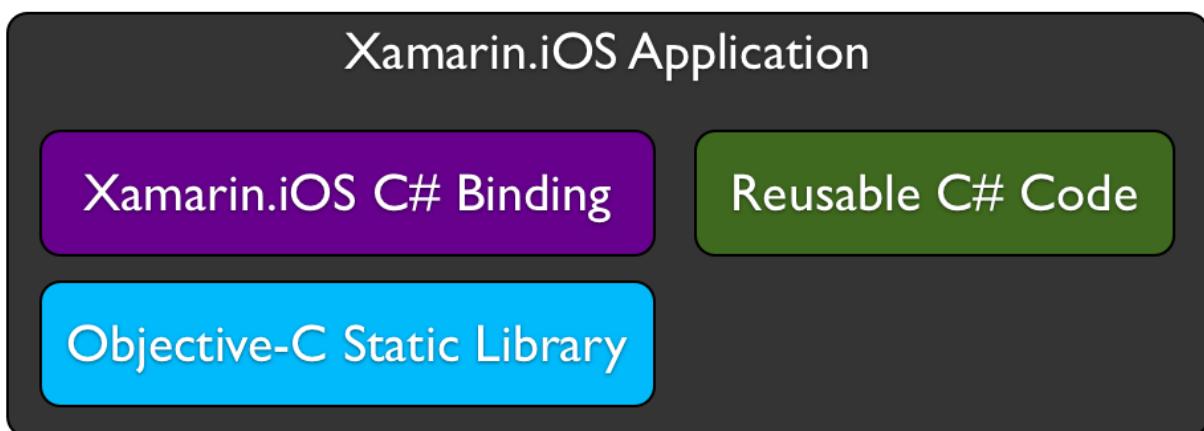
- [Java Integration Overview](#)
- [C# Programming Guide](#)
- [C# Reference](#)
- [Moving to C# and the .NET Framework, for Java Developers](#)

Xamarin for Objective-C Developers

8/4/2022 • 2 minutes to read • [Edit Online](#)

Xamarin offers a path for developers targeting iOS to move their non-user interface code to platform agnostic C# so that it can be used anywhere C# is available, including Android via Xamarin.Android and the various flavors of Windows. However, just because you use C# with Xamarin doesn't mean you can't leverage existing skills and Objective-C code. In fact, knowing Objective-C makes you a better Xamarin.iOS developer because Xamarin exposes all the native iOS and OS X platform APIs you know and love, such as UIKit, Core Animation, Core Foundation and Core Graphics to name a few. At the same time, you get the power of the C# language, including features like LINQ and Generics, as well as rich .NET base class libraries to use in your native applications.

Additionally, Xamarin allows you to leverage existing Objective-C assets via a technology known as bindings. You simply create a static library in Objective-C and expose it to C# via a binding, as illustrated in the following diagram:



This doesn't need to be limited to non-UI code. Bindings can expose user interface code developed in Objective-C as well.

Transitioning from Objective-C

You'll find a plethora of information on our documentation site to help ease the transition to Xamarin, showing how to integrate C# code with what you already know. Some highlights to get you started include:

- [C# Primer for Objective-C Developers](#) - A short primer for Objective-C developers looking to move to Xamarin and the C# language.
- [Walkthrough: Binding an Objective-C Library](#) - A step-by-step walkthrough for reusing existing Objective-C code in a Xamarin.iOS application.

Binding Objective-C

Once you have a grasp of how C# compares to Objective-C and have worked through the binding walkthrough above, you'll be in good shape for transitioning to the Xamarin platform. As a follow up, more detailed information on Xamarin.iOS binding technologies, including a comprehensive binding reference is available in the [Binding Objective-C](#) section.

Cross-Platform Development

Finally, after moving to Xamarin.iOS, you'll want to check out the cross-platform guidance we have, including case studies of reference applications we have developed, along with best practices for creating reusable, cross-platform code contained in the [Building Cross-Platform Applications section](#).

Xamarin.Forms XAML Basics

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The eXtensible Application Markup Language (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 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 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.

Within a 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.

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 you can use [XAML Hot Reload](#) in Visual Studio 2019 or Visual Studio for Mac to view your screen designs as you edit them. 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. 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](#)
- [Creating Mobile Apps book](#)
- [Xamarin.Forms Samples](#)

Related video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Part 1. Getting Started with XAML

8/4/2022 • 15 minutes to read • [Edit Online](#)

 [Download the sample](#)

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.

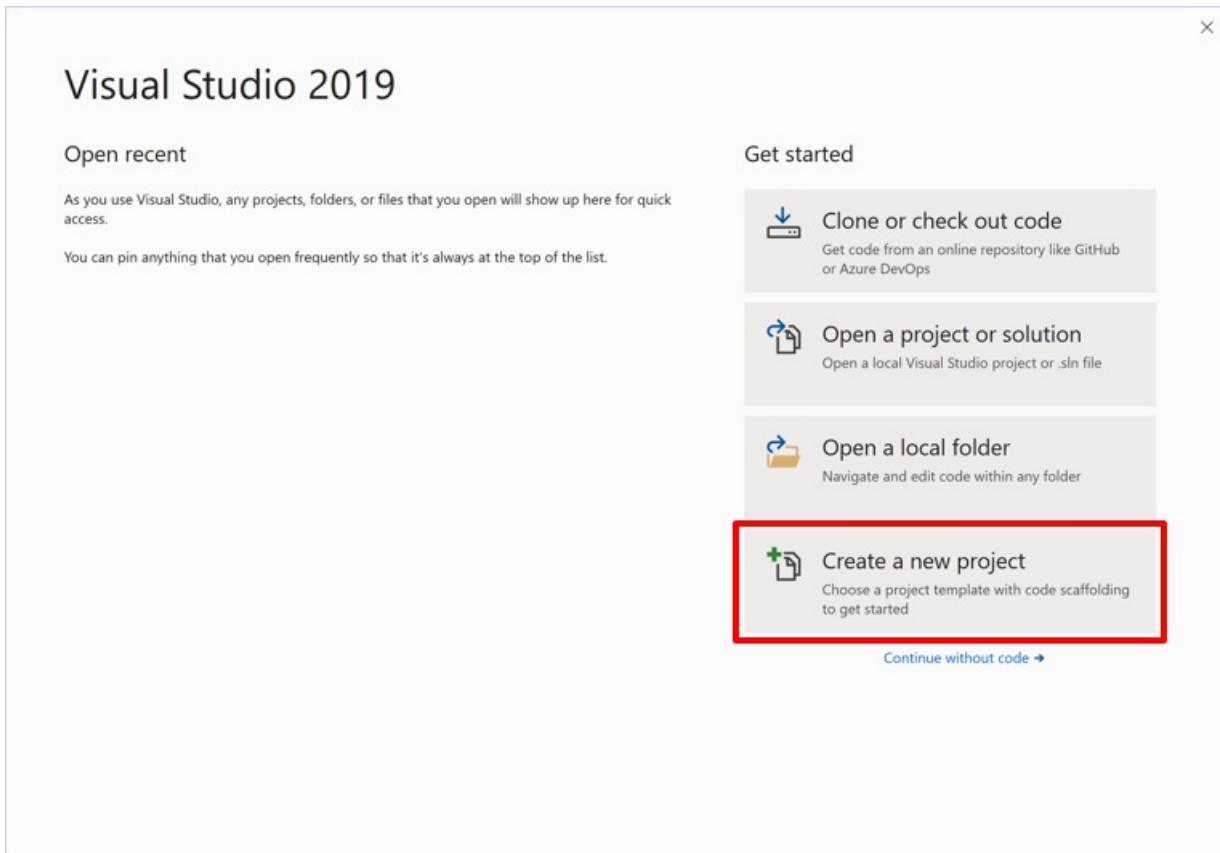
The code-behind file provides code support for the markup. Together, these two files contribute to a new class definition that includes child views and property initialization. Within the XAML file, classes and properties are referenced with XML elements and attributes, and links between the markup and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

Creating the Solution

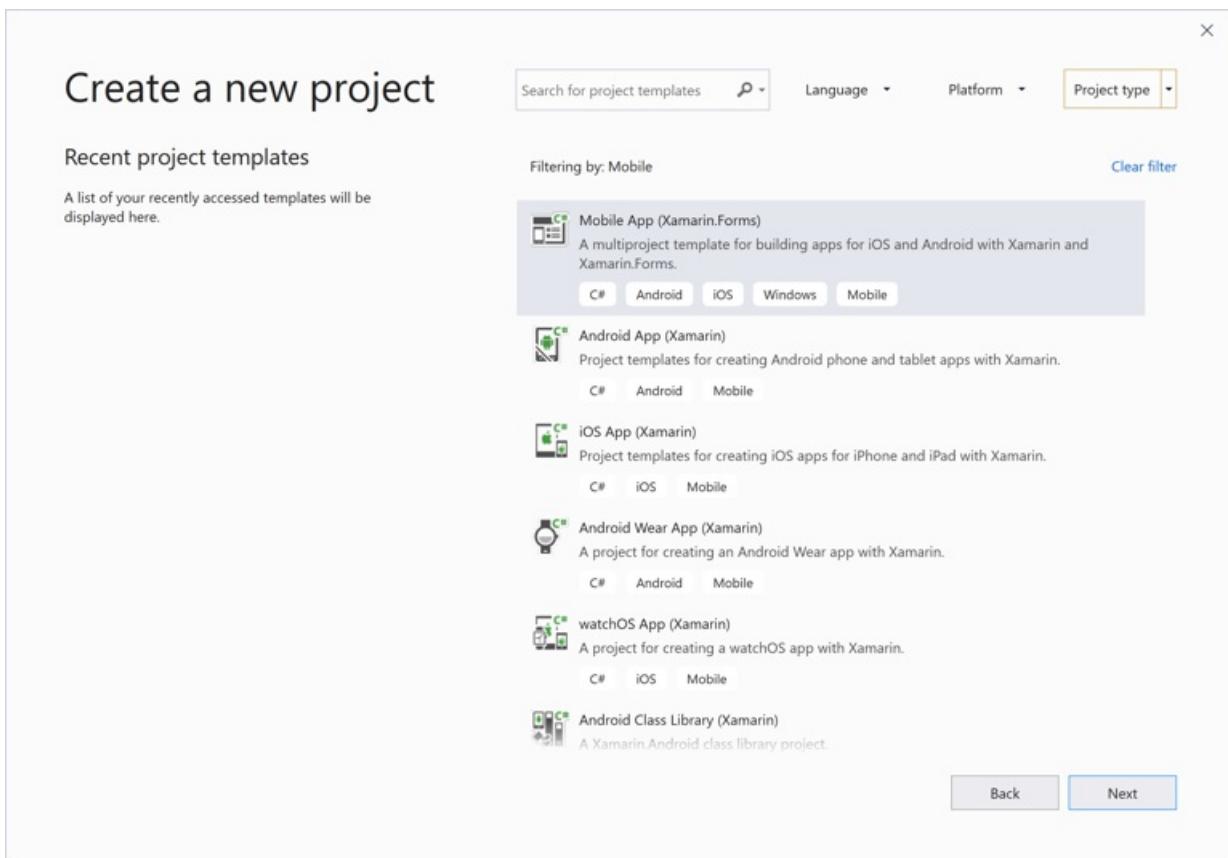
To begin editing your first XAML file, use Visual Studio or Visual Studio for Mac to create a new Xamarin.Forms solution. (Select the tab below corresponding to your environment.)

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

In Windows, launch Visual Studio 2019, and in the start window click **Create a new project** to create a new project:

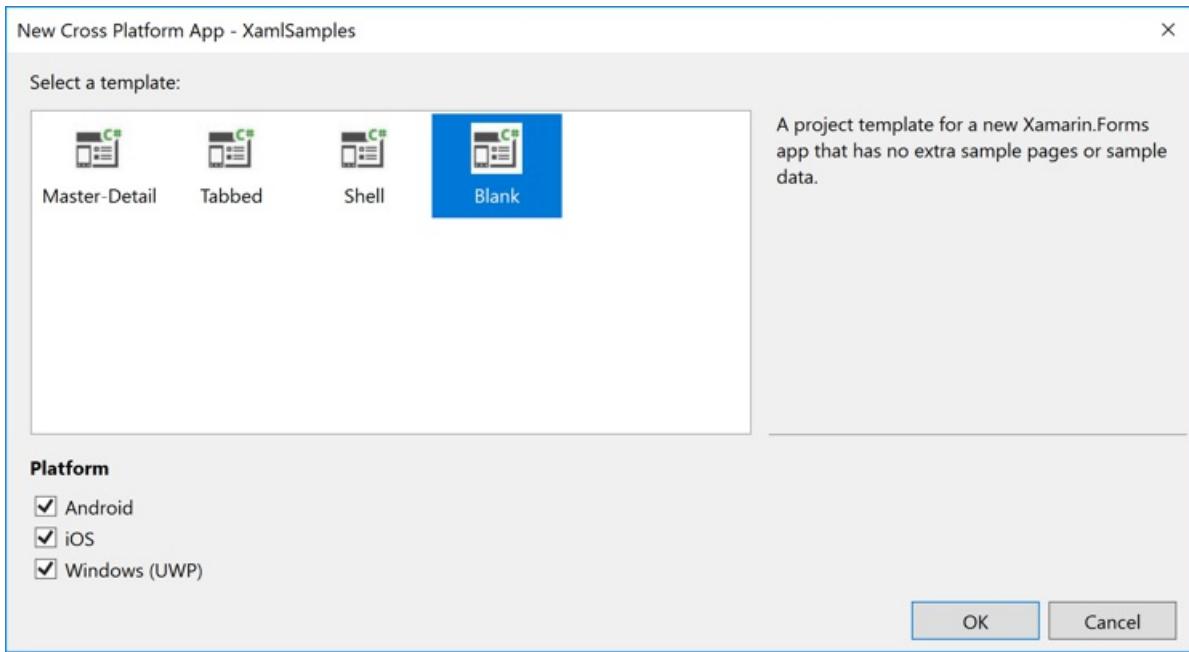


In the **Create a new project** window, select **Mobile** in the **Project type** drop down, select the **Mobile App (Xamarin.Forms)** template, and click the **Next** button:



In the **Configure your new project** window, set the **Project name** to **XamlSamples** (or whatever you prefer), and click the **Create** button.

In the **New Cross Platform App** dialog, click **Blank**, and click the **OK** button:



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 URLs, the first seemingly on Xamarin's web site and the second on Microsoft's. Don't bother checking what those URLs point to. There's nothing there. They are simply URLs 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();
        }
    }
}

```

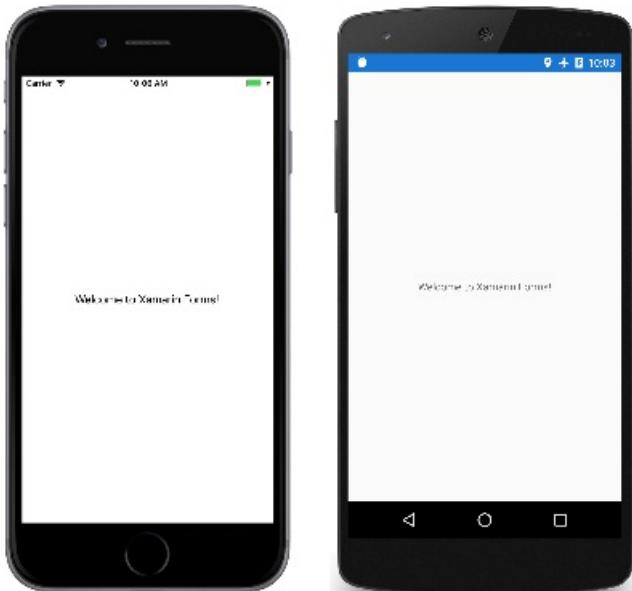
The `MainPage` class derives from `ContentPage`, but notice the `partial` class definition. This suggests that there should be another partial class definition for `MainPage`, but where is it? And what is that `InitializeComponent` method?

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:

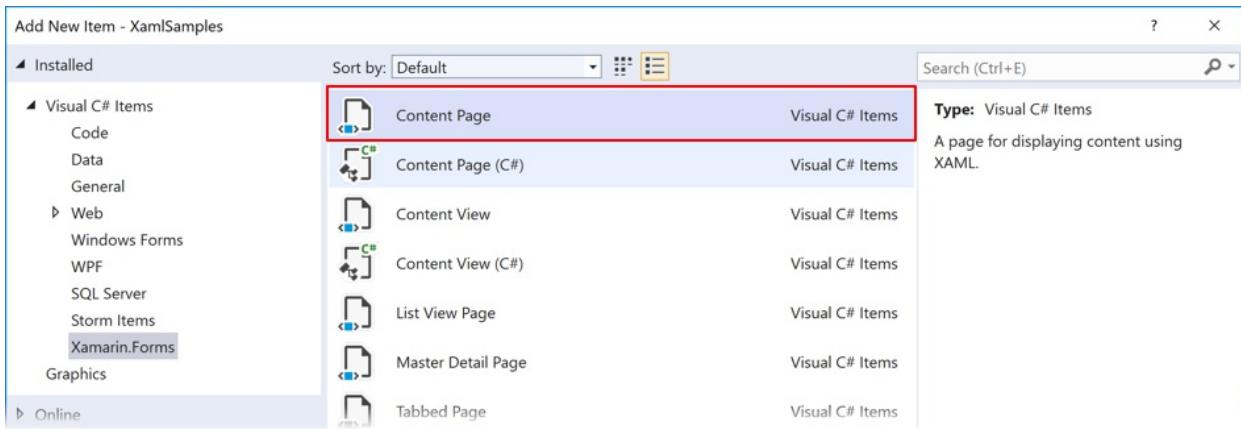


For more interesting visuals, all you need is more interesting XAML.

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, right-click, and select **Add > New Item....** In the **Add New Item** dialog, select **Visual C# Items > Xamarin.Forms > Content Page (C#)** (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**:



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 to 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="#rrggbbaa"
```

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

```
TextColor="#aarrggbbaa"
```

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`:

```

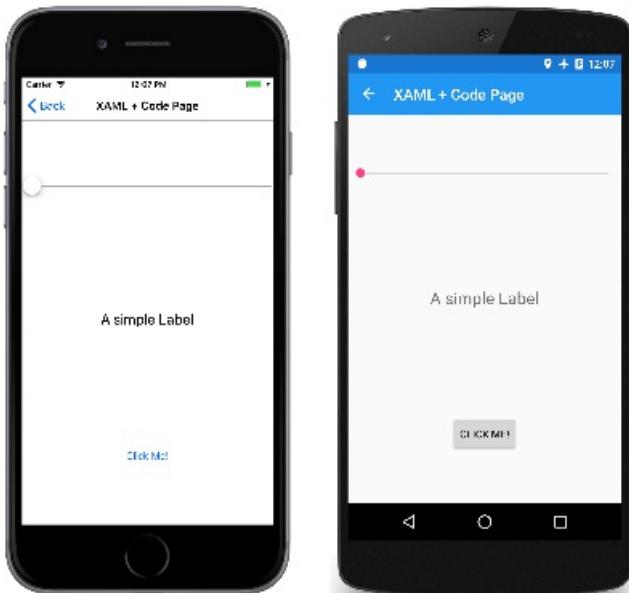
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.XamlPlusCodePage"
    Title="XAML + Code Page">
    <StackLayout>
        <Slider VerticalOptions="CenterAndExpand" />

        <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

8/4/2022 • 10 minutes to read • [Edit Online](#)



[Download the sample](#)

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 property 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" />

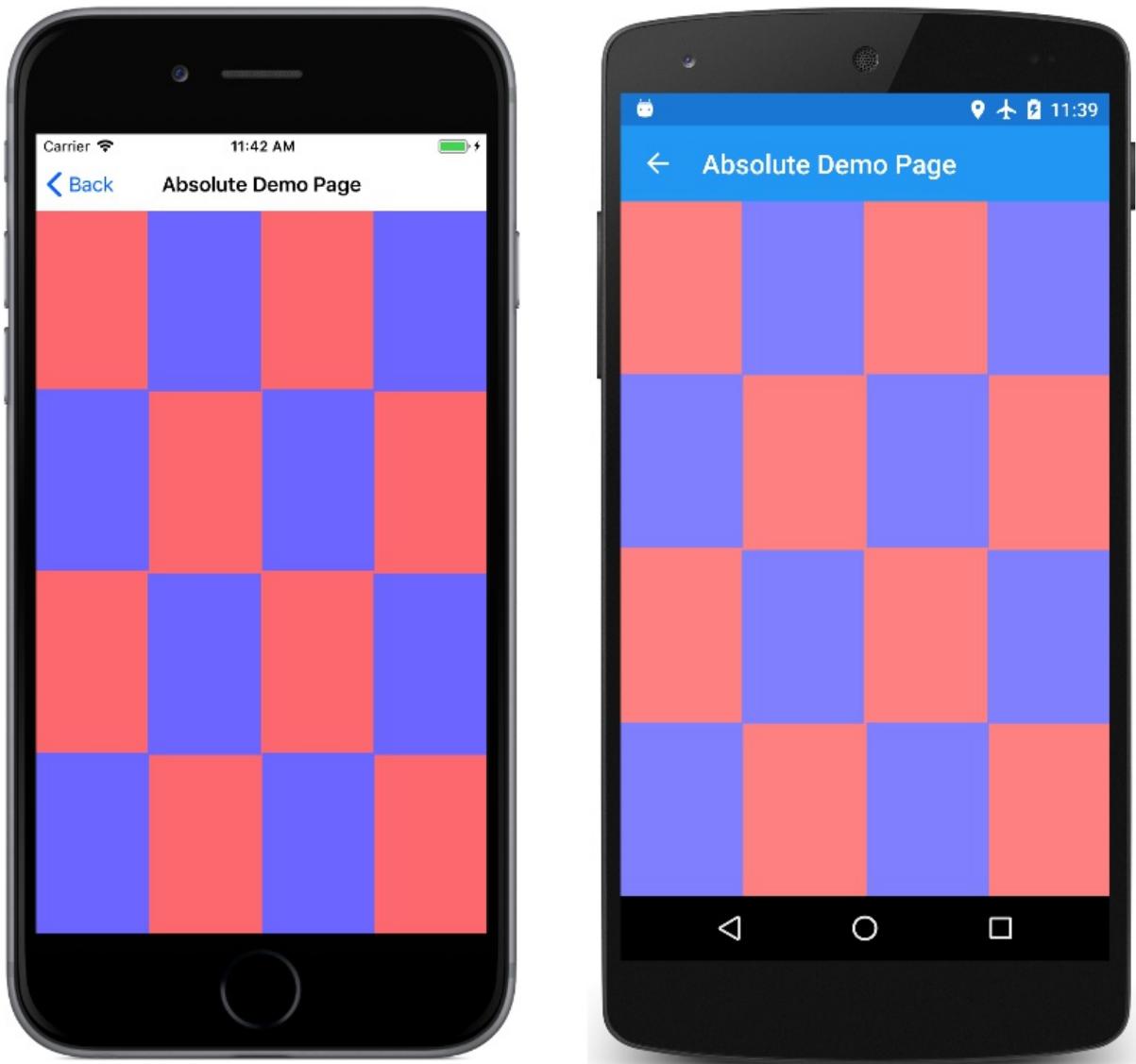
        <BoxView Color="#8080FF"
            AbsoluteLayout.LayoutBounds="0.33, 0.67, 0.25, 0.25"
            AbsoluteLayout.LayoutFlags="All" />

        <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>
```

NOTE

The `OnPlatform` markup extension can also be used in XAML 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. For more information, see [OnPlatform Markup Extension](#).

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

8/4/2022 • 11 minutes to read • [Edit Online](#)



[Download the sample](#)

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}</HorizontalOptions>
    <VerticalOptions>{StaticResource vertOptions}</VerticalOptions>
    <BorderWidth>{StaticResource borderWidth}</BorderWidth>
    <Rotation>{StaticResource rotationAngle}</Rotation>
    <TextColor>Red</TextColor>
    <FontSize>24</FontSize>
</Button>

```

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:Double x:Key="fontSize">24</x:Double>
        </ResourceDictionary>
    </ContentPage.Resources>

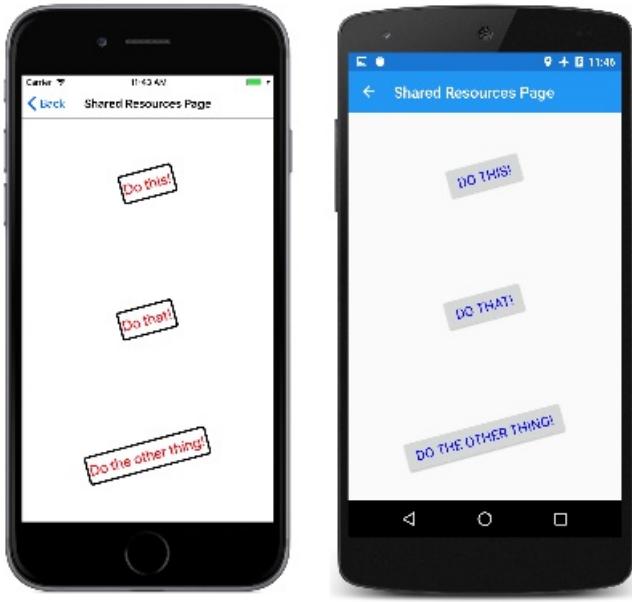
    <StackLayout>
        <Button Text="Do this!">
            HorizontalOptions="{StaticResource horzOptions}"
            VerticalOptions="{StaticResource vertOptions}"
            BorderWidth="{StaticResource borderWidth}"
            Rotation="{StaticResource rotationAngle}"
            TextColor="{StaticResource 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 `netstandard` assembly. Because this is another assembly, you must also specify the assembly name, in this case `netstandard`:

```
xmlns:sys="clr-namespace:System;assembly=netstandard"
```

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.

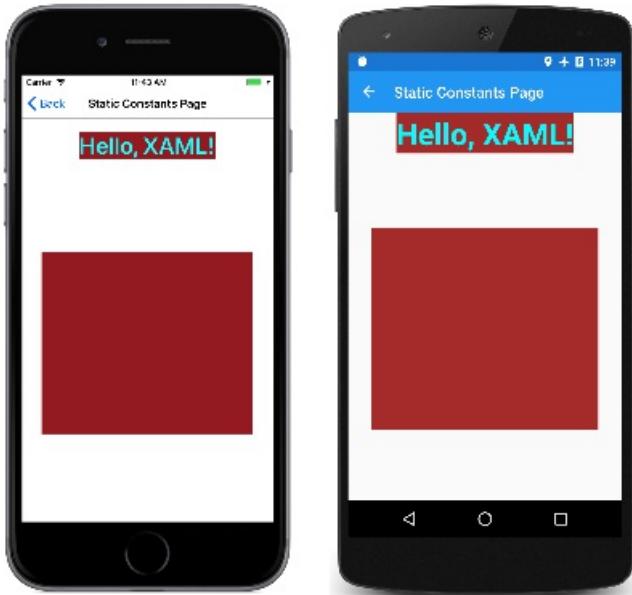
Both these namespace declarations are included in the `StaticConstantsPage` sample. Notice that the `BoxView` dimensions are set to `Math.PI` and `Math.E`, but scaled by a factor of 100:

```
<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=netstandard"
    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 `RelativeSource` markup extension is discussed in [Relative Bindings](#).

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}" />
    <!-- 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}"

```

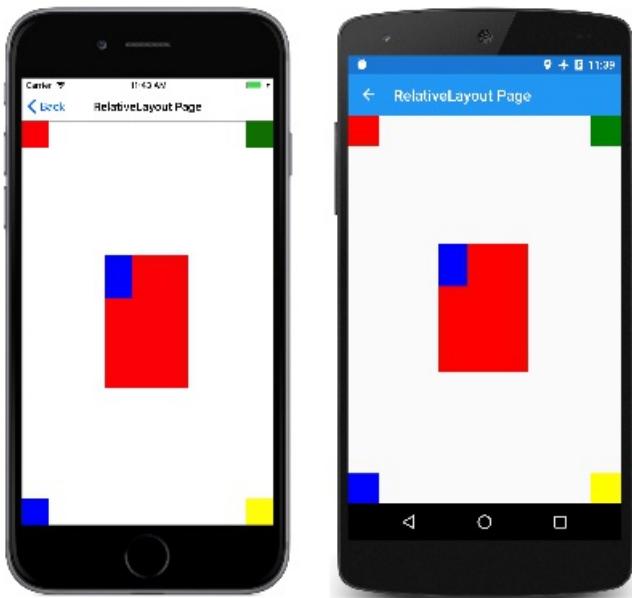
```

        Property=Width,
        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

8/4/2022 • 11 minutes to read • [Edit Online](#)



[Download the sample](#)

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'}"

```

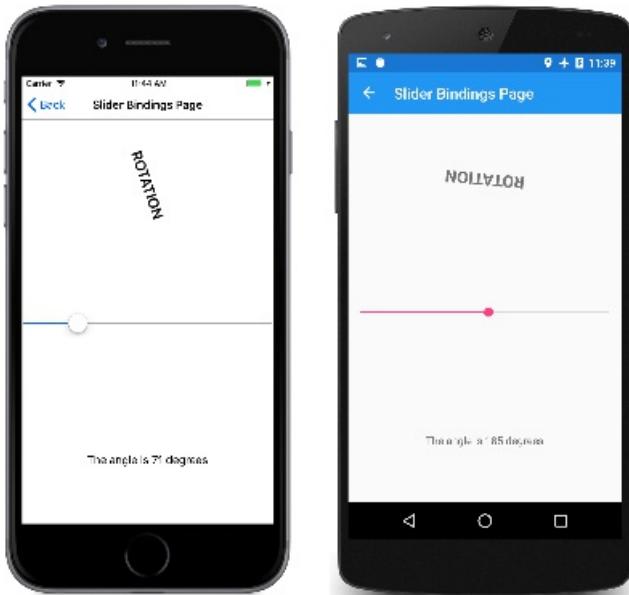
Do whatever is convenient.

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
- `OneTime` — data goes from source to target, but only when the `BindingContext` changes

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"
```

```

        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" />

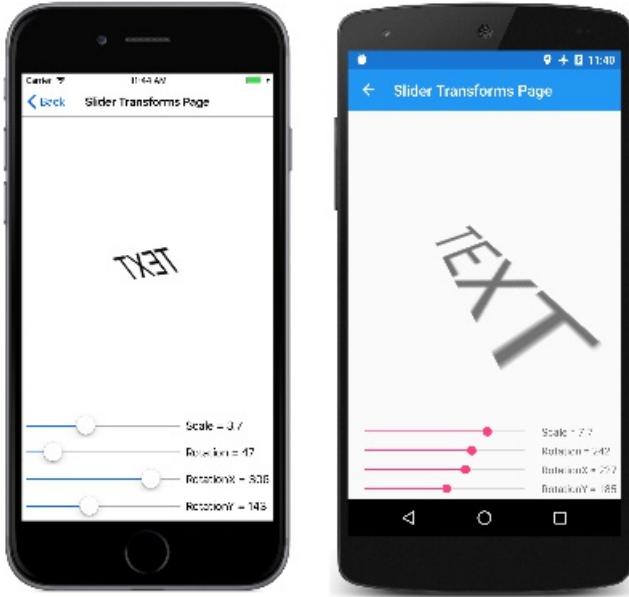
    <!-- Slider and identifying Label for RotationY -->
    <Slider x:Name="rotationYSlider"
        BindingContext="{x:Reference label}"
        Grid.Row="4" Grid.Column="0"
        Maximum="360"
        Value="{Binding RotationY, Mode=OneWayToSource}" />

    <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.

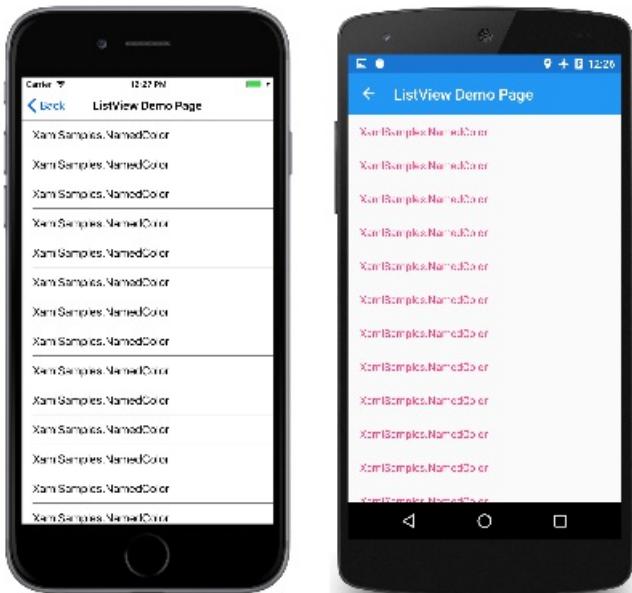
Setting the static `NamedColor.All` property to the `ItemsSource` of a `ListView` is easy using the `x:Static` markup extension:

```
<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:

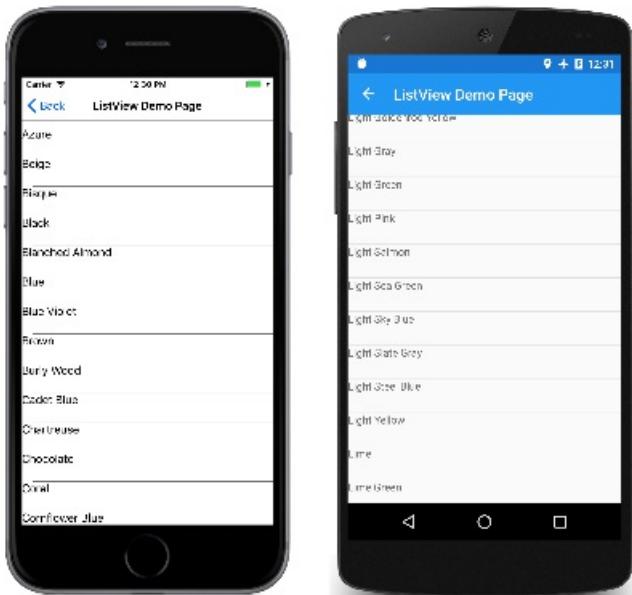
```
<ListView ItemsSource="{x:Static local:NamedColor.All}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <ViewCell.View>
                    <Label Text="{Binding FriendlyName}" />
                </ViewCell.View>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

NOTE

The binding source for cells, and children of cells, is the `ListView.ItemsSource` collection.

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>
                <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" />
                            </StackLayout>
                        </ViewCell>
                    </DataTemplate>
                </DataTemplate>
            </ListView.ItemTemplate>
        <ListView>
    </ContentPage>
```

```

        FontSize="Medium" />

        <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>
</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.

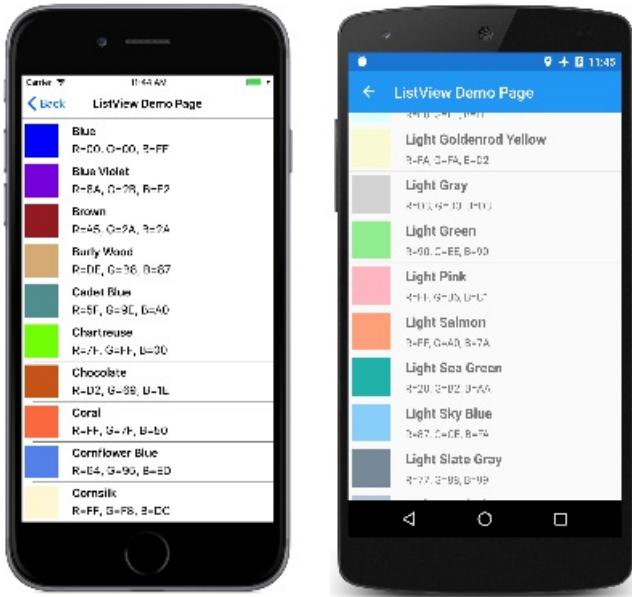
The converter is instantiated in the resource dictionary so it can be shared among multiple bindings:

```
<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

8/4/2022 • 13 minutes to read • [Edit Online](#)



[Download the sample](#)

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=netstandard"
```

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 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`:

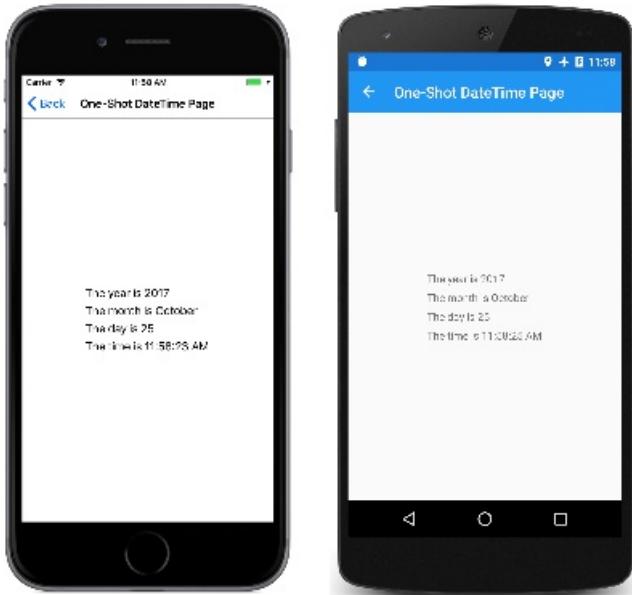
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:sys="clr-namespace:System;assembly=netstandard"
    x:Class="XamlSamples.OneShotDateTimePage"
    Title="One-Shot DateTime Page">

    <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>
```

The 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, 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`, 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">

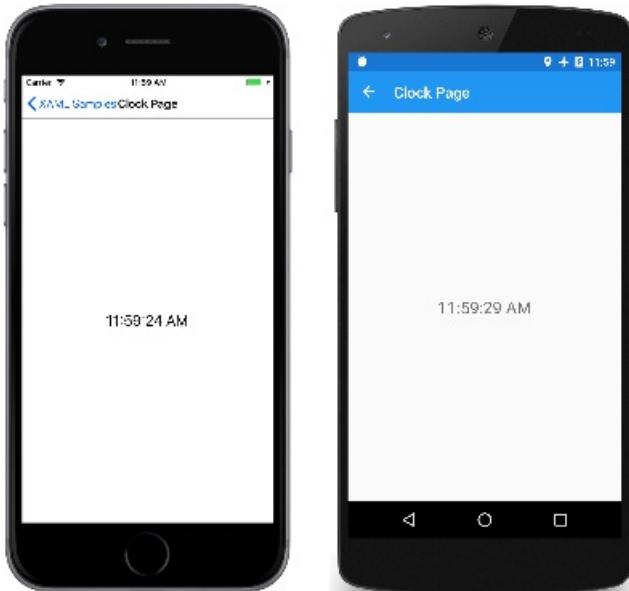
    <Label Text="{Binding DateTime, StringFormat='{0:T}'}"
        FontSize="Large"
        HorizontalOptions="Center"
        VerticalOptions="Center">
        <Label.BindingContext>
            <local:ClockViewModel />
        </Label.BindingContext>
    </Label>
</ContentPage>

```

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 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;  
                }  
            }  
        }  
    }  
}
```

```

        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 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}" />

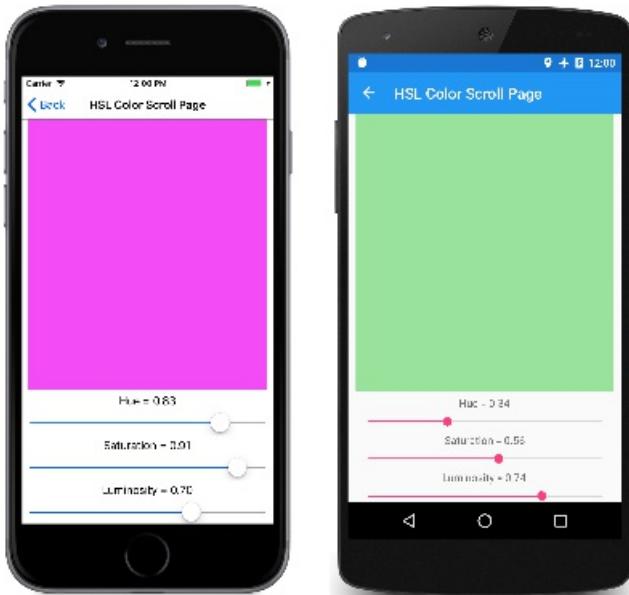
    <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}"
        HorizontalOptions="Center" />

    <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 with 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");
                }
            }
        }

        private void OnPropertyChanged(string propertyName)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

```

        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" />
    </Grid>
</ContentPage>
```

```

        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" />

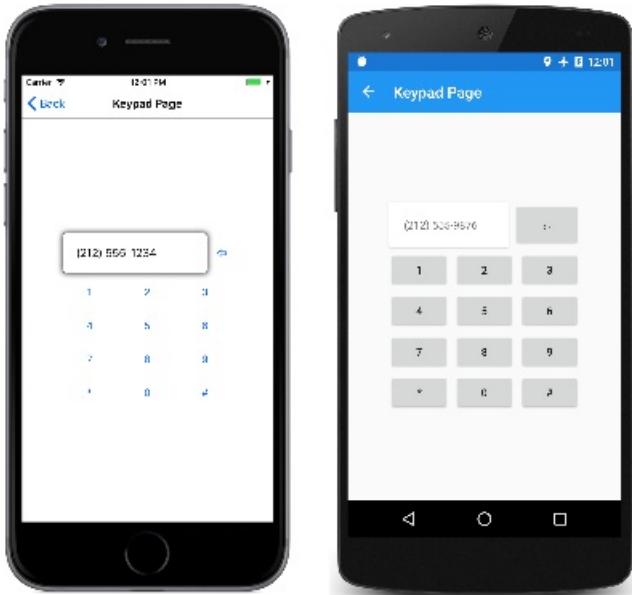
    <Button Text="#"
        Command="{Binding AddCharCommand}"
        CommandParameter="#"
        Grid.Row="4" Grid.Column="2" />

```

</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:

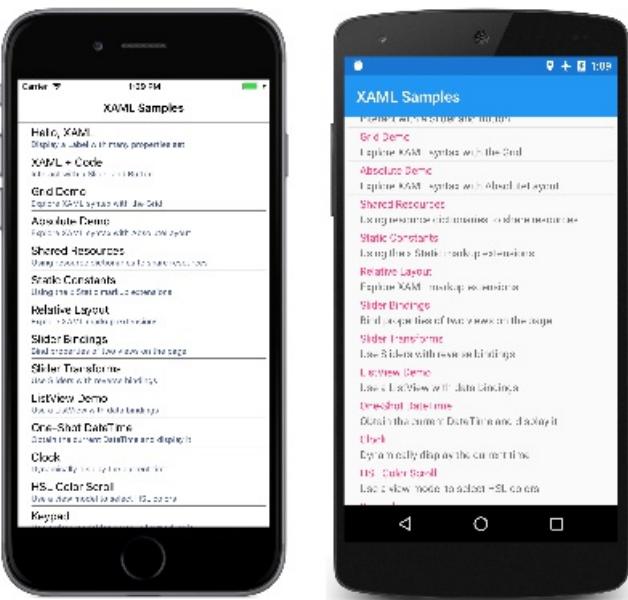
```

<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](#)

Related Videos

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

XAML Controls

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

All of the views that are defined in Xamarin.Forms can be referenced from XAML files.

Views for presentation

VIEW	EXAMPLE
BoxView Displays a rectangle of a particular color. 	<pre><BoxView Color="Accent" WidthRequest="150" HeightRequest="150" HorizontalOptions="Center"></pre>
API / Guide	
Ellipse Displays an ellipse or circle. 	<pre><Ellipse Fill="Red" WidthRequest="150" HeightRequest="50" HorizontalOptions="Center" /></pre>
API / Guide	
Image Displays a bitmap. 	<pre><Image Source="https://aka.ms/campus.jpg" Aspect="AspectFit" HorizontalOptions="Center" /></pre>
API / Guide	

VIEW

Label

Displays one or more lines of text.

Hello, Xamarin.Forms!

[API / Guide](#)

EXAMPLE

```
<Label Text="Hello, Xamarin.Forms!"  
      FontSize="Large"  
      FontAttributes="Italic"  
      HorizontalTextAlignment="Center" />
```

Line

Display a line.

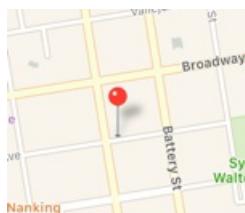


```
<Line X1="40"  
      Y1="0"  
      X2="0"  
      Y2="120"  
      Stroke="Red"  
      HorizontalOptions="Center" />
```

[API / Guide](#)

Map

Displays a map.

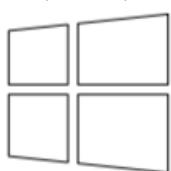


```
<maps:Map ItemsSource="{Binding Locations}" />
```

[API / Guide](#)

Path

Display curves and complex shapes.



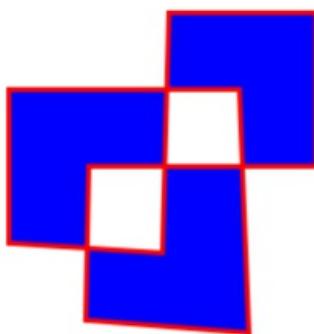
```
<Path Stroke="Black"  
      Aspect="Uniform"  
      HorizontalOptions="Center"  
      HeightRequest="100"  
      WidthRequest="100"  
      Data="M13.9,16.2  
            L32,16.2 32,31.9 13.9,30.1Z  
            M0,16.2  
            L11.9,16.2 11.9,29.9 0,28.6Z  
            M11.9,2  
            L11.9,14.2 0,14.2 0,3.3Z  
            M32,0  
            L32,14.2 13.9,14.2 13.9,1.8Z" />
```

[API / Guide](#)

VIEW

Polygon

Display a polygon.



[API / Guide](#)

EXAMPLE

```
<Polygon Points="0 48, 0 144, 96 150, 100 0, 192  
0, 192 96,  
50 96, 48 192, 150 200 144 48"  
Fill="Blue"  
Stroke="Red"  
StrokeThickness="3"  
HorizontalOptions="Center" />
```

Polyline

Display a series of connected straight lines.



[API / Guide](#)

```
<Polyline Points="0,0 10,30, 15,0 18,60 23,30  
35,30 40,0  
43,60 48,30 100,30"  
Stroke="Red"  
HorizontalOptions="Center" />
```

Rectangle

Display a rectangle or square.



[API / Guide](#)

```
<Rectangle Fill="Red"  
WidthRequest="150"  
HeightRequest="50"  
HorizontalOptions="Center" />
```

WebView

Displays Web pages or HTML content.



[API / Guide](#)

```
<WebView  
Source="https://docs.microsoft.com/xamarin/"  
VerticalOptions="FillAndExpand" />
```

Views that initiate commands

VIEW**Button**

Displays text in a rectangular object.

[API / Guide](#)**EXAMPLE**

```
<Button Text="Click Me!"  
        Font="Large"  
        BorderWidth="1"  
        HorizontalOptions="Center"  
        VerticalOptions="CenterAndExpand"  
        Clicked="OnButtonClicked" />
```

ImageButton

Displays an image in a rectangular object.

[API / Guide](#)**RadioButton**

Allows the selection of one option from a set.

- Apple
- Banana
- Pineapple

[Guide](#)

```
<ImageButton Source="XamarinLogo.png"  
             HorizontalOptions="Center"  
             VerticalOptions="CenterAndExpand"  
             Clicked="OnImageButtonClicked" />
```

RefreshView

Provides pull-to-refresh functionality for scrollable content.

[Guide](#)

```
<RefreshView IsRefreshing="{Binding  
IsRefreshing}"  
            Command="{Binding RefreshCommand}">  
    <!-- Scrollable control goes here -->  
</RefreshView>
```

SearchBar

Accepts user input that it uses to perform a search.

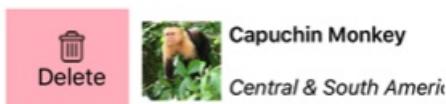
[Guide](#)

```
<SearchBar Placeholder="Enter search term"  
           SearchButtonPressed="OnSearchBarButtonPressed" />
```

VIEW

SwipeView

Provides context menu items that are revealed by a swipe gesture.



[Guide](#)

EXAMPLE

```
<SwipeView>
    <SwipeView.LeftItems>
        <SwipeItems>
            <SwipeItem Text="Delete"
IconImageSource="delete.png"
BackgroundColor="LightPink"
Invoked="OnDeleteInvoked"
/>
        </SwipeItems>
    </SwipeView.LeftItems>
    <!-- Content -->
</SwipeView>
```

Views for setting values

VIEW

CheckBox

Allows the selection of a `boolean` value.



[Guide](#)

EXAMPLE

```
<CheckBox IsChecked="true"
HorizontalOptions="Center"
VerticalOptions="CenterAndExpand" />
```

Slider

Allows the selection of a `double` value from a continuous range.



[API / Guide](#)

```
<Slider Minimum="0"
Maximum="100"
VerticalOptions="CenterAndExpand" />
```

VIEW**Stepper**

Allows the selection of a `double` value from an incremental range.

[API / Guide](#)**EXAMPLE**

```
<Stepper Minimum="0"  
         Maximum="10"  
         Increment="0.1"  
         HorizontalOptions="Center"  
         VerticalOptions="CenterAndExpand" />
```

Switch

Allows the selection of a `boolean` value.

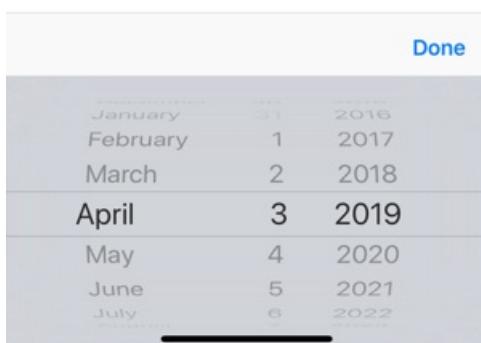
[API / Guide](#)

```
<Switch IsToggled="false"  
        HorizontalOptions="Center"  
        VerticalOptions="CenterAndExpand" />
```

DatePicker

Allows the selection of a date.

Wednesday, April 3, 2019

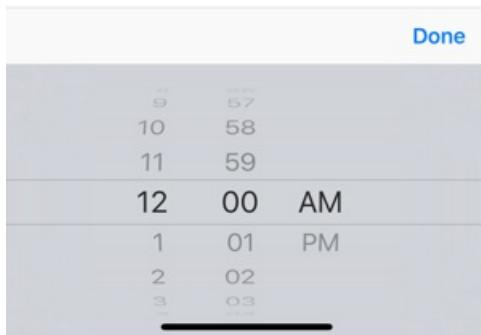
[API / Guide](#)

```
<DatePicker Format="D"  
            VerticalOptions="CenterAndExpand" />
```

TimePicker

Allows the selection of a time.

12:00:00 AM

[API / Guide](#)

```
<TimePicker Format="T"  
            VerticalOptions="CenterAndExpand" />
```

VIEW	EXAMPLE

Views for editing text

VIEW	EXAMPLE
<p>Entry Allows a single line of text to be entered and edited.</p> <div style="border: 1px solid #ccc; padding: 5px; width: fit-content;">Enter email address</div>	<pre><Entry Keyboard="Email" Placeholder="Enter email address" VerticalOptions="CenterAndExpand" /></pre>
<p>API / Guide</p>	

Views to indicate activity

VIEW	EXAMPLE
<p>ActivityIndicator Displays an animation to show that the application is engaged in a lengthy activity, without giving any indication of progress.</p> 	<pre><ActivityIndicator IsRunning="True" VerticalOptions="CenterAndExpand" /></pre>
<p>API / Guide</p>	
<p>ProgressBar Displays an animation to show that the application is progressing through a lengthy activity.</p> 	<pre><ProgressBar Progress=".5" VerticalOptions="CenterAndExpand" /></pre>
<p>API / Guide</p>	

Views that display collections

VIEW**CarouselView**

Displays a scrollable list of data items.



Guide

CollectionView

Displays a scrollable list of selectable data items, using different layout specifications.



Guide

IndicatorView

Displays indicators that represent the number of items in a `CarouselView`.



Guide

EXAMPLE

```
<CarouselView ItemsSource="{Binding Monkeys}">
    ItemTemplate="{StaticResource
    MonkeyTemplate}" />
```

```
<CollectionView ItemsSource="{Binding Monkeys}">
    ItemTemplate="{StaticResource
    MonkeyTemplate}"
    ItemsLayout="VerticalGrid, 2" />
```

```
<IndicatorView x:Name="indicatorView"
    IndicatorColor="LightGray"
    SelectedIndicatorColor="DarkGray"
    />
```

VIEW

ListView

Displays a scrollable list of selectable data items.



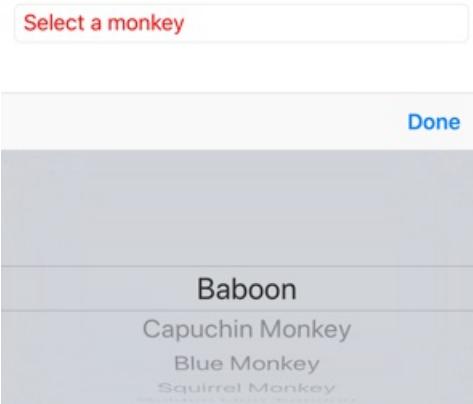
[API / Guide](#)

EXAMPLE

```
<ListView ItemsSource="{Binding Monkeys}">
    ItemTemplate="{StaticResource MonkeyTemplate}" />
```

Picker

Displays a select item from a list of text strings.



[API / Guide](#)

```
<Picker Title="Select a monkey"
        TitleColor="Red">
    <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>
```

TableView

Displays a list of interactive rows.



[API / Guide](#)

```
<TableView Intent="Settings">
    <TableRoot>
        <TableSection Title="Ring">
            <SwitchCell Text="New Voice Mail" />
            <SwitchCell Text="New Mail" On="true" />
        </TableSection>
    </TableRoot>
</TableView>
```

Related links

- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API Documentation](#)

XAML Compilation in Xamarin.Forms

8/4/2022 • 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 in the framework. However, it's enabled in the templates for new projects. It can be explicitly enabled or disabled (`XamlCompilationOptions.Skip`) 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
{
    ...
}
```

While the attribute can be placed anywhere, a good place to put it is in `AssemblyInfo.cs`.

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](#)

XAML Markup Extensions

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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

8/4/2022 • 18 minutes to read • [Edit Online](#)



[Download the sample](#)

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.
- `DataTemplate` – converts a type into a `DataTemplate`.
- `FontImage` – display a font icon in any view that can display an `ImageSource`.
- `AppThemeBinding` – consume a resource based on the current system theme.

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 [Xamarin.Forms control templates](#).
- `RelativeSource` – sets the binding source relative to the position of the binding target, as discussed in the article [Relative Bindings](#).

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=netstandard"
    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=netstandard"
```

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="Ѐ 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">

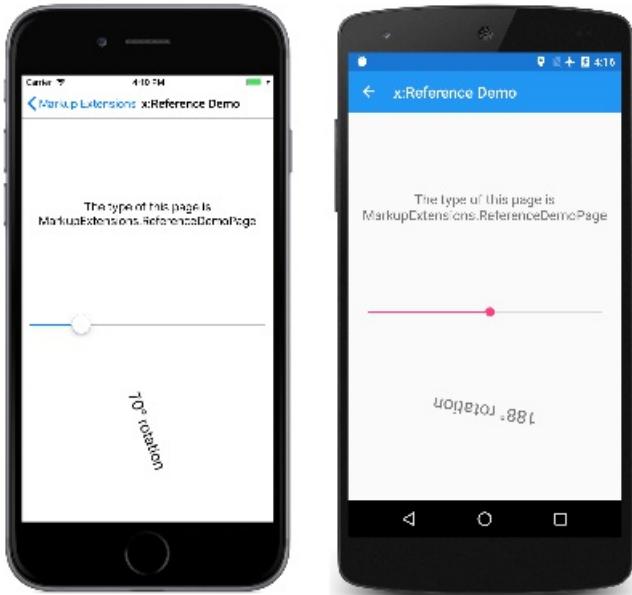
        <Label Text="{Binding Source={x:Reference page},
            StringFormat='The type of this page is {0}'}"
            FontSize="18"
            VerticalOptions="CenterAndExpand"
            HorizontalTextAlignment="Center" />

        <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}" />

                ...
            </TableSection>
        </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();

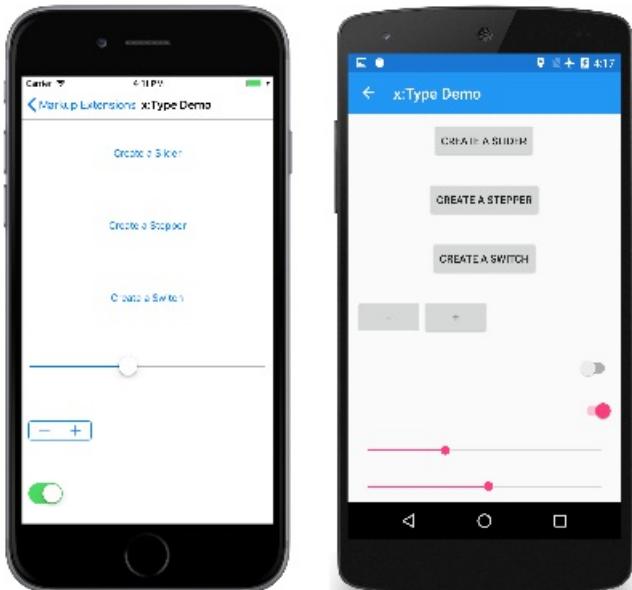
        CreateCommand = new Command<Type>((Type viewType) =>
        {
            View view = (View)Activator.CreateInstance(viewType);
            view.VerticalOptions = LayoutOptions.CenterAndExpand;
            stackLayout.Children.Add(view);
        });
    }

    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 enables 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:

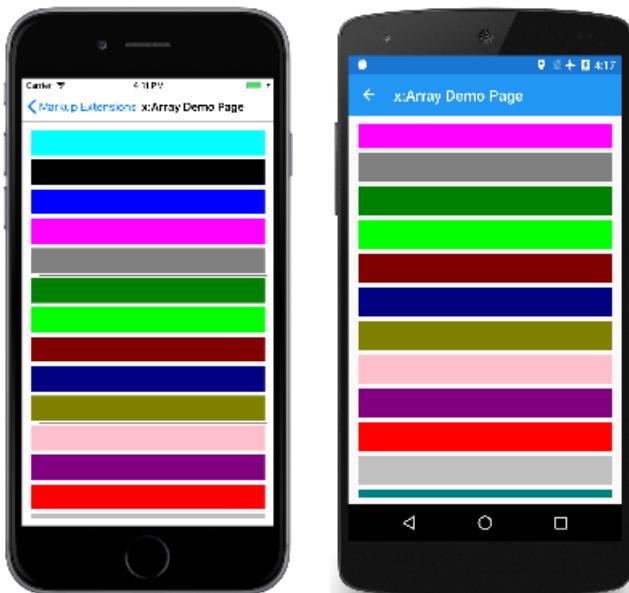
```

<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" />

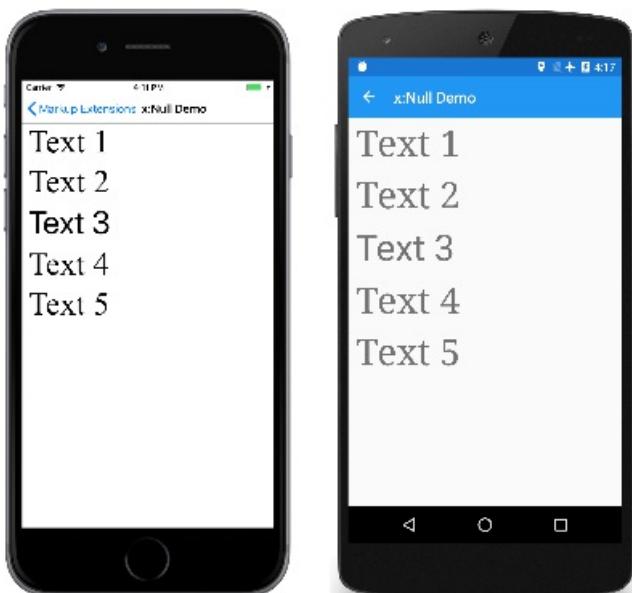
            <Label Text="Text 3"
                FontFamily="{x:Null}" />

            <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 enables 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 can be set to an `IValueConverter` implementation.
- `ConverterParameter` of type `object`, that can be 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. If the `Default` property isn't set, it will default to the `BindableProperty.DefaultValue` property value, provided that the markup extension is targeting a `BindableProperty`.

IMPORTANT

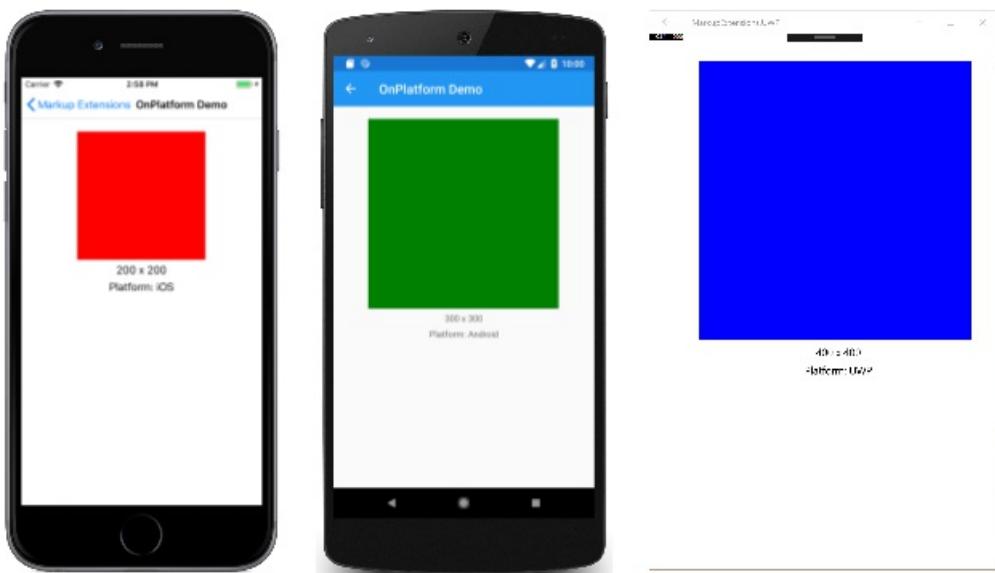
The XAML parser expects that values of the correct type will be provided to properties consuming the `OnPlatform` markup extension. If type conversion is necessary, the `OnPlatform` markup extension will attempt to perform it using the default converters provided by Xamarin.Forms. However, there are 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 extension enables 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 can be set to an `IValueConverter` implementation.
- `ConverterParameter` of type `object`, that can be 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

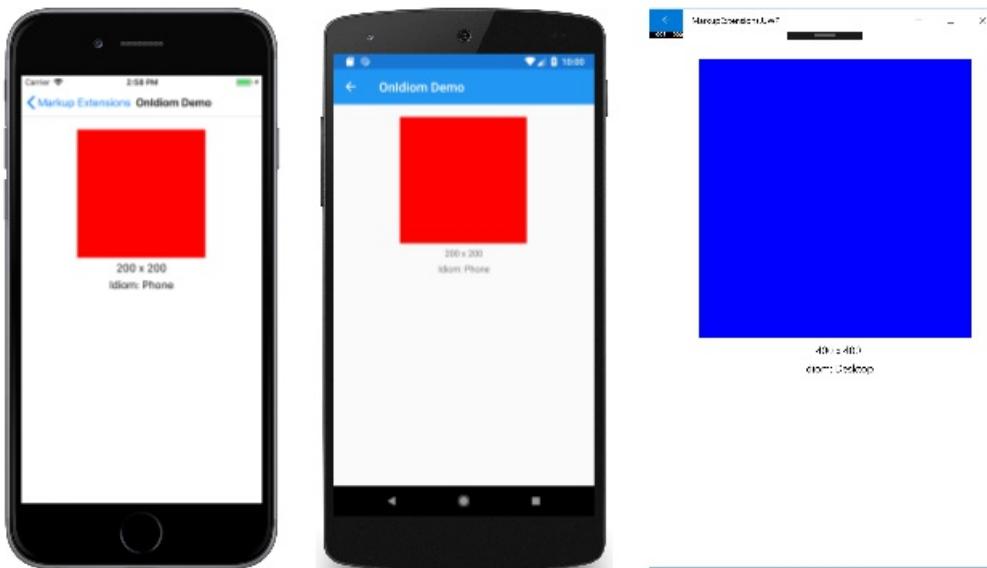
The XAML parser expects that values of the correct type will be provided to properties consuming the `OnIdiom` markup extension. If type conversion is necessary, the `OnIdiom` markup extension will attempt to perform it using the default converters provided by Xamarin.Forms. However, there are some type 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:



DataTemplate markup extension

The `DataTemplate` markup extension enables you to convert a type into a `DataTemplate`. It's supported by the `DataTemplateExtension` class, which defines a `TypeName` property, of type `string`, that is set to the name of the type to be converted into a `DataTemplate`. The `TypeName` property is the content property of `DataTemplateExtension`. Therefore, for XAML markup expressions expressed with curly braces, you can eliminate the `TypeName=` part of the expression.

NOTE

The XAML parser allows the `DataTemplateExtension` class to be abbreviated as `DataTemplate`.

A typical usage of this markup extension is in a Shell application, as shown in the following example:

```
<ShellContent Title="Monkeys"  
             Icon="monkey.png"  
             ContentTemplate="{DataTemplate views:MonkeysPage}" />
```

In this example, `MonkeysPage` is converted from a `ContentPage` to a `DataTemplate`, which is set as the value of the `ShellContent.ContentTemplate` property. This ensures that `MonkeysPage` is only created when navigation to the page occurs, rather than at application startup.

For more information about Shell applications, see [Xamarin.Forms Shell](#).

FontImage markup extension

The `FontImage` markup extension enables you to display a font icon in any view that can display an `ImageSource`. It provides the same functionality as the `FontImageSource` class, but with a more concise representation.

The `FontImage` markup extension is supported by the `FontImageExtension` class, which defines the following properties:

- `FontFamily` of type `string`, the font family to which the font icon belongs.
- `Glyph` of type `string`, the unicode character value of the font icon.
- `Color` of type `Color`, the color to be used when displaying the font icon.
- `Size` of type `double`, the size, in device-independent units, of the rendered font icon. The default value is 30. In addition, this property can be set to a named font size.

NOTE

The XAML parser allows the `FontImageExtension` class to be abbreviated as `FontImage`.

The `Glyph` property is the content property of `FontImageExtension`. Therefore, for XAML markup expressions expressed with curly braces, you can eliminate the `Glyph=` part of the expression provided that it's the first argument.

The [FontImage Demo](#) page shows how to use the `FontImage` markup extension:

```
<Image BackgroundColor="#D1D1D1"
       Source="{FontImage &#xf30c;, FontFamily={OnPlatform iOS=Ionicons, Android=ionicons.ttf#}, Size=44}"
     />
```

In this example, the abbreviated version of the `FontImageExtension` class name is used to display an Xbox icon, from the Ionicons font family, in an `Image`. The expression also uses the `OnPlatform` markup extension to specify different `FontFamily` property values on iOS and Android. In addition, the `Glyph=` part of the expression is eliminated, and the markup extension properties that are set are separated by commas. Note that while the unicode character for the icon is `\uf30c`, it has to be escaped in XAML and so becomes ``.

Here's the program running:



For information about displaying font icons by specifying the font icon data in a `FontImageSource` object, see [Display font icons](#).

AppThemeBinding markup extension

The `AppThemeBinding` markup extension enables you to specify a resource to be consumed, such as an image or color, based on the current system theme.

IMPORTANT

The `AppThemeBinding` markup extension has minimum operating system requirements. For more information, see [Respond to system theme changes in Xamarin.Forms applications](#).

The `AppThemeBinding` markup extension is supported by the `AppThemeBindingExtension` class, which defines the following properties:

- `Default`, of type `object`, that you set to the resource to be used by default.
- `Light`, of type `object`, that you set to the resource to be used when the device is using its light theme.
- `Dark`, of type `object`, that you set to the resource to be used when the device is using its dark theme.
- `Value`, of type `object`, that returns the resource that's currently being used by the markup extension.

NOTE

The XAML parser allows the `AppThemeBindingExtension` class to be abbreviated as `AppBindingTheme`.

The `Default` property is the content property of `AppThemeBindingExtension`. 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.

The [AppThemeBinding Demo](#) page shows how to use the `AppThemeBinding` markup extension:

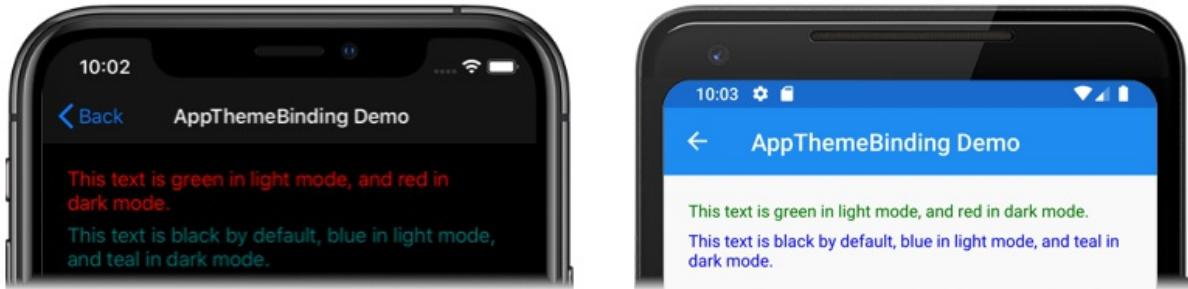
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MarkupExtensions.AppThemeBindingDemoPage"
    Title="AppThemeBinding Demo">
<ContentPage.Resources>

    <Style x:Key="labelStyle"
        TargetType="Label">
        <Setter Property="TextColor"
            Value="{AppThemeBinding Black, Light=Blue, Dark=Teal}" />
    </Style>

</ContentPage.Resources>
<StackLayout Margin="20">
    <Label Text="This text is green in light mode, and red in dark mode."
        TextColor="{AppThemeBinding Light=Green, Dark=Red}" />
    <Label Text="This text is black by default, blue in light mode, and teal in dark mode."
        Style="{StaticResource labelStyle}" />
</StackLayout>
</ContentPage>
```

In this example, the text color of the first `Label` is set to green when the device is using its light theme, and is set to red when the device is using its dark theme. The second `Label` has its `TextColor` property set through a `Style`. This `Style` sets the text color of the `Label` to black by default, to blue when the device is using its light theme, and to teal when the device is using its dark theme.

Here's the program running:



Define 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](#)
- [Xamarin.Forms Shell](#)
- [Respond to system theme changes in Xamarin.Forms applications](#)

Creating XAML Markup Extensions

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

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>

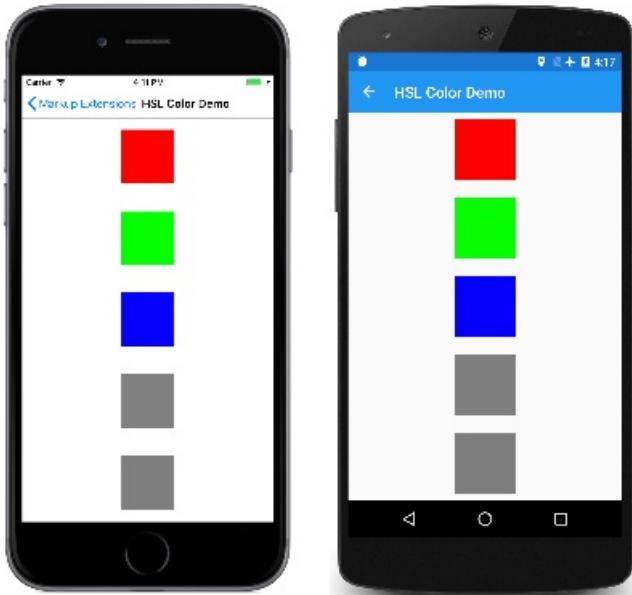
        <BoxView Color="{local:HslColorExtension H=0.67, S=1, L=0.5}" />

        <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 `IXmlLineInfoProvider` 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:

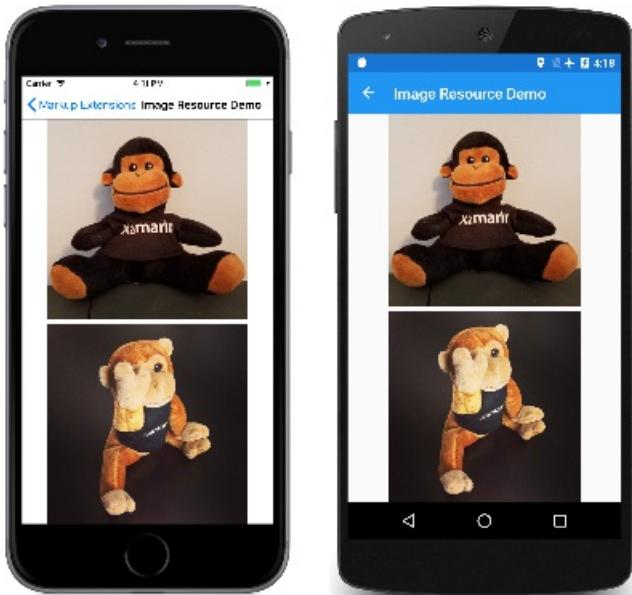
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MarkupExtensions"
    x:Class="MarkupExtensions.ImageResourceDemoPage"
    Title="Image Resource Demo">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition 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 Hot Reload for Xamarin.Forms

8/4/2022 • 6 minutes to read • [Edit Online](#)

XAML Hot Reload plugs into your existing workflow to increase your productivity and save you time. Without XAML Hot Reload, you have to build and deploy your app every time you want to see a XAML change. With Hot Reload, when you save your XAML file the changes are reflected live in your running app. In addition, your navigation state and data will be maintained, enabling you to quickly iterate on your UI without losing your place in the app. Therefore, with XAML Hot Reload, you'll spend less time rebuilding and deploying your apps to validate UI changes.

NOTE

If you're writing a native UWP or WPF app, not using Xamarin.Forms, see [XAML Hot Reload for UWP and WPF](#).

System requirements

IDE/FRAMEWORK	MINIMUM VERSION REQUIRED
Visual Studio 2019	16.9 for changes only mode, 16.4 for full page mode
Visual Studio 2019 for Mac	8.9 for changes only mode, 8.4 for full page mode
Xamarin.Forms	5.0.0.2012 for changes only mode; 4.1 for full page mode

Enable XAML Hot Reload for Xamarin.Forms

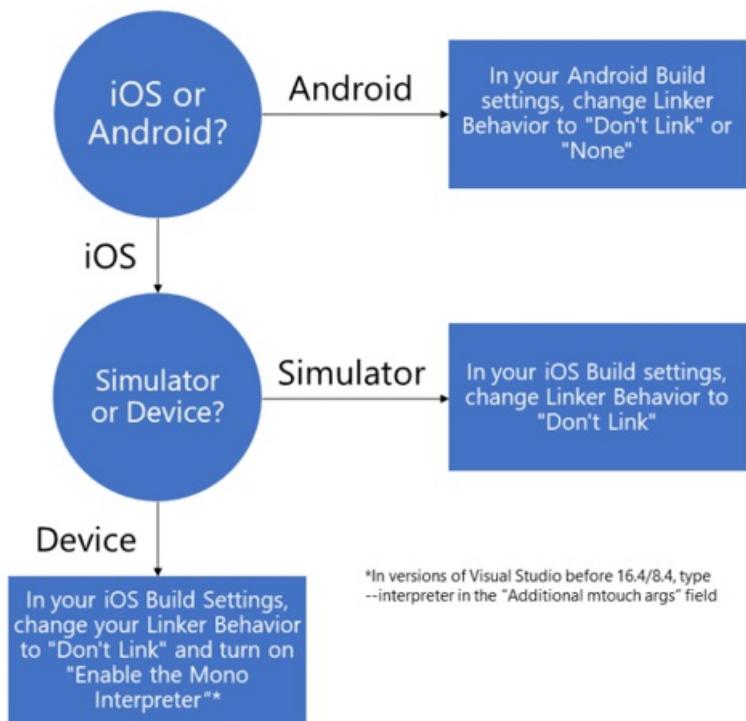
If you are starting from a template, XAML Hot Reload is on by default and the project is configured to work with no additional setup. Debug your Android, iOS, or UWP app on an emulator or physical device and change your XAML to trigger a XAML Hot Reload.

If you're working from an existing Xamarin.Forms solution, no additional installation is required to use XAML Hot Reload, but you might have to double check your configuration to ensure the best experience. First, enable it in your IDE settings:

- On Windows, check the **Enable XAML Hot Reload** checkbox (and the required platforms) at **Tools > Options > Debugging > Hot Reload**.
 - In earlier versions of Visual Studio 2019, the checkbox is at **Tools > Options > Xamarin > Hot Reload**.
- On Mac, check the **Enable Xamarin Hot Reload** checkbox at **Visual Studio > Preferences > Tools for Xamarin > XAML Hot Reload**.
 - In earlier versions of Visual Studio for Mac, the checkbox is at **Visual Studio > Preferences > Projects > Xamarin Hot Reload**.

Then, in your Android and iOS build settings, check that the Linker is set to "Don't Link" or "Link None". To use XAML Hot Reload with a physical iOS device, you also have to check **Enable the Mono interpreter** (Visual Studio 16.4 and above) or add **--interpreter** to your **Additional mtouch args** (Visual Studio 16.3 and below).

You can use the following flowchart to check your existing project's setup for use with XAML Hot Reload:



Hot Reload modes

XAML Hot Reload can work in two different modes - the newer *changes only mode* and the older *full page mode*.

From Visual Studio 16.9 and Visual Studio for Mac 8.9, the default behavior is for changes only mode to be used for all apps that use Xamarin.Forms 5.0 or newer. For older versions of Xamarin.Forms, full page mode is used. However, you can force use of full page mode for all apps in the Hot Reload IDE settings (**Tools > Options > Debugging > Hot Reload** on Windows or **Visual Studio > Preferences > Tools for Xamarin > XAML Hot Reload** on Mac).

Changes only mode parses the XAML to see exactly what changed when you make an edit, and sends just those changes to the running app. This is the same technology used for WPF and UWP Hot Reload. It preserves UI state, since it doesn't recreate the UI for the full page, just updating changed properties on controls affected by edits. Changes only mode also enables use of the [Live Visual Tree](#).

By default, with changes only mode you don't need to save your file to see the changes - updates are applied immediately, as you type. However, you can change this behavior to update only on file save. This can be accomplished by checking the **Apply XAML Hot Reload on document save** checkbox (currently only available on Windows) in the Hot Reload IDE settings. Only updating on document save can sometimes be useful if you make bigger XAML updates and don't wish them to be displayed until they are complete.

Full page mode sends the full XAML file to the running app after you makes edits and save. The running app then reloads the page, recreating its controls - you'll see the UI refresh.

Changes only mode is the future of Hot Reload and we recommend using it whenever possible. It's fast, preserves UI state, and supports [Live Visual Tree](#). Full page mode is still provided for apps that haven't yet been updated to Xamarin.Forms 5.0.

NOTE

You'll need to restart the debug session when switching modes.

XAML errors

Changes only mode: If you make a change the Hot Reload XAML parser sees as invalid, it will show the error underlined in the editor and include it in the errors window. These Hot Reload errors have an error code starting with "XHR" (for XAML Hot Reload). If there are any such errors on the page, Hot Reload won't apply changes, even if made on other parts of the page. Fix all the errors for Hot Reload to start working again for the page.

Full page mode: If you make a change that XAML Hot Reload can't reload, it will show the error underlined in the editor and include it in the errors window. These changes, known as rude edits, include mistyping your XAML or wiring a control to an event handler that doesn't exist. Even with a rude edit, you can continue to reload without restarting the app - make another change elsewhere in the XAML file and hit save. The rude edit won't be reloaded, but your other changes will continue to be applied.

Reload on multiple platforms at once

XAML Hot Reload supports simultaneous debugging in Visual Studio and Visual Studio for Mac. You can deploy an Android and an iOS target at the same time to see your changes reflected on both platforms at once. To debug on multiple platforms, see:

- [Windows How To: Set multiple startup projects](#)
- [Mac Set multiple startup projects](#)

Known limitations

- Xamarin.Forms targets beyond Android, iOS, and UWP (for example, macOS) aren't currently supported.
- Use of [XamlCompilation(XamlCompilationOptions.Skip)], disabling XAML compilation, isn't supported and can cause issues with the Live Visual Tree.
- You can't add, remove, or rename files or NuGet packages during a XAML Hot Reload session. If you add or remove a file or NuGet package, rebuild and redeploy your app to continue using XAML Hot Reload.
- Set your linker to **Don't Link** or **Link None** for the best experience. The **Link SDK only** setting works most of the time, but it may fail in certain cases. Linker settings can be found in your Android and iOS build options.
- Debugging on a physical iPhone requires the interpreter to use XAML Hot Reload. To do this, open the project settings, select the iOS Build tab, and ensure **Enable the Mono interpreter** setting is enabled. You may need to change the **Platform** option at the top of the property page to **iPhone**.
- XAML Hot Reload can't reload C# code, including event handlers, custom controls, page code-behind, and additional classes.

Troubleshooting

- Bring up the XAML Hot Reload output to see status messages, which can help in troubleshooting:
 - **Windows:** bring up Output with View > Output and select **Xamarin Hot Reload** under Show output from: at the top
 - **Mac:** hover over **XAML Hot Reload** in the status bar to show that pad
- If XAML Hot Reload fails to initialize:
 - Update your Xamarin.Forms version.
 - Ensure you are on the latest version of the IDE.
 - Set your Android or iOS Linker settings to **Don't Link** in the project's build settings.
- If nothing happens upon saving your XAML file, ensure that XAML Hot Reload is enabled in the IDE.
- If you're debugging on a physical iPhone and your app becomes unresponsive, check that the interpreter is enabled. To turn it on, check **Enable the Mono interpreter** (Visual Studio 16.4/8.4 and up) or add --interpreter to the Additional mtouch arguments field (Visual Studio 16.3/8.3 and prior) in your iOS

Build settings.

To report a bug, use **Help > Send Feedback > Report a Problem** on Windows, and **Help > Report a Problem** on Mac.

Related links

- [Live Visual Tree](#)
- [Tips and Tricks for XAML Hot Reload](#)
- [XAML Hot Reload for Xamarin.Forms In-Depth: The Xamarin Show](#)

Xamarin.Forms live visual tree

8/4/2022 • 2 minutes to read • [Edit Online](#)

You can receive a real-time view of your running XAML code with the **Live Visual Tree**. It shows a tree view of the UI elements of your running Xamarin.Forms application.

Requirements

- Use Xamarin.Forms 5.0 or newer.
- Have *changes only* Hot Reload enabled (it's enabled by default).

Usage

With the requirements met, debug your app and Live Visual Tree window will show the runtime UI hierarchy of your app.

- **Windows:** By default, it appears on the IDE's left. If you don't see it, use **Debug > Windows > Live Visual Tree** to show it.
- **Mac:** By default, it appears on the IDE's right. If you don't see it, use **View > Debug Windows > Live Visual Tree** to show it.

Use the tree view to inspect the runtime UI hierarchy for your app, expanding/collapsing nodes to focus on particular parts of the UI.

Live visual tree toolbar

The view of XAML elements is simplified by default using the **Just My XAML** feature. Toggle the **Show Just My XAML** button, rightmost on the Live Visual Tree toolbar, to show all UI elements. If you wish you can [disable this setting](#) in options to always show all XAML elements.

NOTE

Visual Studio for Mac doesn't currently support the **Just My XAML** feature.

The structure of the XAML has a lot of elements that you're probably not directly interested in, and if you don't know the code well you might have a hard time navigating the tree to find what you're looking for. Therefore, the **Live Visual Tree** has multiple approaches that let you use the application's UI to help you find the element you want to examine.

Select element in the running application (currently only supported for UWP apps). You can enable this mode when you select the leftmost button on the **Live Visual Tree** toolbar. With this mode on, you can select a UI element in the application, and the **Live Visual Tree** automatically updates to show the node in the tree corresponding to that element, and its properties.

Display layout adorners in the running application (currently only supported for UWP apps). You can enable this mode when you select the button that is immediately to the right of the Enable selection button. When **Display layout adorners** is on, it causes the application window to show horizontal and vertical lines along the bounds of the selected object so you can see what it aligns to, as well as rectangles showing the margins.

Preview Selection. You can enable this mode by selecting the third button from the left on the **Live Visual Tree**

toolbar. This mode shows the XAML where the element was declared, if you have access to the source code of the application.

Related links

- [Write and debug running XAML code with XAML Hot Reload](#)

Xamarin.Forms XAML Toolbox

8/4/2022 • 2 minutes to read • [Edit Online](#)

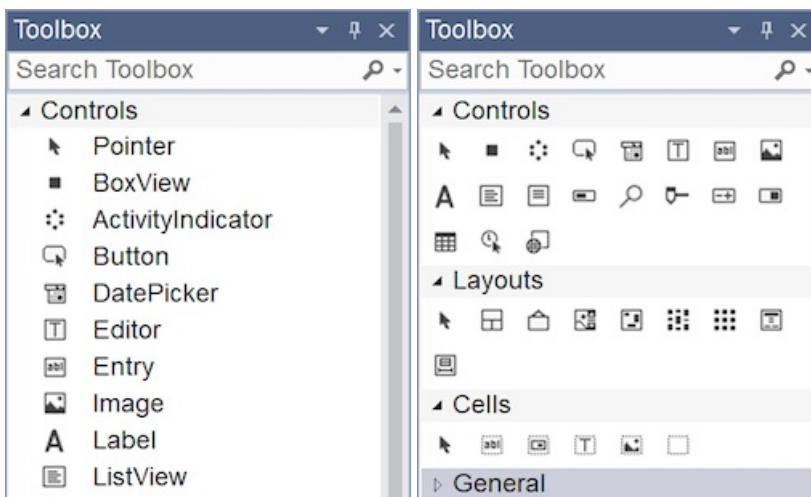
Visual Studio 2017 version 15.8 and Visual Studio for Mac 7.6 now have a Toolbox available while editing Xamarin.Forms XAML files. The toolbox contains all the built-in Xamarin.Forms controls and layouts, which can be dragged into the XAML editor.

- [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 XAML 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

8/4/2022 • 3 minutes to read • [Edit Online](#)

See your Xamarin.Forms layouts rendered as you type

WARNING

The XAML Previewer has been deprecated in Visual Studio 2019 version 16.8 and Visual Studio for Mac version 8.8, and replaced by the XAML Hot Reload feature in Visual Studio 2019 version 16.9 and Visual Studio for Mac version 8.9. Learn more about XAML Hot Reload in [the documentation](#).

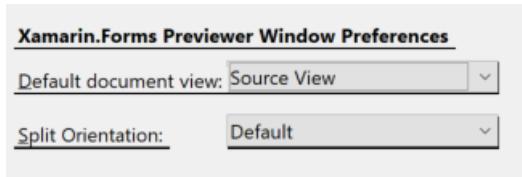
Overview

The XAML Previewer shows you how your Xamarin.Forms XAML page will look on iOS and Android. When you make changes to your XAML, you'll see them previewed immediately alongside your code. The XAML Previewer is available in Visual Studio and Visual Studio for Mac.

Getting started

Visual Studio 2019

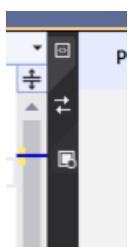
You can open the XAML Previewer by clicking the arrows on the split view pane. If you want to change the default split view behavior, use the **Tools > Options > Xamarin > Xamarin.Forms XAML Previewer** dialog. In this dialog, you can select the default document view and the split orientation.



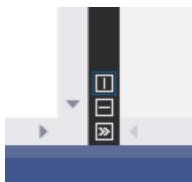
When you open a XAML file, the editor will open either full-sized or next to the previewer, based on the settings selected in the **Tools > Options > Xamarin > Xamarin.Forms XAML Previewer** dialog. However, the split can be changed for each file in the editor window.

XAML preview controls

Choose whether you want to see your code, the XAML Previewer, or both by selecting these buttons on the split view pane. The middle button swaps what side the Previewer and your code are on:



You can change whether the screen is split vertically or horizontally, or collapse one pane altogether:

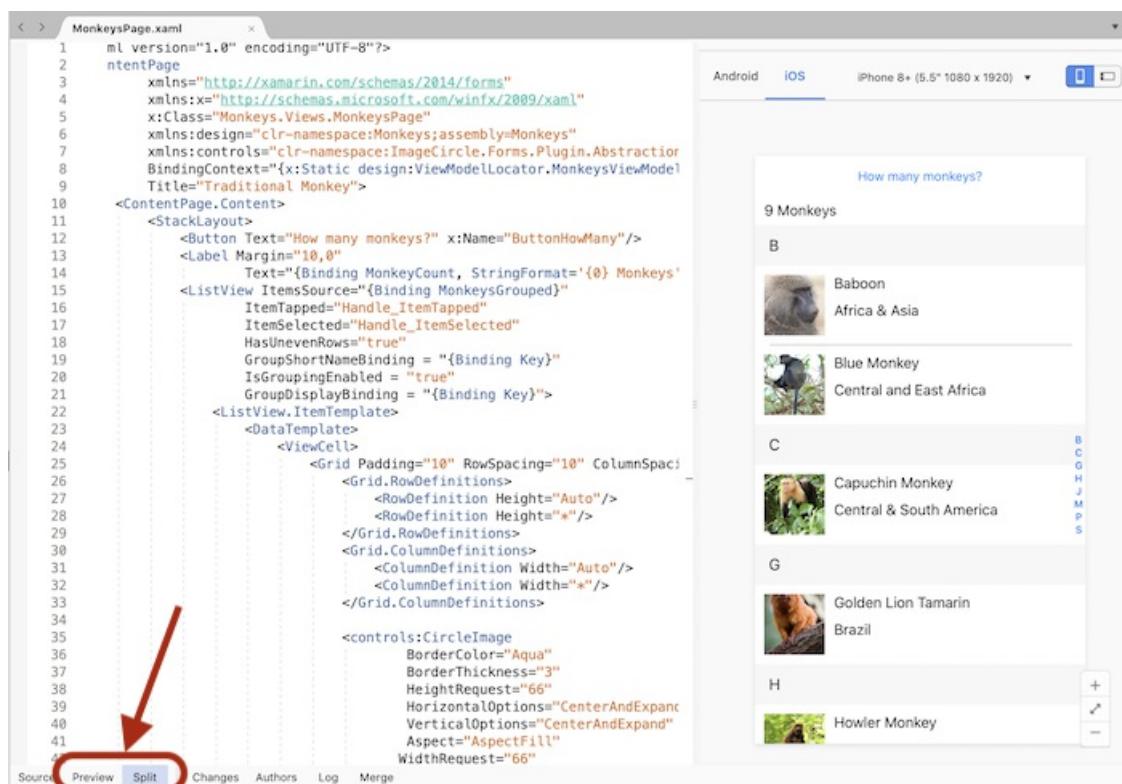


Enable or disable the XAML Previewer

You can turn the XAML Previewer off in the **Tools > Options > Xamarin > Xamarin.Forms XAML Previewer** dialog by selecting **Default XML Editor** as your **Default XAML Editor**. This also turns off the Document Outline, Property Panel, and XAML Toolbox. To turn the XAML Previewer and those tools back on, change your **Default XAML Editor** to **Xamarin.Forms Previewer**.

Visual Studio for Mac

The **Preview** button is displayed on the editor when you open a XAML page. Show or hide the Previewer by pressing the **Preview** or **Split** buttons in the bottom-left of any XAML document window:



NOTE

In older versions of Visual Studio for Mac, the **Preview** button was located in the top-right of the window.

Enable or Disable the XAML Previewer

You can turn the XAML Previewer off in the **Visual Studio > Preferences > Text Editor > XAML** dialog by selecting **Default XML Editor** as your **Default XAML Editor**. This also turns off the Document Outline, Property Panel, and XAML Toolbox. To turn the XAML Previewer and those tools back on, change your **Default XAML Editor** to **Xamarin.Forms Previewer**.

XAML previewer options

The options along the top of the preview pane are:

- **Android** – show the Android version of the screen
- **iOS** – show the iOS version of the screen (*Note: If you're using Visual Studio on Windows, you must be paired to a Mac to use this mode*)

- **Device** - Drop-down list of Android or iOS devices including resolution and screen size
- **Portrait (icon)** – uses portrait orientation for the preview
- **Landscape (icon)** – uses landscape orientation for the preview

Detect design mode

The static `DesignMode.IsEnabled` property tells you if the application is running in the previewer. Using it, you can specify code that will only execute when the application is or isn't running in the previewer:

```
if (DesignMode.IsEnabled)
{
    // Previewer only code
}

if (!DesignMode.IsEnabled)
{
    // Don't run in the Previewer
}
```

This property is useful if you initialize a library in your page constructor that fails to run at design time.

Troubleshooting

Check the issues below and the [Xamarin Forums](#), if the Previewer isn't working.

XAML Previewer isn't showing or shows an error

- It can take some time for the Previewer to start up - you'll see "Initializing Render" until it's ready.
- Try closing and reopening the XAML file.
- Ensure that your `App` class has a parameterless constructor.
- Check your Xamarin.Forms version - it has to be at least Xamarin.Forms 3.6. You can update to the latest Xamarin.Forms version through NuGet.
- Check your JDK installation - previewing Android requires at least [JDK 8](#).
- Try wrapping any initialized classes in the page's C# code behind in `if (!DesignMode.IsEnabled)`.

Custom controls aren't rendering

Try building your project. The previewer shows the control's base class if it fails to render the control, or if the control's creator opted-out of design time rendering. For more information, see [Render Custom Controls in the XAML Previewer](#).

Use Design Time Data with the XAML Previewer

8/4/2022 • 3 minutes to read • [Edit Online](#)

Some layouts are hard to visualize without data. Use these tips to make the most out of previewing your data-heavy pages in the XAML Previewer.

WARNING

The XAML Previewer has been deprecated in Visual Studio 2019 version 16.8 and Visual Studio for Mac version 8.8, and replaced by the XAML Hot Reload feature in Visual Studio 2019 version 16.9 and Visual Studio for Mac version 8.9. Learn more about XAML Hot Reload in [the documentation](#).

NOTE

If you are using Windows Presentation Foundation (WPF) or UWP, see [Use Design Time Data with the XAML Designer for desktop applications](#)

Design time data basics

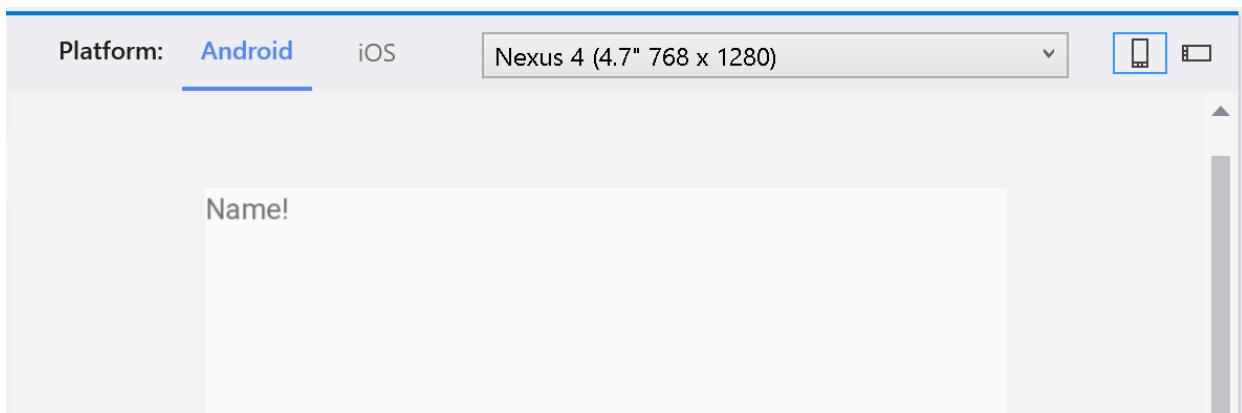
Design time data is fake data you set to make your controls easier to visualize in the XAML Previewer. To get started, add the following lines of code to the header of your XAML page:

```
xmlns:d="http://xamarin.com/schemas/2014/forms/design"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d"
```

After adding the namespaces, you can put `d:` in front of any attribute or control to show it in the XAML Previewer. Elements with `d:` aren't shown at runtime.

For example, you can add text to a label that usually has data bound to it.

```
<Label Text="{Binding Name}" d:Text="Name!" />
```

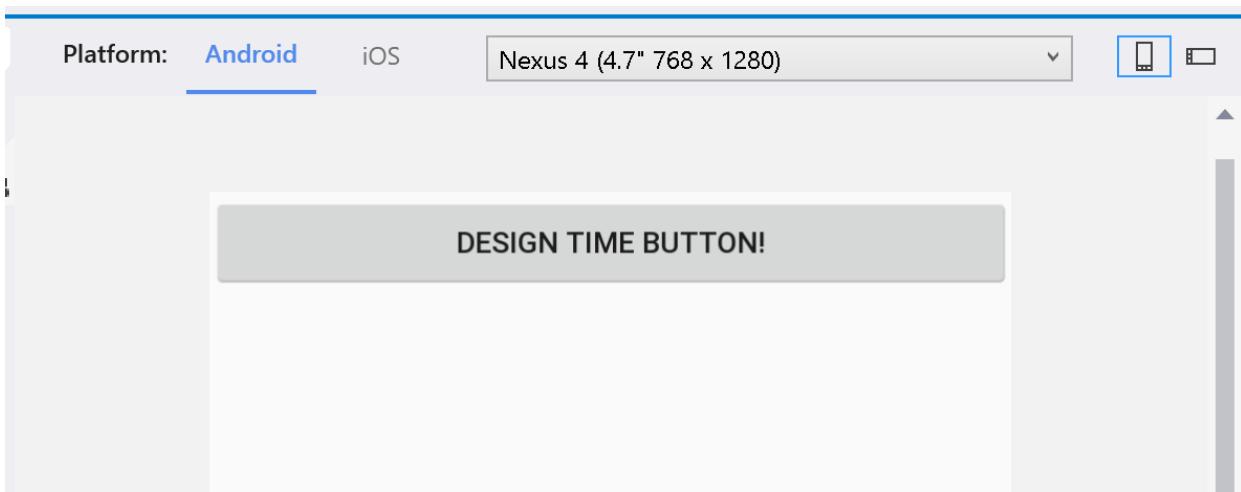


In this example, without `d:Text`, the XAML Previewer would show nothing for the label. Instead, it shows "Name!" where the label will have real data at runtime.

You can use `d:` with any attribute for a Xamarin.Forms control, like colors, font sizes, and spacing. You can even

add it to the control itself:

```
<d:Button Text="Design Time Button" />
```



In this example, the button only appears at design time. Use this method to put a placeholder in for a [custom control not supported by the XAML Previewer](#).

Preview images at design time

You can set a design time Source for images that are bound to the page or loaded in dynamically. In your Android project, add the image you want to show in the XAML Previewer to the **Resources > Drawable** folder. In your iOS project, add the image to the **Resources** folder. You can then show that image in the XAML Previewer at design time:

```
<Image Source={Binding ProfilePicture} d:Source="DesignTimePicture.jpg" />
```

Platform: **Android**

iOS

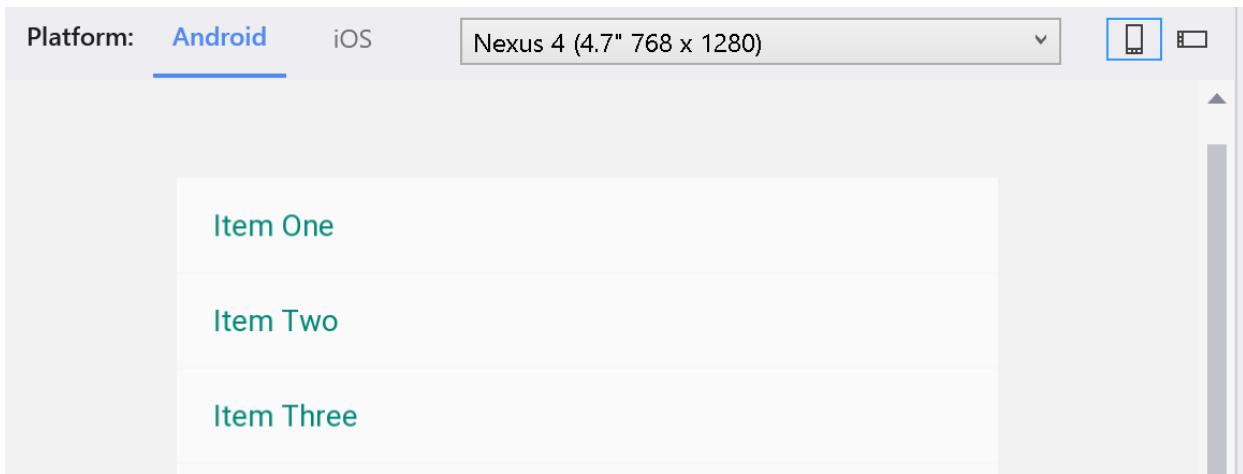
Nexus 4 (4.7" 768 x 1280)



Design time data for ListViews

ListViewes are a popular way to display data in a mobile app. However, they're difficult to visualize without real data. To use design time data with them, you have to create a design time array to use as an ItemsSource. The XAML Previewer displays what is in that array in your ListView at design time.

```
<StackLayout>
    <ListView ItemsSource="{Binding Items}">
        <d:ListView.ItemsSource>
            <x:Array Type="{x:Type x:String}">
                <x:String>Item One</x:String>
                <x:String>Item Two</x:String>
                <x:String>Item Three</x:String>
            </x:Array>
        </d:ListView.ItemsSource>
        <ListView.ItemTemplate>
            <DataTemplate>
                <TextCell Text="{Binding ItemName}"
                    d:Text="{Binding .}" />
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</StackLayout>
```



This example will show a ListView of three TextCells in the XAML Previewer. You can change `x:String` to an existing data model in your project.

You can also create an array of data objects. For example, public properties of a `Monkey` data object can be constructed as design time data:

```
namespace Monkeys.Models
{
    public class Monkey
    {
        public string Name { get; set; }
        public string Location { get; set; }
    }
}
```

To use the class in XAML you will need to import the namespace in the root node:

```
xmlns:models="clr-namespace:Monkeys.Models"
```

```
<StackLayout>
    <ListView ItemsSource="{Binding Items}">
        <d:ListView.ItemsSource>
            <x:Array Type="{x:Type models:Monkey}">
                <models:Monkey Name="Baboon" Location="Africa and Asia"/>
                <models:Monkey Name="Capuchin Monkey" Location="Central and South America"/>
                <models:Monkey Name="Blue Monkey" Location="Central and East Africa"/>
            </x:Array>
        </d:ListView.ItemsSource>
        <ListView.ItemTemplate>
            <DataTemplate x:DataType="models:Monkey">
                <TextCell Text="{Binding Name}"
                    Detail="{Binding Location}" />
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</StackLayout>
```

The benefit here is that you can bind to the actual model that you plan to use.

Alternative: Hardcode a static ViewModel

If you don't want to add design time data to individual controls, you can set up a mock data store to bind to your page. Refer to James Montemagno's [blog post on adding design-time data](#) to see how to bind to a static ViewModel in XAML.

Troubleshooting

Requirements

Design time data requires a minimum version of Xamarin.Forms 3.6.

IntelliSense shows squiggly lines under my design time data

This is a known issue and will be fixed in an upcoming version of Visual Studio. The project will still build without errors.

The XAML Previewer stopped working

Try closing and reopening the XAML file, and cleaning and rebuilding your project.

Render Custom Controls in the XAML Previewer

8/4/2022 • 2 minutes to read • [Edit Online](#)

Custom controls sometimes don't work as expected in the XAML Previewer. Use the guidance in this article to understand the limitations of previewing your custom controls.

WARNING

The XAML Previewer has been deprecated in Visual Studio 2019 version 16.8 and Visual Studio for Mac version 8.8, and replaced by the XAML Hot Reload feature in Visual Studio 2019 version 16.9 and Visual Studio for Mac version 8.9. Learn more about XAML Hot Reload in [the documentation](#).

Basic Preview mode

Even if you haven't built your project, the XAML Previewer will render your pages. Until you build, any control that relies on code-behind will show its base Xamarin.Forms type. When your project is built, the XAML Previewer will try to show custom controls with design time rendering enabled. If the render fails, it will show the base Xamarin.Forms type.

Enable design time rendering for custom controls

If you make your own custom controls, or use controls from a third-party library, the Previewer might display them incorrectly. Custom controls must opt in to design time rendering to appear in the previewer, whether you wrote the control or imported it from a library. With controls you've created, add the `[DesignTimeVisible(true)]` to your control's class to show it in the Previewer:

```
namespace MyProject
{
    [DesignTimeVisible(true)]
    public class MyControl : BaseControl
    {
        // Your control's code here
    }
}
```

Use [James Montemagno's ImageCirclePlugin's base class](#) as an example.

SkiaSharp controls

Currently, SkiaSharp controls are only supported when you're previewing on iOS. They won't render on the Android preview.

Troubleshooting

Check your Xamarin.Forms version

Make sure you have at least Xamarin.Forms 3.6 installed. You can update your Xamarin.Forms version on NuGet.

Even with `[DesignTimeVisible(true)]`, my custom control isn't rendering properly.

Custom controls that rely heavily on code-behind or backend data don't always work in the XAML Previewer.

You can try:

- Moving the control so it doesn't initialize if [design mode is enabled](#)
- Setting up [design time data](#) to show fake data from the backend

The XAML Previewer shows the error "Custom Controls aren't rendering properly"

Try cleaning and rebuilding your project, or closing and reopening the XAML file.

XAML Namespaces in Xamarin.Forms

8/4/2022 • 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 ResourceDictionary . The key's value is used to retrieve the XAML resource, and is typically used as the argument for the StaticResource 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 and x:FactoryMethod attributes, see [Passing Arguments in XAML](#). For more information about the x:TypeArguments attribute, see [Generics in XAML with Xamarin.Forms](#).

NOTE

In addition to the namespace attributes listed above, Xamarin.Forms also includes markup extensions that can be consumed through the x namespace prefix. For more information, see [Consuming XAML Markup Extensions](#).

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>
```

For information about defining a custom namespace schema, see [XAML Custom Namespace Schemas](#).

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](#)
- [Generics in XAML with Xamarin.Forms](#)

XAML Custom Namespace Schemas in Xamarin.Forms

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

Types in a library can be referenced in XAML by declaring a XAML namespace for the library, with the namespace declaration specifying the Common Language Runtime (CLR) namespace name and an assembly name:

```
<ContentPage ...>
    xmlns:controls="clr-namespace:MyCompany.Controls;assembly=MyCompany.Controls">
    ...
</ContentPage>
```

However, specifying a CLR namespace and assembly name in a `xmlns` definition can be awkward and error prone. In addition, multiple XAML namespace declarations may be required if the library contains types in multiple namespaces.

An alternative approach is to define a custom namespace schema, such as

`http://mycompany.com/schemas/controls`, that maps to one or more CLR namespaces. This enables a single XAML namespace declaration to reference all the types in an assembly, even if they are in different namespaces. It also enables a single XAML namespace declaration to reference types in multiple assemblies.

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

Defining a custom namespace schema

The sample application contains a library that exposes some simple controls, such as `CircleButton`:

```
using Xamarin.Forms;

namespace MyCompany.Controls
{
    public class CircleButton : Button
    {
        ...
    }
}
```

All the controls in the library reside in the `MyCompany.Controls` namespace. These controls can be exposed to a calling assembly through a custom namespace schema.

A custom namespace schema is defined with the `XmlNsDefinitionAttribute` class, which specifies the mapping between a XAML namespace and one or more CLR namespaces. The `XmlNsDefinitionAttribute` takes two arguments: the XAML namespace name, and the CLR namespace name. The XAML namespace name is stored in the `XmlNsDefinitionAttribute.XmlNamespace` property, and the CLR namespace name is stored in the `XmlNsDefinitionAttribute.ClrNamespace` property.

NOTE

The `XmlnsDefinitionAttribute` class also has a property named `AssemblyName`, which can be optionally set to the name of the assembly. This is only required when a CLR namespace referenced from a `XmlnsDefinitionAttribute` is in a external assembly.

The `XmlnsDefinitionAttribute` should be defined at the assembly level in the project that contains the CLR namespaces that will be mapped in the custom namespace schema. The following example shows the `AssemblyInfo.cs` file from the sample application:

```
using Xamarin.Forms;
using MyCompany.Controls;

[assembly: Preserve]
[assembly: XmlnsDefinition("http://mycompany.com/schemas/controls", "MyCompany.Controls")]
```

This code creates a custom namespace schema that maps the `http://mycompany.com/schemas/controls` URL to the `MyCompany.Controls` CLR namespace. In addition, the `Preserve` attribute is specified on the assembly, to ensure that the linker preserves all the types in the assembly.

IMPORTANT

The `Preserve` attribute should be applied to classes in the assembly that are mapped through the custom namespace schema, or applied to the entire assembly.

The custom namespace schema can then be used for type resolution in XAML files.

Consuming a custom namespace schema

To consume types from the custom namespace schema, the XAML compiler requires that there's a code reference from the assembly that consumes the types, to the assembly that defines the types. This can be accomplished by adding a class containing an `Init` method to the assembly that defines the types that will be consumed through XAML:

```
namespace MyCompany.Controls
{
    public static class Controls
    {
        public static void Init()
        {
        }
    }
}
```

The `Init` method can then be called from the assembly that consumes types from the custom namespace schema:

```
using Xamarin.Forms;
using MyCompany.Controls;

namespace CustomNamespaceSchemaDemo
{
    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            Controls.Init();
            InitializeComponent();
        }
    }
}
```

WARNING

Failure to include such a code reference will result in the XAML compiler being unable to locate the assembly containing the custom namespace schema types.

To consume the `CircleButton` control, a XAML namespace is declared, with the namespace declaration specifying the custom namespace schema URL:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:controls="http://mycompany.com/schemas/controls"
             x:Class="CustomNamespaceSchemaDemo.MainPage">
    <StackLayout Margin="20,35,20,20">
        ...
        <controls:CircleButton Text="+" 
            BackgroundColor="Fuchsia"
            BorderColor="Black"
            CircleDiameter="100" />
        <controls:CircleButton Text="-" 
            BackgroundColor="Teal"
            BorderColor="Silver"
            CircleDiameter="70" />
        ...
    </StackLayout>
</ContentPage>
```

`CircleButton` instances can then be added to the `ContentPage` by declaring them with the `controls` namespace prefix.

To find the custom namespace schema types, Xamarin.Forms will search referenced assemblies for `XmlNsDefinitionAttribute` instances. If the `xmlns` attribute for an element in a XAML file matches the `XmlNamespace` property value in a `XmlNsDefinitionAttribute`, Xamarin.Forms will attempt to use the `XmlNsDefinitionAttribute.ClrNamespace` property value for resolution of the type. If type resolution fails, Xamarin.Forms will continue to attempt type resolution based on any additional matching `XmlNsDefinitionAttribute` instances.

The result is that two `CircleButton` instances are displayed:



Related links

- [Custom Namespace Schemas \(sample\)](#)
- [XAML Namespace Recommended Prefixes](#)
- [XAML Namespaces in Xamarin.Forms](#)

XAML Namespace Recommended Prefixes in Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)

The `XmlnsPrefixAttribute` class can be used by control authors to specify a recommended prefix to associate with a XAML namespace, for XAML usage. The prefix is useful when supporting object tree serialization to XAML, or when interacting with a design environment that has XAML editing features. For example:

- XAML text editors could use the `XmlnsPrefixAttribute` as a hint for an initial XAML namespace `xmlns` mapping.
- XAML design environments could use the `XmlnsPrefixAttribute` to add mappings to the XAML when dragging objects out of a toolbox and onto a visual design surface.

Recommended namespace prefixes should be defined at the assembly level with the `XmlnsPrefixAttribute` constructor, which takes two arguments: a string that specifies the identifier of a XAML namespace, and a string that specifies a recommended prefix:

```
[assembly: XmlnsPrefix("http://xamarin.com/schemas/2014/forms", "xf")]
```

Prefixes should use short strings, because the prefix is typically applied to all serialized elements that come from the XAML namespace. Therefore, the prefix string length can have a noticeable effect on the size of the serialized XAML output.

NOTE

More than one `XmlnsPrefixAttribute` can be applied to an assembly. For example, if you have an assembly that defines types for more than one XAML namespace, you could define different prefix values for each XAML namespace.

Related links

- [XAML Custom Namespace Schemas](#)
- [XAML Namespaces in Xamarin.Forms](#)

Xamarin.Forms Bindable Properties

8/4/2022 • 8 minutes to read • [Edit Online](#)



[Download the sample](#)

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` field 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`.

Create 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.

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.

Create 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.

IMPORTANT

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.

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 `EventNameProperty`, of type `string`. The property is owned by the `EventToCommandBehavior` class, and has a default value of `null`.

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 [Detect 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 [Create a default value with a Func](#).

Create 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); }
}
```

Consume 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.

Detect 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.

IMPORTANT

The `BindableObject` type has a `CoerceValue` method that can be called to force a reevaluation of the value of its `BindableProperty` argument, by invoking its coerce value callback.

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, propertyChanged: ForceCoerceValue);
...

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;
}

static void ForceCoerceValue(BindableObject bindable, object oldValue, object newValue)
{
    bindable.CoerceValue(AngleProperty);
}

```

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. In addition, when the `MaximumAngle` property changes the coerce value callback is invoked on the `Angle` property by calling the `CoerceValue` method.

Create 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.

Related links

- [XAML Namespaces](#)
- [Event To Command Behavior \(sample\)](#)
- [Validation Callback \(sample\)](#)
- [Coerce Value Callback \(sample\)](#)
- [BindableProperty API](#)
- [BindableObject API](#)

Attached Properties

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

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](#).

Create 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.

Create 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.

IMPORTANT

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.

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 `HasShadowProperty`, of type `bool`. The property is owned by the `ShadowEffect` class, and has a default value of `false`.

For more information about creating bindable properties, including parameters that can be specified during creation, see [Create a bindable property](#).

Create 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);
}
```

Consume 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);
```

Consume 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](#).

Related links

- [Bindable Properties](#)
- [XAML Namespaces](#)
- [Shadow Effect \(sample\)](#)
- [BindableProperty API](#)
- [BindableObject API](#)

Xamarin.Forms resource dictionaries

8/4/2022 • 8 minutes to read • [Edit Online](#)



[Download the sample](#)

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 be referenced and applied to elements by using the `StaticResource` or `DynamicResource` markup extension. In C#, resources can also be defined in a `ResourceDictionary` and then referenced 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 be stored as fields or properties, and accessed directly without having to first retrieve them from a dictionary.

Create resources in XAML

Every `VisualElement` derived object has a `Resources` property, which is a `ResourceDictionary` that can contain resources. Similarly, an `Application` derived object has a `Resources` property, which is a `ResourceDictionary` that can contain resources.

A Xamarin.Forms application contains only 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` containing resources. 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.
- 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.

With the exception of implicit styles, each resource in resource dictionary must have a unique string key that's defined with the `x:Key` attribute.

The following XAML shows resources defined in an application level `ResourceDictionary` in the `App.xaml` file:

```

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

        <Thickness x:Key="PageMargin">20</Thickness>

        <!-- Colors -->
        <Color x:Key="AppBackgroundColor">AliceBlue</Color>
        <Color x:Key="NavigationBarColor">#1976D2</Color>
        <Color x:Key="NavigationBarTextColor">White</Color>
        <Color x:Key="NormalTextColor">Black</Color>

        <!-- Implicit styles -->
        <Style TargetType="{x:Type NavigationPage}">
            <Setter Property="BarBackgroundColor"
                   Value="{StaticResource NavigationBarColor}" />
            <Setter Property="BarTextColor"
                   Value="{StaticResource NavigationBarTextColor}" />
        </Style>

        <Style TargetType="{x:Type ContentPage}"
               ApplyToDerivedTypes="True">
            <Setter Property="BackgroundColor"
                   Value="{StaticResource AppBackgroundColor}" />
        </Style>
    </Application.Resources>
</Application>

```

In this example, the resource dictionary defines a `Thickness` resource, multiple `Color` resources, and two implicit `Style` resources. For more information about the `App` class, see [Xamarin.Forms App Class](#).

NOTE

It's also valid to place all resources between explicit `ResourceDictionary` tags. However, since Xamarin.Forms 3.0 the `ResourceDictionary` tags are not required. Instead, the `ResourceDictionary` object is created automatically, and you can insert the resources directly between the `Resources` property-element tags.

Consume resources in XAML

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 reference a resource from the `ResourceDictionary` with the `StaticResource` or `DynamicResource` markup extension.

The `staticResource` markup extension is similar to the `DynamicResource` markup extension in that both use a dictionary key to reference a value from a resource dictionary. However, while the `StaticResource` markup extension performs a single dictionary lookup, the `DynamicResource` markup extension 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 resource changes to be made in an application. For more information about markup extensions, see [XAML Markup Extensions](#).

The following XAML example shows how to consume resources, and also defines additional resources in a `StackLayout`:

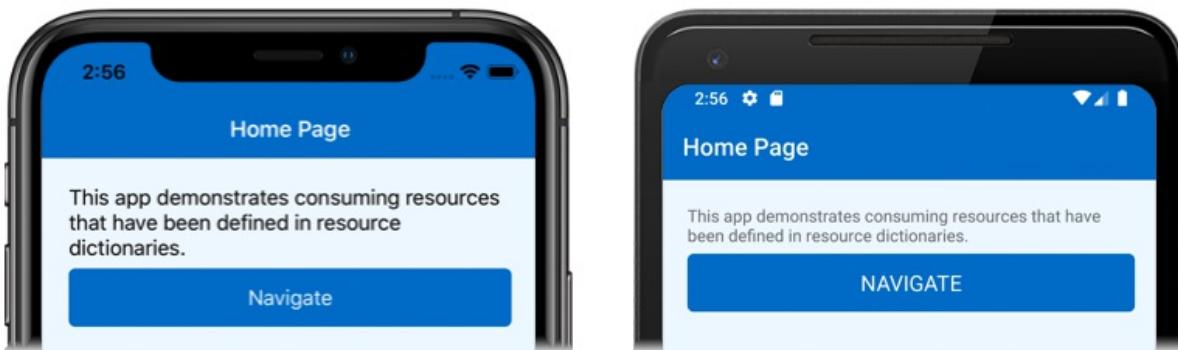
```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ResourceDictionaryDemo.HomePage"
    Title="Home Page">
    <StackLayout Margin="{StaticResource PageMargin}">
        <StackLayout.Resources>
            <!-- Implicit style -->
            <Style TargetType="Button">
                <Setter Property="FontSize" Value="Medium" />
                <Setter Property="BackgroundColor" Value="#1976D2" />
                <Setter Property="TextColor" Value="White" />
                <Setter Property="CornerRadius" Value="5" />
            </Style>
        </StackLayout.Resources>

        <Label Text="This app demonstrates consuming resources that have been defined in resource
dictionaries." />
        <Button Text="Navigate"
            Clicked="OnNavigateButtonClicked" />
    </StackLayout>
</ContentPage>

```

In this example, the `ContentPage` object consumes the implicit style defined in the application level resource dictionary. The `StackLayout` object consumes the `PageMargin` resource defined in the application level resource dictionary, while the `Button` object consumes the implicit style defined in the `StackLayout` resource dictionary. This results in the appearance shown in the following screenshots:



IMPORTANT

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](#).

Resource lookup behavior

The following lookup process occurs when a resource is referenced with the `StaticResource` or `DynamicResource` markup extension:

- The requested key is checked for in the resource dictionary, if it exists, for the element that sets the property. If the requested key is found, its value is returned and the lookup process terminates.
- If a match isn't found, the lookup process searches the visual tree upwards, checking the resource dictionary of each parent element. If the requested key is found, its value is returned and the lookup process terminates. Otherwise the process continues upwards until the root element is reached.
- If a match isn't found at the root element, the application level resource dictionary is examined.
- If a match still isn't found, a `XamlParseException` is thrown.

Therefore, when the XAML parser encounters a `StaticResource` or `DynamicResource` markup extension, 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 thrown.

Override resources

When resources share keys, resources defined lower in the visual tree will take precedence over those defined higher up. For example, setting an `AppBackgroundColor` resource to `AliceBlue` at the application level will be overridden by a page level `AppBackgroundColor` resource set to `Teal`. Similarly, a page level `AppBackgroundColor` resource will be overridden by a control level `AppBackgroundColor` resource.

Stand-alone resource dictionaries

A class derived from `ResourceDictionary` can also be in a stand-alone XAML file. The XAML 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). Delete the code-behind file, and in the XAML file change the name of the base class from `ContentView` or `ContentPage` to `ResourceDictionary`. In addition, remove the `x:Class` attribute from the root tag of the file.

The following XAML example shows a `ResourceDictionary` named `MyResourceDictionary.xaml`:

```
<ResourceDictionary xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml">
    <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}"
                    HorizontalTextAlignment="End" />
            </Grid>
        </ViewCell>
    </DataTemplate>
</ResourceDictionary>
```

In this example, the `ResourceDictionary` contains a single resource, which is an object of type `DataTemplate`. `MyResourceDictionary.xaml` can be consumed by merging it into another resource dictionary.

By default, the linker will remove stand-alone XAML files from release builds when the linker behavior is set to link all assemblies. To ensure that stand-alone XAML files remain in a release build:

1. Add a custom `Preserve` attribute to the assembly containing the stand-alone XAML files. For more information, see [Preserving code](#).
2. Set the `Preserve` attribute at the assembly level:

```
[assembly:Preserve(AllMembers = true)]
```

For more information about linking, see [Linking Xamarin.iOS apps](#) and [Linking on Android](#).

Merged resource dictionaries

Merged resource dictionaries combine one or more `ResourceDictionary` objects into another `ResourceDictionary`.

Merge local resource dictionaries

A local `ResourceDictionary` file can be merged into another `ResourceDictionary` by creating a `ResourceDictionary` object whose `Source` property is set to the filename of the XAML file with the resources:

```
<ContentPage ...>
<ContentPage.Resources>
    <!-- Add more resources here -->
    <ResourceDictionary Source="MyResourceDictionary.xaml" />
    <!-- Add more resources here -->
</ContentPage.Resources>
...
</ContentPage>
```

This syntax does not instantiate the `MyResourceDictionary` class. Instead, it references the XAML file. For that reason, when setting the `Source` property, a code-behind file isn't required, and the `x:Class` attribute can be removed from the root tag of the `MyResourceDictionary.xaml` file.

IMPORTANT

The `Source` property can only be set from XAML.

Merge resource dictionaries from other assemblies

A `ResourceDictionary` can also be merged into another `ResourceDictionary` by adding it into the `MergedDictionaries` property of the `ResourceDictionary`. This technique allows resource dictionaries to be merged, regardless of the assembly in which they reside. Merging resource dictionaries from external assemblies requires the `ResourceDictionary` to have a build action set to `EmbeddedResource`, to have a code-behind file, and to define the `x:Class` attribute in the root tag of the file.

WARNING

The `ResourceDictionary` class also defines a `MergedWith` property. However, this property has been deprecated and should no longer be used.

The following code example shows two resource dictionaries being added to the `MergedDictionaries` collection of a page level `ResourceDictionary`:

```

<ContentPage ...>
    xmlns:local="clr-namespace:ResourceDictionaryDemo"
    xmlns:theme="clr-namespace:MyThemes;assembly=MyThemes">
    <ContentPage.Resources>
        <ResourceDictionary>
            <!-- Add more resources here -->
            <ResourceDictionary.MergedDictionaries>
                <!-- Add more resource dictionaries here -->
                <local:MyResourceDictionary />
                <theme:LightTheme />
                <!-- Add more resource dictionaries here -->
            </ResourceDictionary.MergedDictionaries>
            <!-- Add more resources here -->
        </ResourceDictionary>
    </ContentPage.Resources>
    ...
</ContentPage>

```

In this example, a resource dictionary from the same assembly, and a resource dictionary from an external assembly, are merged into the page level resource dictionary. In addition, you can also add other `ResourceDictionary` objects within the `MergedDictionaries` property-element tags, and other resources outside of those tags.

IMPORTANT

There can be only one `MergedDictionaries` property-element tag in a `ResourceDictionary`, but you can put as many `ResourceDictionary` objects in there as required.

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 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.

Related links

- [Resource Dictionaries \(sample\)](#)
- [XAML Markup Extensions](#)
- [Xamarin.Forms Styles](#)
- [Linking Xamarin.iOS apps](#)
- [Linking on Android](#)
- [ResourceDictionary API](#)

Related video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Passing Arguments in XAML

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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:Array`
- `x:Boolean`
- `x[Byte]`
- `x:Char`
- `x:DateTime`
- `x:Decimal`
- `x:Double`
- `x:Int16`
- `x:Int32`
- `x:Int64`
- `x:Object`
- `x:Single`
- `x:String`
- `x:TimeSpan`

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`.

For more information about generic type arguments, see [Generics in Xamarin.Forms XAML](#).

Related Links

- [Passing Constructor Arguments \(sample\)](#)
- [Calling Factory Methods \(sample\)](#)
- [XAML Namespaces](#)

- [Generics in Xamarin.Forms XAML](#)

Generics in Xamarin.Forms XAML

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms XAML provides support for consuming generic CLR types by specifying the generic constraints as type arguments. This support is provided by the `x:TypeArguments` directive, which passes the constraining type arguments of a generic to the constructor of the generic type.

IMPORTANT

Defining generic classes in Xamarin.Forms XAML, with the `x:TypeArguments` directive, is unsupported.

Type arguments are specified as a string, and are typically prefixed, such as `sys:String` and `sys:Int32`. Prefixing is required because the typical types of CLR generic constraints come from libraries that are not mapped to the default Xamarin.Forms namespace. However, the XAML 2009 built-in types such as `x:String` and `x:Int32`, can also be specified as type arguments, where `x` is the XAML language namespace for XAML 2009. For more information about the XAML 2009 built-in types, see [XAML 2009 Language Primitives](#).

Multiple type arguments can be specified by using a comma delimiter. In addition, if a generic constraint uses generic types, the nested constraint type arguments should be contained in parentheses.

NOTE

The `x:Type` markup extension supplies a CLR type reference for a generic type, and has a similar function to the `typeof` operator in C#. For more information, see [x:Type markup extension](#).

Single primitive type argument

A single primitive type argument can be specified as a prefixed string argument using the `x:TypeArguments` directive:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:scg="clr-namespace:System.Collections.Generic;assembly=netstandard"
    ...>
<CollectionView>
    <CollectionView.ItemsSource>
        <scg>List x:TypeArguments="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>
        </scg>List>
    </CollectionView.ItemsSource>
</CollectionView>
</ContentPage>
```

In this example, `System.Collections.Generic` is defined as the `scg` XAML namespace. The

`CollectionView.ItemsSource` property is set to a `List<T>` that's instantiated with a `string` type argument, using the XAML 2009 built-in `x:String` type. The `List<string>` collection is initialized with multiple `string` items.

Alternatively, but equivalently, the `List<T>` collection can be instantiated with the CLR `String` type:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:scg="clr-namespace:System.Collections.Generic;assembly=netstandard"
    xmlns:sys="clr-namespace:System;assembly=netstandard"
    ...
<CollectionView>
    <CollectionView.ItemsSource>
        <scg:List x:TypeArguments="sys:String">
            <sys:String>Baboon</sys:String>
            <sys:String>Capuchin Monkey</sys:String>
            <sys:String>Blue Monkey</sys:String>
            <sys:String>Squirrel Monkey</sys:String>
            <sys:String>Golden Lion Tamarin</sys:String>
            <sys:String>Howler Monkey</sys:String>
            <sys:String>Japanese Macaque</sys:String>
        </scg:List>
    </CollectionView.ItemsSource>
</CollectionView>
</ContentPage>
```

Single object type argument

A single object type argument can be specified as a prefixed string argument using the `x:TypeArguments` directive:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:models="clr-namespace:GenericsDemo.Models"
    xmlns:scg="clr-namespace:System.Collections.Generic;assembly=netstandard"
    ...
    <CollectionView>
        <CollectionView.ItemsSource>
            <scg>List x:TypeArguments="models:Monkey">
                <models:Monkey Name="Baboon"
                    Location="Africa and Asia"

ImageUrl="https://upload.wikimedia.org/wikipedia/commons/thumb/f/fc/Papio_anubis_%28Serengeti%2C_2009%29.jpg
/200px-Papio_anubis_%28Serengeti%2C_2009%29.jpg" />
                <models:Monkey Name="Capuchin Monkey"
                    Location="Central and South America"

ImageUrl="https://upload.wikimedia.org/wikipedia/commons/thumb/4/40/Capuchin_Costa_Rica.jpg/200px-
Capuchin_Costa_Rica.jpg" />
                <models:Monkey Name="Blue Monkey"
                    Location="Central and East Africa"

ImageUrl="https://upload.wikimedia.org/wikipedia/commons/thumb/8/83/BlueMonkey.jpg/220px-BlueMonkey.jpg" />
            </scg>List>
        </CollectionView.ItemsSource>
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <Grid Padding="10">
                    <Grid.RowDefinitions>
                        <RowDefinition Height="Auto" />
                        <RowDefinition Height="Auto" />
                    </Grid.RowDefinitions>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto" />
                        <ColumnDefinition Width="Auto" />
                    </Grid.ColumnDefinitions>
                    <Image Grid.RowSpan="2"
                        Source="{Binding ImageUrl}"
                        Aspect="AspectFill"
                        HeightRequest="60"
                        WidthRequest="60" />
                    <Label Grid.Column="1"
                        Text="{Binding Name}"
                        FontAttributes="Bold" />
                    <Label Grid.Row="1"
                        Grid.Column="1"
                        Text="{Binding Location}"
                        FontAttributes="Italic"
                        VerticalOptions="End" />
                </Grid>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</ContentPage>

```

In this example, `GenericsDemo.Models` is defined as the `models` XAML namespace, and `System.Collections.Generic` is defined as the `scg` XAML namespace. The `CollectionView.ItemsSource` property is set to a `List<T>` that's instantiated with a `Monkey` type argument. The `List<Monkey>` collection is initialized with multiple `Monkey` items, and a `DataTemplate` that defines the appearance of each `Monkey` object is set as the `ItemTemplate` of the `CollectionView`.

Multiple type arguments

Multiple type arguments can be specified as prefixed string arguments, delimited by a comma, using the

`x>TypeArguments` directive. When a generic constraint uses generic types, the nested constraint type arguments are contained in parentheses:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:models="clr-namespace:GenericsDemo.Models"
    xmlns:scg="clr-namespace:System.Collections.Generic;assembly=netstandard"
    ...
    <CollectionView>
        <CollectionView.ItemsSource>
            <scg:List x:TypeArguments="scg:KeyValuePair(x:String,models:Monkey)">
                <scg:KeyValuePair x:TypeArguments="x:String,models:Monkey">
                    <x:Arguments>
                        <x:String>Baboon</x:String>
                        <models:Monkey Location="Africa and Asia" />
                    </x:Arguments>
                </scg:KeyValuePair>
                <scg:KeyValuePair x:TypeArguments="x:String,models:Monkey">
                    <x:Arguments>
                        <x:String>Capuchin Monkey</x:String>
                        <models:Monkey Location="Central and South America" />
                    </x:Arguments>
                </scg:KeyValuePair>
                <scg:KeyValuePair x:TypeArguments="x:String,models:Monkey">
                    <x:Arguments>
                        <x:String>Blue Monkey</x:String>
                        <models:Monkey Location="Central and East Africa" />
                    </x:Arguments>
                </scg:KeyValuePair>
            </scg:List>
        </CollectionView.ItemsSource>
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <Grid Padding="10">
                    <Grid.RowDefinitions>
                        <RowDefinition Height="Auto" />
                        <RowDefinition Height="Auto" />
                    </Grid.RowDefinitions>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto" />
                        <ColumnDefinition Width="Auto" />
                    </Grid.ColumnDefinitions>
                    <Image Grid.RowSpan="2"
                        Source="{Binding Value.ImageUrl}"
                        Aspect="AspectFill"
                        HeightRequest="60"
                        WidthRequest="60" />
                    <Label Grid.Column="1"
                        Text="{Binding Key}"
                        FontAttributes="Bold" />
                    <Label Grid.Row="1"
                        Grid.Column="1"
                        Text="{Binding Value.Location}"
                        FontAttributes="Italic"
                        VerticalOptions="End" />
                </Grid>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</ContentPage>

```

In this example, `GenericsDemo.Models` is defined as the `models` XAML namespace, and `System.Collections.Generic` is defined as the `scg` XAML namespace. The `CollectionView.ItemsSource` property is set to a `List<T>` that's instantiated with a `KeyValuePair< TKey , TValue >` constraint, with the inner constraint type arguments `string` and `Monkey`. The `List<KeyValuePair<string,Monkey>>` collection is initialized with multiple `KeyValuePair` items, using the non-default `KeyValuePair` constructor, and a `DataTemplate` that defines the appearance of each `Monkey` object is set as the `ItemTemplate` of the `CollectionView`. For information on passing arguments to a non-default constructor, see [Passing constructor arguments](#).

Related links

- [Generics in XAML \(sample\)](#)
- [XAML 2009 Language Primitives](#)
- [x:Type markup extension](#)
- [Passing constructor arguments](#)

XAML Field Modifiers in Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)

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

- `private` – specifies that the generated field for the XAML element is accessible only within the body of the class in which it is declared.
- `public` – specifies that the generated field for the XAML element has no access restrictions.
- `protected` – specifies that the generated field for the XAML element is accessible within its class and by derived class instances.
- `internal` – specifies that the generated field for the XAML element is accessible only within types in the same assembly.
- `notpublic` – specifies that the generated field for the XAML element is accessible only within types in the same assembly.

By default, if the value of the attribute isn't set, the generated field for the element will be `private`.

NOTE

The value of the attribute can use any casing, as it will be converted to lowercase by Xamarin.Forms.

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="internal" />
<Label x:Name="publicLabel" x:FieldModifier="public" />
```

IMPORTANT

The `x:FieldModifier` attribute cannot be used to specify the access level of a XAML class.

Loading XAML at Runtime in Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Xamarin.Forms.Xaml` namespace includes two `LoadFromXaml` extension methods that can be used to load, and parse XAML at runtime.

Background

When a Xamarin.Forms XAML class is constructed, the `LoadFromXaml` method is indirectly called. This occurs because the code-behind file for a XAML class calls the `InitializeComponent` method from its constructor:

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

When Visual Studio builds a project containing a XAML file, it parses the XAML file to generate a C# code file (for example, `MainPage.xaml.g.cs`) that contains the definition of the `InitializeComponent` method:

```
private void InitializeComponent()
{
    global::Xamarin.Forms.Xaml.Extensions.LoadFromXaml(this, typeof(MainPage));
    ...
}
```

The `InitializeComponent` method calls the `LoadFromXaml` method to extract the XAML file (or its compiled binary) from the .NET Standard library. After extraction, it initializes all of 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.

Loading XAML at runtime

The `LoadFromXaml` methods are `public`, and therefore can be called from Xamarin.Forms applications to load, and parse XAML at runtime. This permits scenarios such as an application downloading XAML from a web service, creating the required view from the XAML, and displaying it in the application.

WARNING

Loading XAML at runtime has a significant performance cost, and generally should be avoided.

The following code example shows a simple usage:

```
using Xamarin.Forms.Xaml;
...
string navigationButtonXAML = "<Button Text=\"Navigate\" />";
Button navigationButton = new Button().LoadFromXaml(navigationButtonXAML);
...
_stackLayout.Children.Add(navigationButton);
```

In this example, a `Button` instance is created, with its `Text` property value being set from the XAML defined in the `string`. The `Button` is then added to a `StackLayout` that has been defined in the XAML for the page.

NOTE

The `LoadFromXaml` extension methods allow a generic type argument to be specified. However, it's rarely necessary to specify the type argument, as it will be inferred from the type of the instance its operating on.

The `LoadFromXaml` method can be used to inflate any XAML, with the following example inflating a `ContentPage` and then navigating to it:

```
using Xamarin.Forms.Xaml;
...
// See the sample for the full XAML string
string pageXAML = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\r\n<ContentPage
xmlns=\"http://xamarin.com/schemas/2014/forms\"\nxmlns:x=\"http://schemas.microsoft.com/winfx/2009/xaml\"\nx
:Class=\"LoadRuntimeXAML.CatalogItemsPage\"\nTitle=\"Catalog Items\">\n</ContentPage>";
ContentPage page = new ContentPage().LoadFromXaml(pageXAML);
await Navigation.PushAsync(page);
```

Accessing elements

Loading XAML at runtime with the `LoadFromXaml` method does not permit strongly-typed access to the XAML elements that have specified runtime object names (using `x:Name`). However, these XAML elements can be retrieved using the `FindByName` method, and then accessed as required:

```
// See the sample for the full XAML string
string pageXAML = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\r\n<ContentPage
xmlns=\"http://xamarin.com/schemas/2014/forms\"\nxmlns:x=\"http://schemas.microsoft.com/winfx/2009/xaml\"\nx
:Class=\"LoadRuntimeXAML.CatalogItemsPage\"\nTitle=\"Catalog Items\">\n<StackLayout>\n<Label
x:Name=\"monkeyName\"\n/>\n</StackLayout>\n</ContentPage>";
ContentPage page = new ContentPage().LoadFromXaml(pageXAML);

Label monkeyLabel = page.FindByName<Label>("monkeyName");
monkeyLabel.Text = "Seated Monkey";
...
```

In this example, the XAML for a `ContentPage` is inflated. This XAML includes a `Label` named `monkeyName`, which is retrieved using the `FindByName` method, before its `Text` property is set.

Related links

- [LoadRuntimeXAML \(sample\)](#)

Xamarin.Forms Accessibility

8/4/2022 • 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, to improve usability and accessibility. For more information, see [Keyboard Accessibility](#).

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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Automation Properties in Xamarin.Forms

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

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);
```

IMPORTANT

The `AutomationProperties.LabeledByProperty` is not yet supported on iOS.

NOTE

Note that the `SetValue` method can also be used to set the `AutomationProperties.IsInAccessibleTree` attached property – `entry.SetValue(AutomationProperties.LabeledByProperty, nameLabel);`

Accessibility intricacies

The following sections describe the intricacies of setting accessibility values on certain controls.

NavigationView

On Android, to set the text that screen readers will read for the back arrow in the action bar in a `NavigationView`, set the `AutomationProperties.Name` and `AutomationProperties.HelpText` properties on a `Page`. However, note that this will not have an effect on OS back buttons.

FlyoutPage

On iOS and the Universal Windows Platform (UWP), to set the text that screen readers will read for the toggle button on a `FlyoutPage`, either set the `AutomationProperties.Name`, and `AutomationProperties.HelpText` properties on the `FlyoutPage`, or on the `IconImageSource` property of the `Flyout` page.

On Android, to set the text that screen readers will read for the toggle button on a `FlyoutPage`, add string resources to the Android project:

```
<resources>
    <string name="app_name">Xamarin Forms Control Gallery</string>
    <string name="btnMDPAutomationID_open">Open Side Menu message</string>
    <string name="btnMDPAutomationID_close">Close Side Menu message</string>
</resources>
```

Then set the `AutomationId` property of the `IconImageSource` property of the `Flyout` page to the appropriate string:

```
var flyout = new ContentPage { ... };
flyout.IconImageSource.AutomationId = "btnMDPAutomationID";
```

ToolbarItem

On iOS, Android, and UWP, screen readers will read the `Text` property value of `ToolbarItem` instances, provided that `AutomationProperties.Name` or `AutomationProperties.HelpText` values aren't defined.

On iOS and UWP the `AutomationProperties.Name` property value will replace the `Text` property value that is read by the screen reader.

On Android, the `AutomationProperties.Name` and/or `AutomationProperties.HelpText` property values will completely replace the `Text` property value that is both visible and read by the screen reader. Note that this is a limitation of APIs less than 26.

Related Links

- [Attached Properties](#)
- [Accessibility \(sample\)](#)

Keyboard Accessibility in Xamarin.Forms

8/4/2022 • 3 minutes to read • [Edit Online](#)

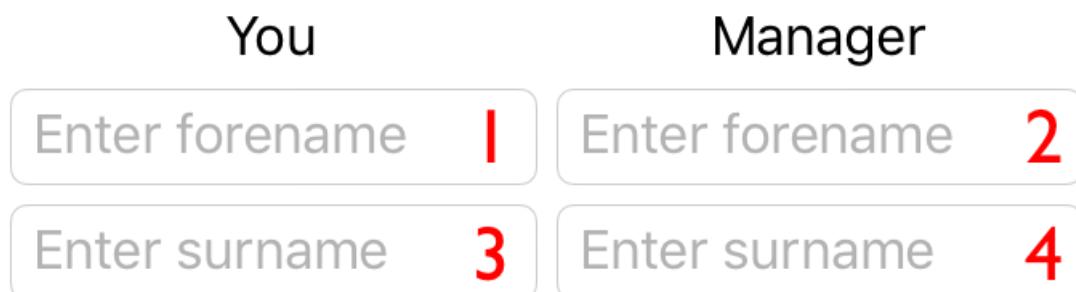
 [Download the sample](#)

Users who use screen readers, or have mobility issues, can have difficulty using applications that don't provide appropriate keyboard access. Xamarin.Forms applications can have an expected tab order specified to improve their usability and accessibility. Specifying a tab order for controls enables keyboard navigation, prepares application pages to receive input in a particular order, and permits screen readers to read focusable elements to the user.

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 read by screen readers, 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.

WARNING

On the Universal Windows Platform, the `TabIndex` property of each control must be set to `int.MaxValue` for the tab order to be identical to the control declaration order.

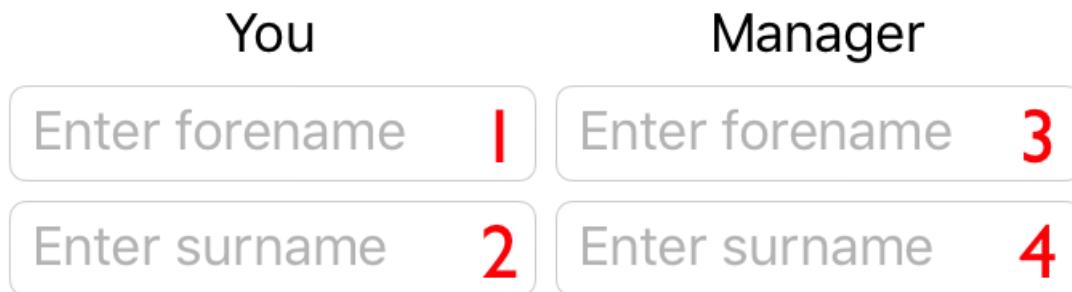
The following XAML example shows the `TabIndex` property set on input controls to enable column-first tab navigation:

```

<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.

IMPORTANT

Screen readers on iOS and Android will respect the [TabIndex](#) of a [VisualElement](#) when reading the accessible elements on the screen.

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.IsTabStop](#) 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 `IsTabStop` 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

8/4/2022 • 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](#) (see 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

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 from `App.xaml.cs` in the UWP project.

For more information, see [Setup Windows Projects](#), which includes steps to add a UWP project to an existing `Xamarin.Forms` solution that doesn't target UWP.

Related video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Forms App Lifecycle

8/4/2022 • 2 minutes to read • [Edit Online](#)

The `Application` base class provides 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 respond to lifecycle changes:

- `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

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");
}
```

IMPORTANT

On Android, the `OnStart` method will be called on rotation as well as when the application first starts, if the main activity lacks `ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation` in the `[Activity()]` attribute.

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 appear 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` - raised when a page is modally pushed.
- `ModalPushed` - raised after a page has been pushed modally.
- `ModalPopping` - raised when a page is modally popped.
- `ModalPopped` - raised after a page has been popped modally.

NOTE

The `ModalPopping` event arguments, of type `ModalPoppingEventArgs`, contain a `Cancel` property. When `Cancel` is set to `true` the modal pop is cancelled.

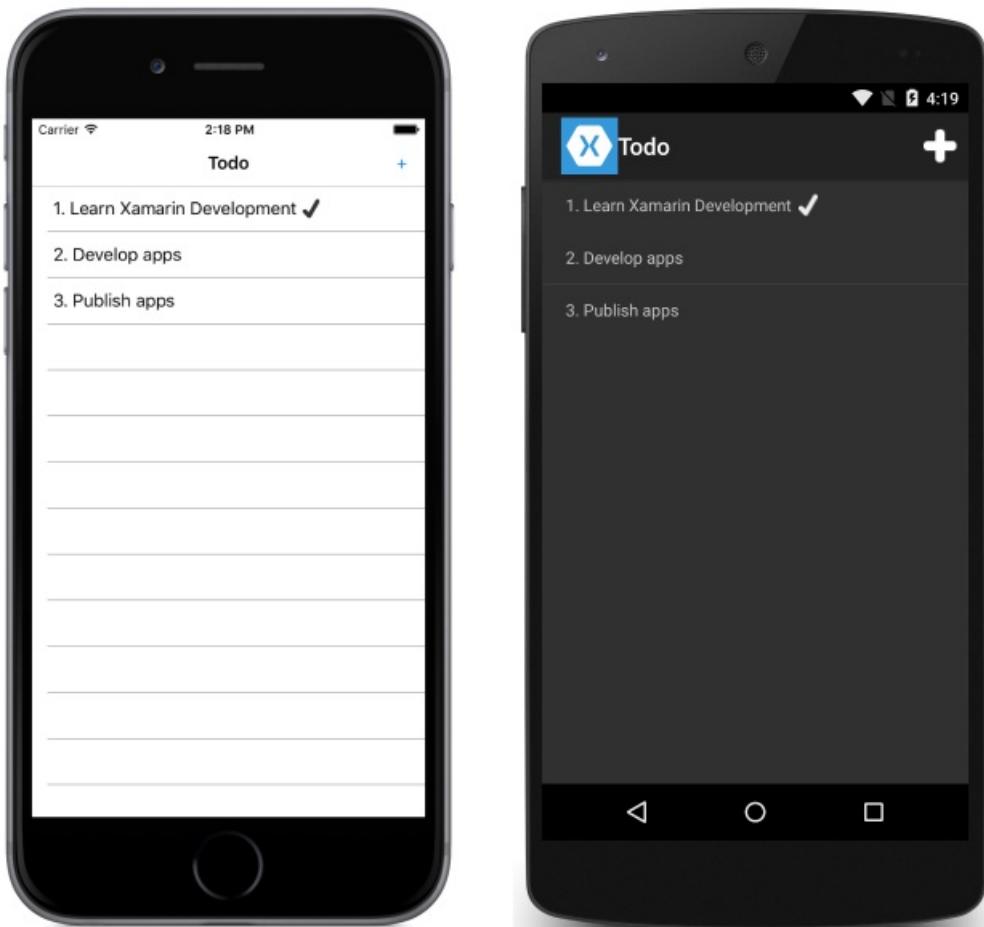
Application Indexing and Deep Linking

8/4/2022 • 8 minutes to read • [Edit Online](#)

 [Download the sample](#)

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 [Xamarin.Forms Local Databases](#).

NOTE

Xamarin.Forms application indexing and deep linking functionality is only available on the iOS and Android platforms, and requires a minimum of 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, ensure that your iOS platform project sets the `Entitlements.plist` file as the custom entitlements file for signing the bundle.

To use iOS Universal Links:

1. Add an Associated Domains entitlement to your app, with the `applinks` key, including all the domains your app will support.
2. Add an Apple App Site Association file to your website.
3. Add the `applinks` key to the Apple App Site Association file.

For more information, see [Allowing Apps and Websites to Link to Your Content](#) on developer.apple.com.

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, add a declaration to use the `Xamarin.Forms.Platform.Android.AppLinks` namespace.
3. In the `MainActivity.cs` file, add a declaration to use the `Firebase` namespace.
4. In a web browser, create a new project via the [Firebase Console](#).
5. In the Firebase Console, add Firebase to your Android app, and enter the required data.
6. Download the resulting `google-services.json` file.
7. Add the `google-services.json` file to the root directory of the Android project, and set its **Build Action** to `GoogleServicesJson`.
8. In the `MainActivity.OnCreate` override, add the following line of code underneath `Forms.Init(this, bundle)`:

```
FirebaseApp.InitializeApp(this);
AndroidAppLinks.Init(this);
```

When `google-services.json` is added to the project (and the `GoogleServicesJson*` build action is set), the build process extracts the client ID and API key and then adds these credentials to the generated manifest file.

NOTE

In this article, the terms application links and deep links are often used interchangeably. However, on Android these terms have separate meanings. On Android, a deep link is an intent filter that allows users to directly enter a specific activity in the app. Clicking on a deep link might open a disambiguation dialog, which allows the user to select one of multiple apps that can handle the URL. An Android app link is a deep link based on your website URL, which has been verified to belong to your website. Clicking on an app link opens your app if it's installed, without opening a disambiguation dialog.

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($"http://{App.AppName}/{pageType}?id={item.ID}", UriKind.RelativeOrAbsolute),
        IsLinkActive = true,
        Thumbnail = ImageSource.FromFile("monkey.png")
    };

    pageLink.KeyValues.Add("contentType", "TodoItemPage");
    pageLink.KeyValues.Add("appName", App.AppName);
    pageLink.KeyValues.Add("companyName", "Xamarin");

    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=2
```

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 2.

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:

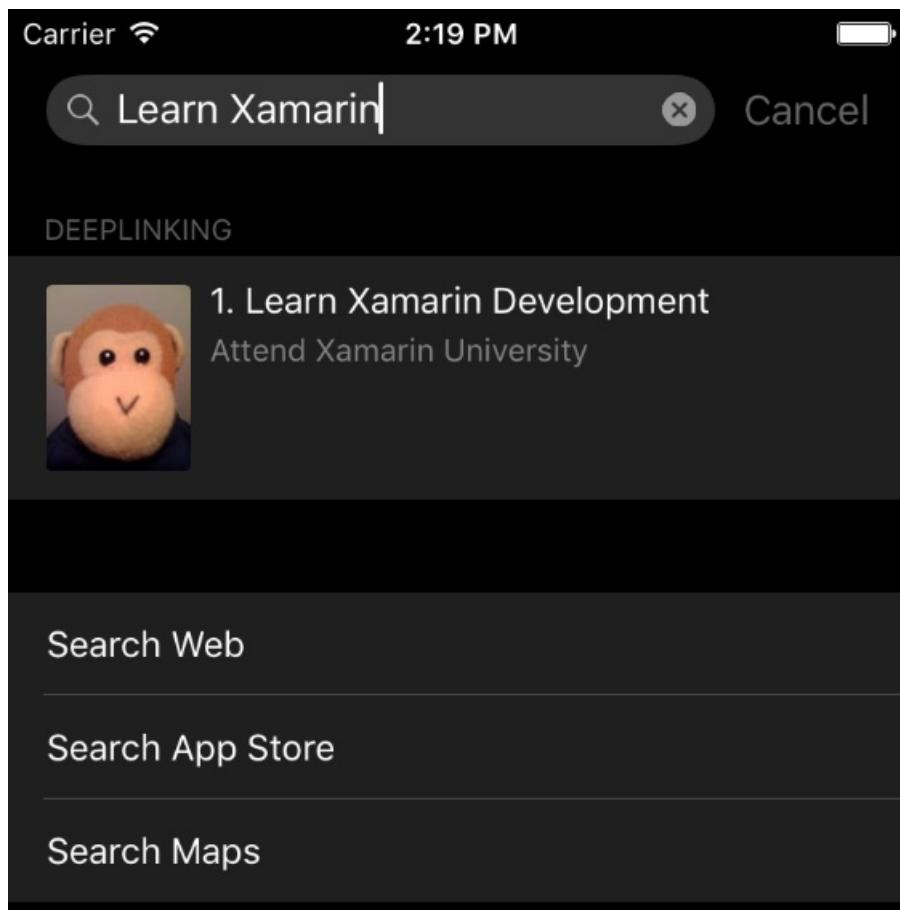
```
Application.Current.AppLinks.RegisterLink (appLink);
```

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, StringComparison.OrdinalIgnoreCase))
            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 = await App.Database.GetItemAsync(int.Parse(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.IsLinkActive` 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.IsLinkActive` property to `true` in the `Page.OnAppearing` override:

```
protected override void OnAppearing()
{
    appLink = GetAppLink(BindingContext as TodoItem);
    if (appLink != null)
    {
        appLink.IsLinkActive = true;
    }
}
```

Similarly, when the page represented by a deep link is navigated away from, the `AppLinkEntry.IsLinkActive` 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](#).

Related Links

- [Deep Linking \(sample\)](#)
- [iOS Search APIs](#)
- [App-Linking in Android 6.0](#)
- [AppLinkEntry](#)
- [IAppLinkEntry](#)

- [IApplinks](#)

Xamarin.Forms Behaviors

8/4/2022 • 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 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.

Introduction to Behaviors

8/4/2022 • 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](#).
- **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

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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](#).

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:

```

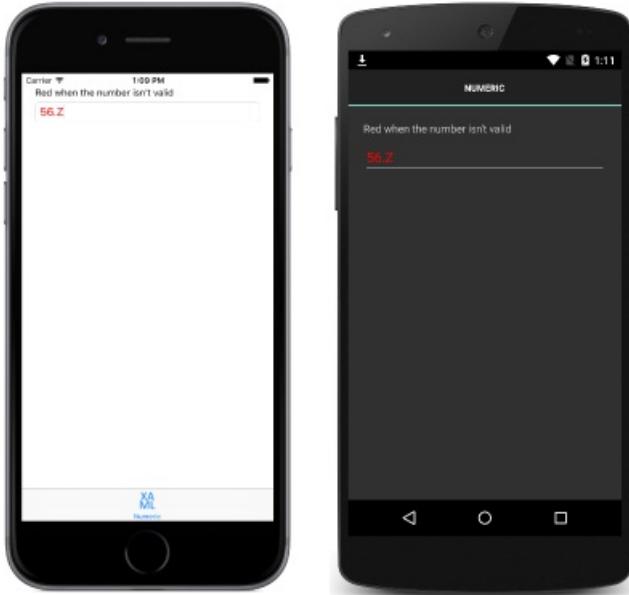
<ContentPage ... xmlns:local="clr-namespace:WorkingWithBehaviors;assembly=WorkingWithBehaviors" ...>
    ...
    <Entry Placeholder="Enter a System.Double" local:NumericValidationBehavior.AttachBehavior="true" />
    ...
</ContentPage>

```

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 \(sample\)](#)

Create Xamarin.Forms behaviors

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

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>

```

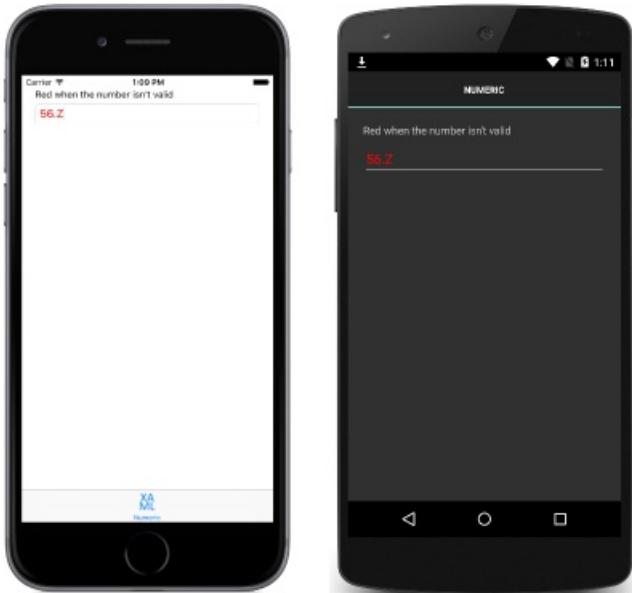
The equivalent `Entry` in C# is shown in the following code example:

```

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 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<T>](#)

Reusable EffectBehavior

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

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<T>](#)

Xamarin.Forms Custom Renderers

8/4/2022 • 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 Pin

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 a custom renderer 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.

Customizing a WebView

A Xamarin.Forms `WebView` is a view that displays web and HTML content in your app. This article explains how to create a custom renderer that extends the `WebView` to allow C# code to be invoked from JavaScript.

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.

Introduction to Custom Renderers

8/4/2022 • 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:

Hello, Custom Renderer !
In Shared Code

Hello, Custom Renderer !
In Shared Code

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

8/4/2022 • 3 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`
- Android (FastRenderers) - `Xamarin.Forms.Platform.Android.FastRenderers`
- Universal Windows Platform (UWP) – `Xamarin.Forms.Platform.UWP`

For more information about fast renderers, see [Xamarin.Forms Fast Renderers](#).

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`

NOTE

For information about creating custom renderers for Shell applications, see [Xamarin.Forms Shell Custom Renderers](#).

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
<code>ContentPage</code>	<code>PageRenderer</code>	<code>UIViewController</code>	<code>ViewGroup</code>		<code>FrameworkElement</code>
<code>FlyoutPage</code>	<code>PhoneFlyoutPageRenderer</code> (iOS – Phone), <code>TabletFlyoutPageRenderer</code> (iOS – Tablet), <code>MasterDetailRenderer</code> (Android), <code>FlyoutPageRenderer</code> (Android AppCompat), <code>FlyoutPageRenderer</code> (UWP)	<code>UIViewController</code> (Phone), <code>UISplitViewController</code> (Tablet)	<code>DrawerLayout</code> (v4)	<code>DrawerLayout</code> (v4)	<code>FrameworkElement</code> (Custom Control)

PAGE	RENDERER	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
NavigationPage	NavigationRenderer (iOS and Android), NavigationPageRenderer (Android AppCompat), NavigationPageRenderer (UWP)	UIToolbar	ViewGroup	ViewGroup	FrameworkElement (Custom Control)
TabbedPage	TabbedRenderer (iOS and Android), TabbedPageRenderer (Android AppCompat), TabbedPageRenderer (UWP)	UIView	ViewPager	ViewPager	FrameworkElement (Pivot)
TemplatedPage	PageRenderer	UIViewController	ViewGroup		FrameworkElement
CarouselPage	CarouselPageRenderer	UIScrollView	ViewPager	ViewPager	FrameworkElement (FlipView)

Layouts

The following table lists the renderer and native control classes that implement each Xamarin.Forms [Layout](#) type:

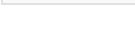
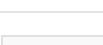
LAYOUT	RENDERER	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
ContentPresenter	ViewRenderer	UIView	View	View	FrameworkElement
ContentView	ViewRenderer	UIView	View	View	FrameworkElement
FlexLayout	ViewRenderer	UIView	View	View	FrameworkElement
Frame	FrameRenderer	UIView	ViewGroup	CardView	Border
ScrollView	ScrollViewRenderer	UIScrollView	ScrollView	ScrollView	ScrollViewer
TempledView	ViewRenderer	UIView	View	View	FrameworkElement
AbsoluteLayout	ViewRenderer	UIView	View	View	FrameworkElement
Grid	ViewRenderer	UIView	View	View	FrameworkElement

LAYOUT	RENDERER	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
<code>RelativeLayout</code>	ViewRenderer	UIView	View	View	FrameworkElement
<code>StackLayout</code>	ViewRenderer	UIView	View	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
<code>ActivityIndicator</code>	ActivityIndicator Renderer	UIActivityIndicatorView	ProgressBar		ProgressBar
<code>BoxView</code>	BoxRenderer (iOS and Android), BoxViewRenderer (UWP)	UIView	ViewGroup		Rectangle
<code>Button</code>	ButtonRenderer	UIButton	Button	AppCompatButton	Button
<code>CarouselView</code>	CarouselViewRenderer	UICollectionView		RecyclerView	ListViewBase
<code>CheckBox</code>	CheckBoxRenderer	UIButton		AppCompatCheckBox	CheckBox
<code>CollectionView</code>	CollectionViewRenderer	UICollectionView		RecyclerView	ListViewBase
<code>DatePicker</code>	DatePickerRenderer	UITextField	EditText		DatePicker
<code>Editor</code>	EditorRenderer	UITextView	EditText		TextBox
<code>Ellipse</code>	EllipseRenderer	CALayer	View		Ellipse
<code>Entry</code>	EntryRenderer	UITextField	EditText		TextBox
<code>Image</code>	ImageRenderer	UIImageView	ImageView		Image
<code>ImageButton</code>	ImageButtonRenderer	UIButton		AppCompatImageButton	Button
<code>IndicatorView</code>	IndicatorViewRenderer	UIPageControl		LinearLayout	
<code>Label</code>	LabelRenderer	UILabel	TextView		TextBlock

VIEWS	RENDERER	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
 Line	LineRenderer	CALayer	View		Line
 ListView	ListViewRenderer	UITableView	ListView		ListView
 Map	MapRenderer	MKMapView	MapView		MapControl
 Path	PathRenderer	CALayer	View		Path
 Picker	PickerRenderer	UITextField	EditText	EditText	ComboBox
 Polygon	PolygonRenderer	CALayer	View		Polygon
 Polyline	PolylineRenderer	CALayer	View		Polyline
 ProgressBar	ProgressBarRenderer	UIProgressView	ProgressBar		ProgressBar
 RadioButton	RadioButtonRenderer	UIButton		AppCompatRadioButton	RadioButton
 Rectangle	RectangleRenderer	CALayer	View		Rectangle
 RefreshView	RefreshViewRenderer	UIView		SwipeRefreshLayout	RefreshContainer
 SearchBar	SearchBarRenderer	UISearchBar	SearchView		AutoSuggestBox
 Slider	SliderRenderer	UISlider	SeekBar		Slider
 Stepper	StepperRenderer	UIStepper	LinearLayout		Control
 SwipeView	SwipeViewRenderer	UIView		View	SwipeControl
 Switch	SwitchRenderer	UISwitch	Switch	SwitchCompat	ToggleSwitch
 TableView	TableViewRenderer	UITableView	ListView		ListView
 TimePicker	TimePickerRenderer	UITextField	EditText		TimePicker
 WebView	WkWebViewRenderer (iOS), WebViewRenderer (Android and UWP)	WkWebView	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
EntryCell	EntryCellRenderer	UITableViewCell with a UITextField	LinearLayout with a TextView and EditText	DataTemplate with a TextBox
SwitchCell	SwitchCellRenderer	UITableViewCell with a UISwitch	Switch	DataTemplate with a Grid containing a TextBlock and ToggleSwitch
TextCell	TextCellRenderer	UITableViewCell	LinearLayout with two TextViews	DataTemplate with a StackPanel containing two TextBlocks
ImageCell	ImageCellRenderer	UITableViewCell with a UIImage	LinearLayout with two TextViews and an ImageView	DataTemplate with a Grid containing an Image and two TextBlocks
ViewCell	ViewCellRenderer	UITableViewCell	View	DataTemplate with a ContentPresenter

Related links

- [Xamarin.Forms Fast Renderers](#)
- [Xamarin.Forms Shell Custom Renderers](#)

Customizing an Entry

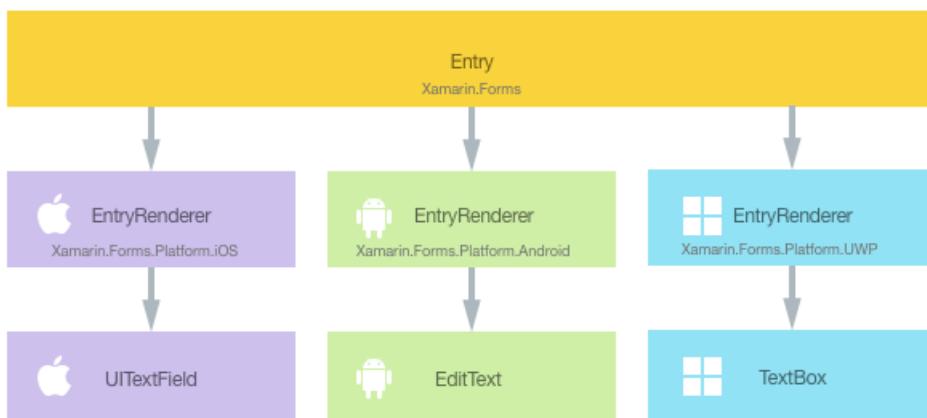
8/4/2022 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

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.

IMPORTANT

This article explains how to create a simple custom renderer. However, it's not necessary to create a custom renderer to implement an `Entry` that has a different background color on each platform. This can be more easily accomplished by using the `Device` class, or the `OnPlatform` markup extension, to provide platform-specific values. For more information, see [Providing Platform-Specific Values](#) and [OnPlatform Markup Extension](#).

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

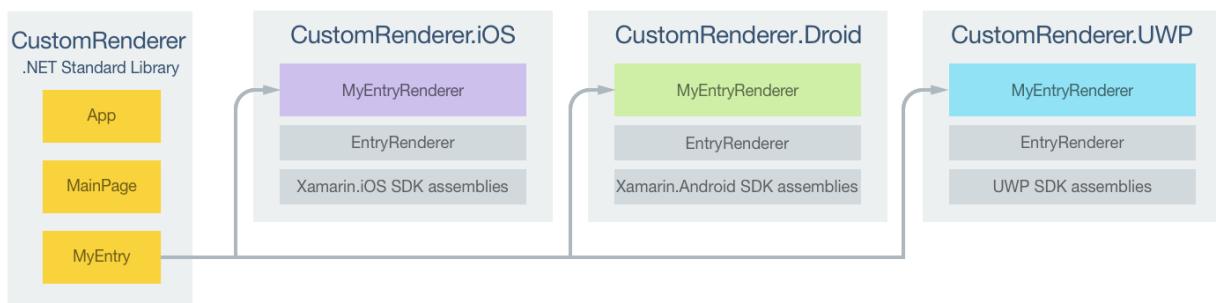
The process for creating the custom renderer class is as follows:

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

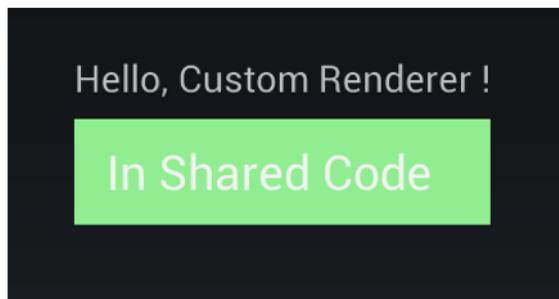
It is optional to provide a custom renderer in each platform project. If a custom renderer isn't registered, then the default renderer for the control's base class will be used.

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:

Hello, Custom Renderer !
In Shared Code



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

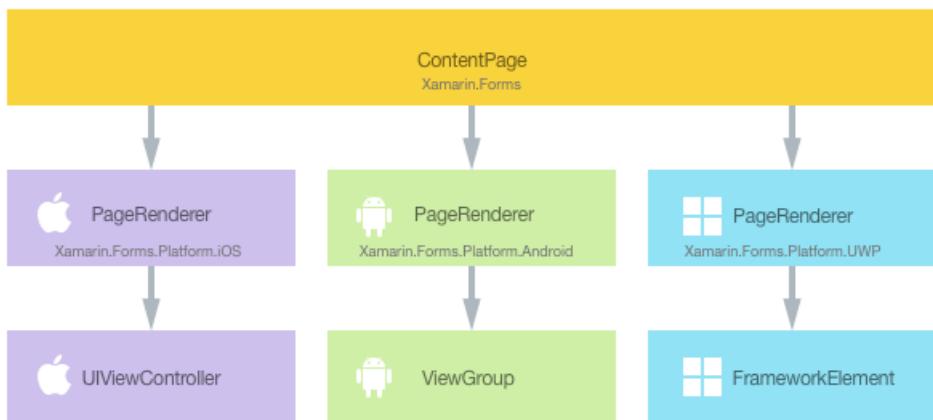
8/4/2022 • 7 minutes to read • [Edit Online](#)

 [Download the sample](#)

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

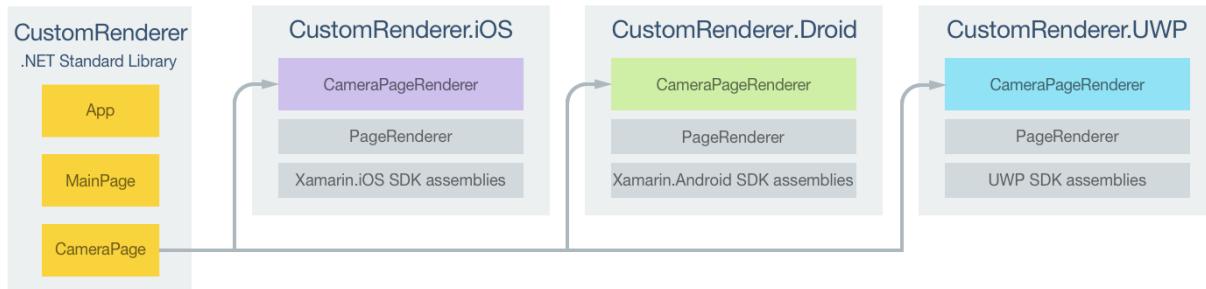
The process for creating the custom renderer class is as follows:

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.

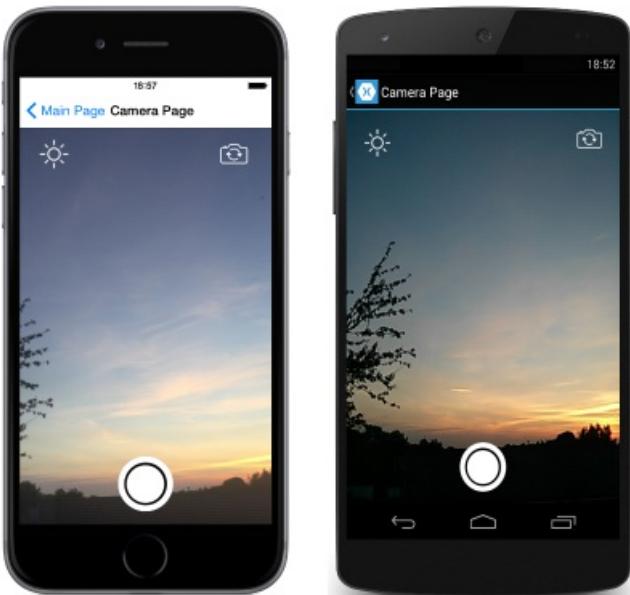
NOTE

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 Map Pin

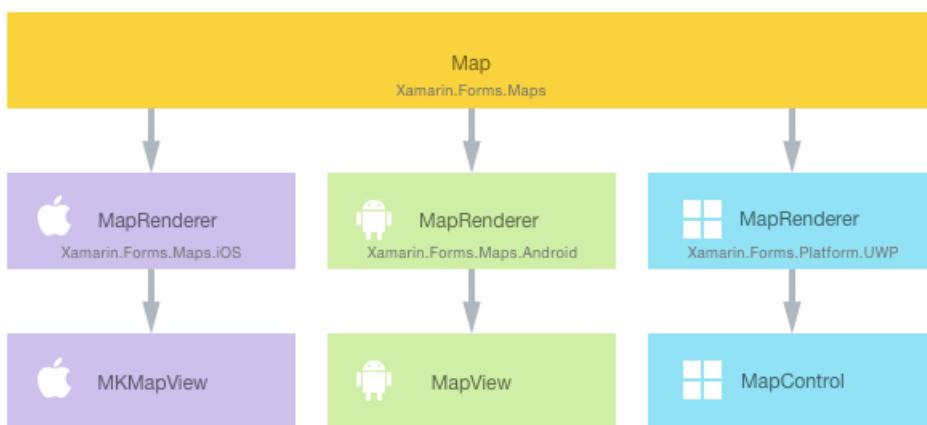
8/4/2022 • 20 minutes to read • [Edit Online](#)

 [Download the sample](#)

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 Name { get; set; }
    public string Url { get; set; }
}
```

This class defines a `CustomPin` as inheriting the properties of the `Pin` class, and adding `Name` and `Url` properties.

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">
    <local:CustomMap x:Name="customMap"
        MapType="Street" />
</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()
    {
        CustomMap customMap = new CustomMap
        {
            MapType = MapType.Street
        };
        // ...
        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.

The location of the map, and the pins it contains, are initialized as shown in the following code example:

```

public MapPage()
{
    // ...
    CustomPin 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",
        Name = "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

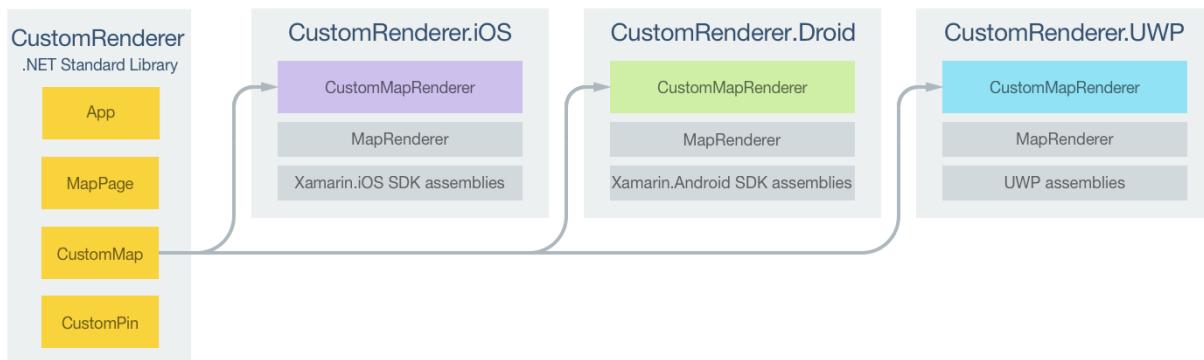
The process for creating the custom renderer class is as follows:

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.

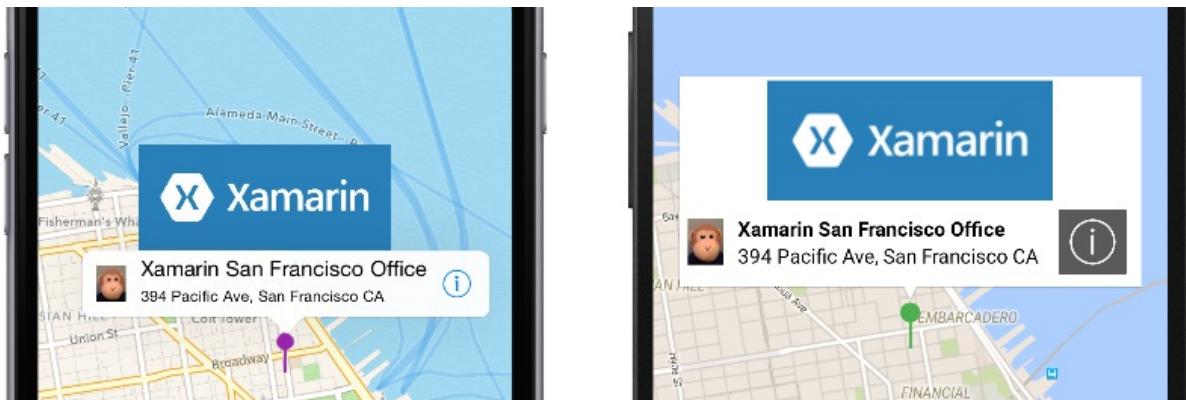
NOTE

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.View> 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
    }
}
```

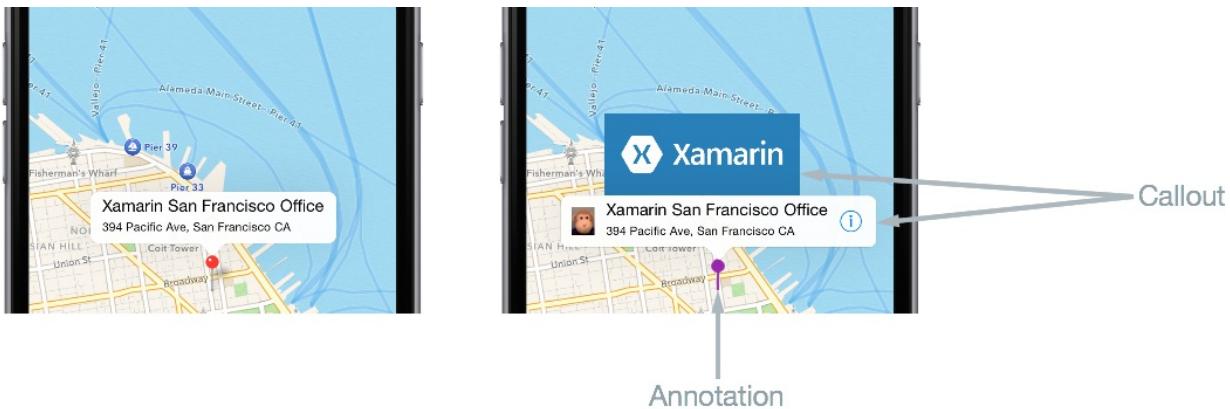
The native control should be configured and event handlers subscribed to only when the custom renderer is attached to a new `Xamarin.Forms` element. Similarly, any event handlers that were subscribed to should be unsubscribed from only when the element that the renderer is attached to changes. Adopting 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.Name);
    if (annotationView == null)
    {
        annotationView = new CustomMKAnnotationView(annotation, customPin.Name);
        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).Name = customPin.Name;
        ((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:

1. The `GetCustomPin` method is called to return the custom pin data for the annotation.
2. To conserve memory, the annotation's view is pooled for reuse with the call to `DequeueReusableAnnotation`.
3. The `CustomMKAnnotationView` class extends the `MKAnnotationView` class with `Name` 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.Name` property is set to the `CustomPin.Name` 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](#).
- The `MKAnnotationView.CanShowCallout` property is set to `true` so that the callout is displayed when the annotation is tapped.
 - 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)
{
    CustomMKAnnotationView customView = e.View as CustomMKAnnotationView;
    customPinView = new UIView();

    if (customView.Name.Equals("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 `Name` 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)
{
    CustomMKAnnotationView 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:

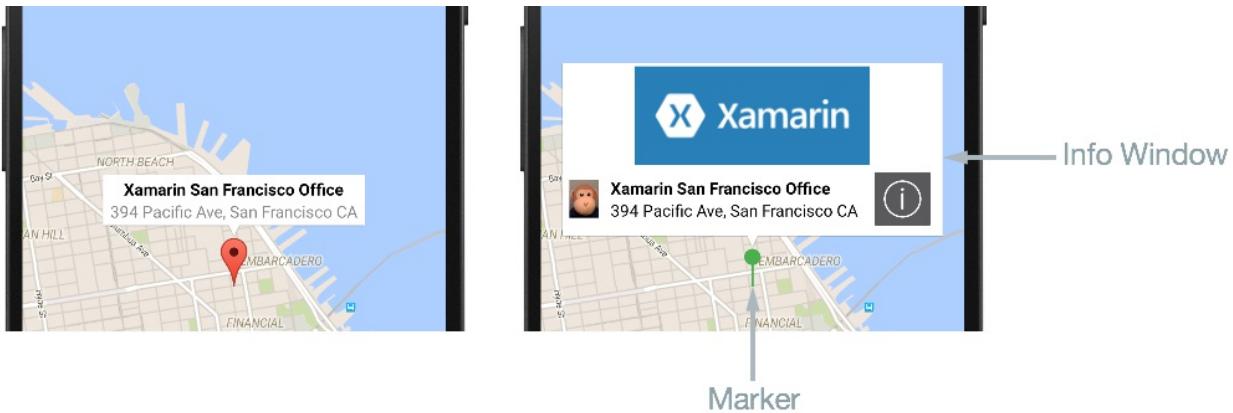
```
void OnDidDeselectAnnotationView(object sender, MKAnnotationEventArgs e)
{
    if (!e.View.Selected)
    {
        customPinView.RemoveFromSuperview();
        customPinView.Dispose();
        customPinView = null;
    }
}
```

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;
            }
        }

        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 retrieves the list of custom pins from the control. 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.GetSystemService(Context.LayoutInflaterService) as
Android.Views.LayoutInflator;
    if (inflater != null)
    {
        Android.Views.View view;

        var customPin = GetCustomPin(marker);
        if (customPin == null)
        {
            throw new Exception("Custom pin not found");
        }

        if (customPin.Name.Equals("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 `LayoutInflator` 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.Name` 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.

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");
    }

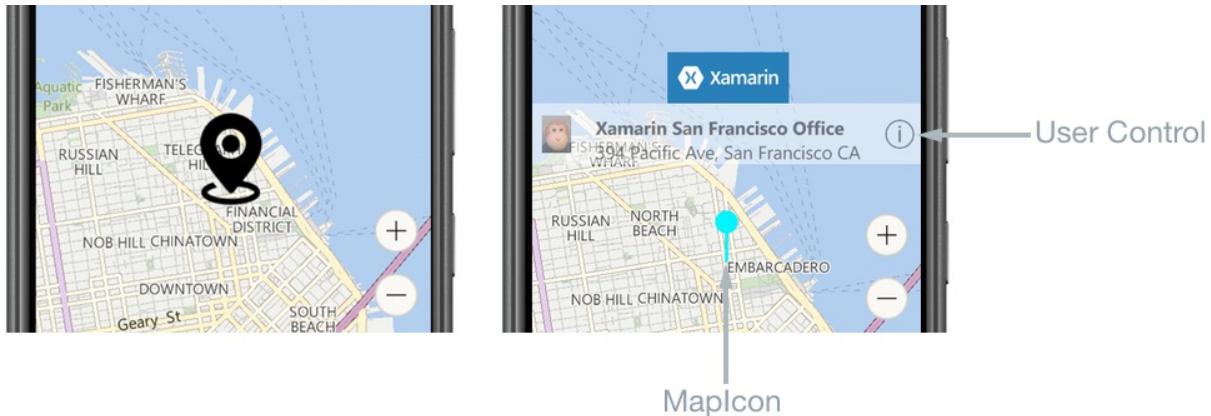
    if (!string.IsNullOrWhiteSpace(customPin.Url))
    {
        var url = Android.Net.Uri.Parse(customPin.Url);
        var intent = new Intent(Intent.ActionView, url);
        intent.AddFlags(ActivityFlags.NewTask);
        Android.App.Application.Context.StartActivity(intent);
    }
}
```

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 `UserControl1`, which is displayed in response to the user tapping on the map icon. The `UserControl1` 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, MapElementClickEventArgs 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.Name.Equals("Xamarin"))
            {
                if (mapOverlay == null)
                {
                    mapOverlay = new XamarinMapOverlay(customPin);
                }

                var snPosition = new BasicGeoposition { Latitude = customPin.Position.Latitude, Longitude = customPin.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.

Related Links

- [Maps Control](#)
- [iOS Maps](#)
- [Maps API](#)
- [Customized Pin \(sample\)](#)

Customizing a ListView

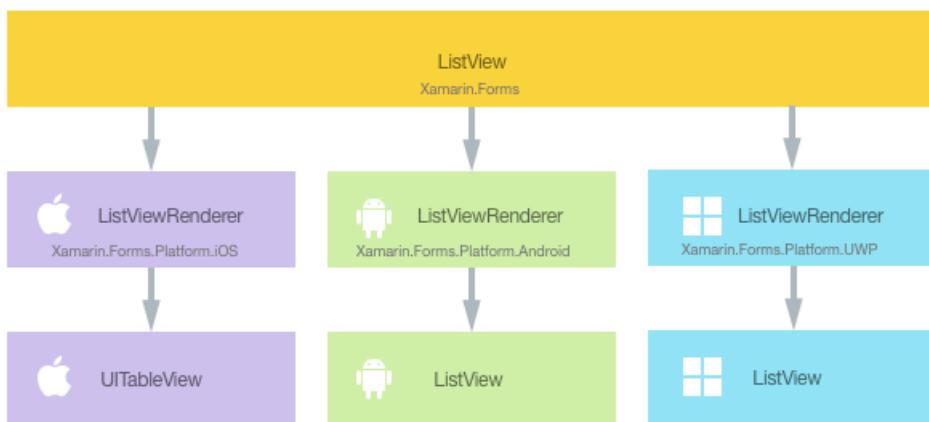
8/4/2022 • 16 minutes to read • [Edit Online](#)

 [Download the sample](#)

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.

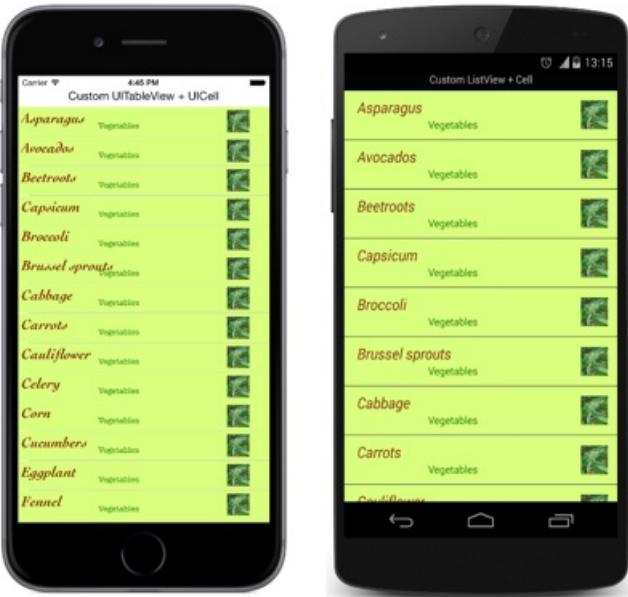
NOTE

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 `UITableViewCell` 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.IsNullOrWhiteSpace (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 NativeiOSListViewCell : UITableViewCell
{
    UILabel headingLabel, subheadingLabel;
    UIImageView imageView;

    public NativeiOSListViewCell (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 `NativeiOSListViewCell` 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 NativeAndroidListAdapter(_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 `NativeAndroidListAdapter` 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="10dip">
        <TextView
            android:id="@+id/Text1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="#FF7F3300"
            android:textSize="20dip"
            android:textStyle="italic" />
        <TextView
            android:id="@+id/Text2"
            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`.

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 it's `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

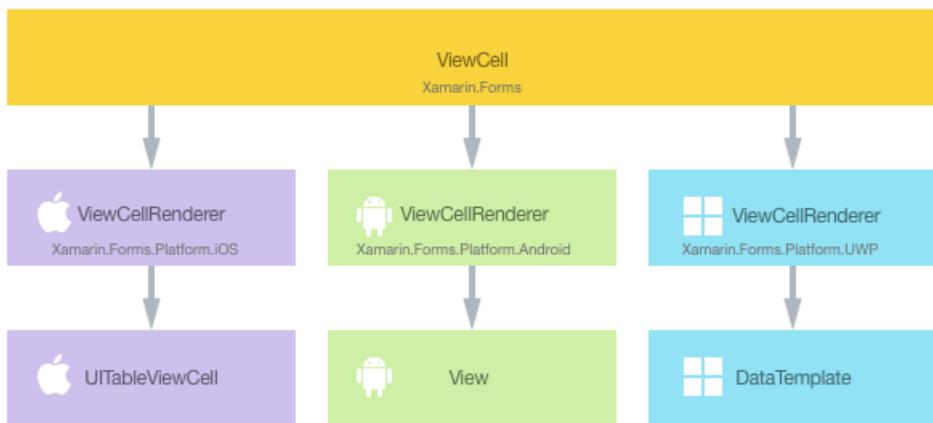
8/4/2022 • 14 minutes to read • [Edit Online](#)

 [Download the sample](#)

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:

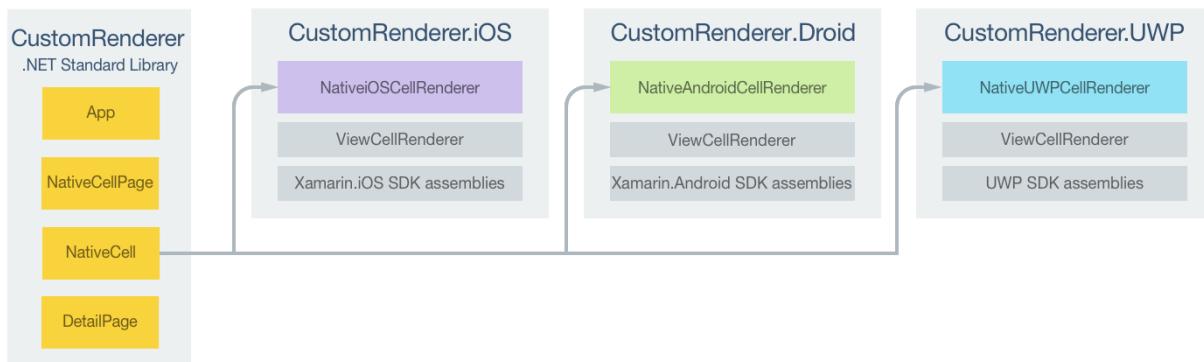
1. Create a subclass of the `ViewCellRenderer` class that renders the custom cell.

- 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`.

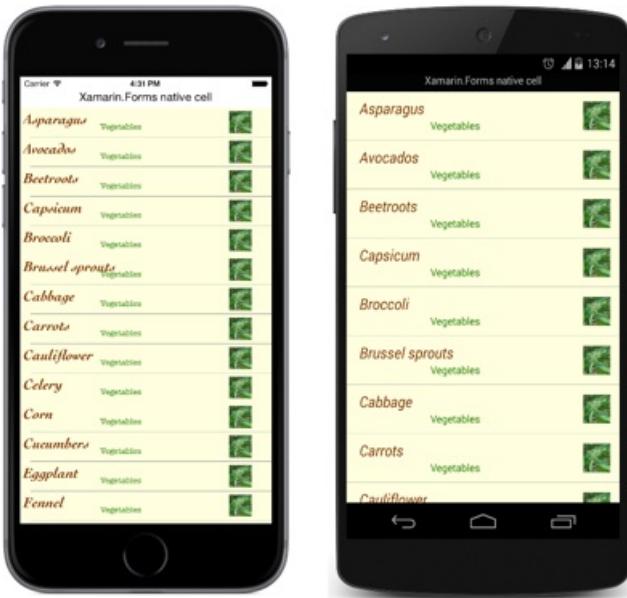
NOTE

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, UITableViewCell 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 SubtitleLabel { get; set; }
    public UIImageView CellImageView { get; set; }

    public NativeCell NativeCell { get; private set; }
    public Element Element => NativeCell;

    public NativeiOSCell(string cellId, NativeCell cell) : base(TableViewCellStyle.Default, cellId)
    {
        NativeCell = cell;

        SelectionStyle = TableViewCellSelectionStyle.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
        };

        SubtitleLabel = new UILabel()
        {
            Font = UIFont.FromName("AmericanTypewriter", 12f),
            TextColor = UIColor.FromRGB(38, 127, 0),
            TextAlignment = NSTextAlignment.Center,
            BackgroundColor = UIColor.Clear
        };

        ContentView.Add(HeadingLabel);
        ContentView.Add(SubtitleLabel);
        ContentView.Add(CellImageView);
    }

    public void UpdateCell(NativeCell cell)
    {
        HeadingLabel.Text = cell.Name;
        SubtitleLabel.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);
        SubtitleLabel.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`, `SubtitleLabel`, 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="10dp">
        <TextView
            android:id="@+id/HeadingText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="#FF7F3300"
            android:textSize="20dp"
            android:textStyle="italic" />
        <TextView
            android:id="@+id/SubheadingText"
            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`.

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 \(sample\)](#)

Customizing a WebView

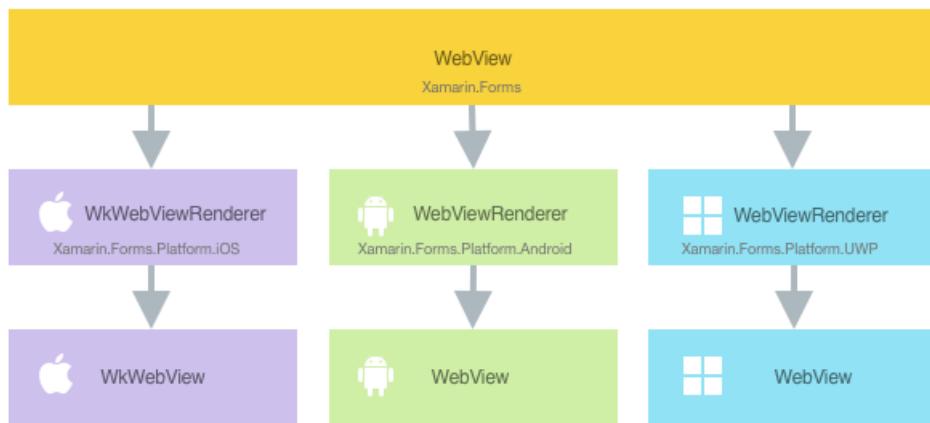
8/4/2022 • 14 minutes to read • [Edit Online](#)

 [Download the sample](#)

A `Xamarin.Forms` `WebView` is a view that displays web and HTML content in your app. This article explains how to create a custom renderer that extends the `WebView` 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 `WebView` is rendered by a `Xamarin.Forms` application on iOS, the `WkWebViewRenderer` class is instantiated, which in turn instantiates a native `WkWebView` control. On the Android platform, the `WebViewRenderer` class instantiates a native `WebView` control. On the Universal Windows Platform (UWP), the `WebViewRenderer` class instantiates a native `WebView` 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 customizations by creating a custom renderer for a `WebView` 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 `Xamarin.Forms` `WebView` 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 [Invoke C# from JavaScript](#). For more information about the HTML page, see [Create the Web Page](#).

NOTE

A `WebView` can invoke a JavaScript function from C#, and return any result to the calling C# code. For more information, see [Invoking JavaScript](#).

Create the HybridWebView

The `HybridWebView` custom control can be created by subclassing the `WebView` class:

```
public class HybridWebView : WebView
{
    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 project.

Consume 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="0,40,0,0">
    <local:HybridWebView x:Name="hybridWebView"
        Uri="index.html" />
</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 HybridWebViewPageCS()
{
    var hybridWebView = new HybridWebView
    {
        Uri = "index.html"
    };
    // ...
    Padding = new Thickness(0, 40, 0, 0);
    Content = hybridWebView;
}

```

The `HybridWebView` instance will be used to display a native web control on each platform. Its `Uri` property is set to an HTML file that is stored in each platform 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 web controls by allowing C# code to be invoked from JavaScript.

Create the custom renderer on each platform

The process for creating the custom renderer class is as follows:

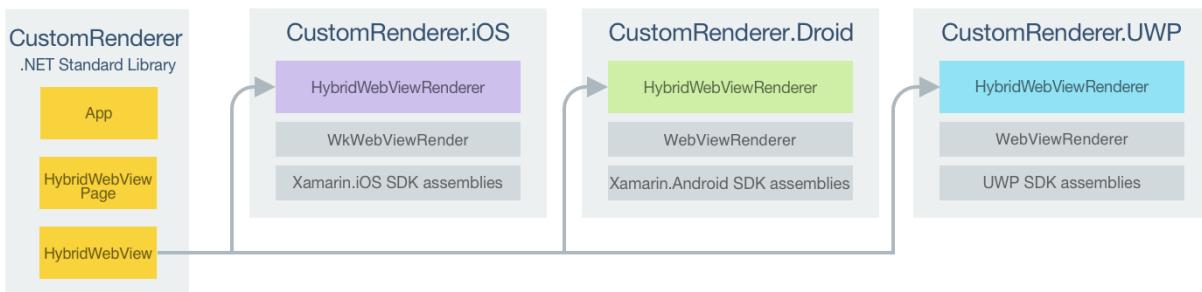
1. Create a subclass of the `WKWebViewRenderer` class on iOS, and the `WebViewRenderer` class on Android and UWP, that renders the custom control.
2. Override the `OnElementChanged` method that renders the `WebView` and write logic to customize it. This method is called when a `HybridWebView` object is created.

3. Add an `ExportRenderer` attribute to the custom renderer class or `AssemblyInfo.cs`, 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 renderer classes, which derive from the `WkWebViewRenderer` class on iOS, and from the `WebViewRenderer` class on Android and UWP. This results in each `HybridWebView` custom control being rendered with native web controls, as shown in the following screenshots:



The `WkWebViewRenderer` and `WebViewRenderer` classes expose the `OnElementChanged` method, which is called when the Xamarin.Forms custom control is created to render the corresponding native web control. This method takes a `VisualElementChangedEventArgs` 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 renderer class, is the place to perform the native web control customization. A reference to the Xamarin.Forms control 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 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 custom renderer class.

Create 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-2.1.4.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.

On iOS, this HTML file resides in the Content folder of the platform project, with a build action of `BundleResource`. On Android, this HTML file resides in the Assets/Content folder of the platform project, with a build action of `AndroidAsset`.

Invoke 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.

Create 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 : WkWebViewRenderer, IWKScriptMessageHandler
    {
        const string JavaScriptFunction = "function invokeCSharpAction(data)
{window.webkit.messageHandlers.invokeAction.postMessage(data);}";
        WKUserContentController userController;

        public HybridWebViewRenderer() : this(new WKWebViewConfiguration())
        {
        }

        public HybridWebViewRenderer(WKWebViewConfiguration config) : base(config)
        {
            userController = config.UserContentController;
            var script = new WKUserScript(new NSString(JavaScriptFunction),
WKUserScriptInjectionTime.AtDocumentEnd, false);
            userController.AddUserScript(script);
            userController.AddScriptMessageHandler(this, "invokeAction");
        }

        protected override void OnElementChanged(VisualElementChangedEventArgs e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                userController.RemoveAllUserScripts();
                userController.RemoveScriptMessageHandler("invokeAction");
                HybridWebView hybridWebView = e.OldElement as HybridWebView;
                hybridWebView.Cleanup();
            }

            if (e.NewElement != null)
            {
                string filename = Path.Combine(NSBundle.MainBundle.BundlePath,
$"Content/{{(HybridWebView)Element}.Uri}");
                LoadRequest(new URLRequest(new NSURL(filename, false)));
            }
        }

        public void DidReceiveScriptMessage(WKUserContentController userContentController, WKScriptMessage
message)
        {
            ((HybridWebView)Element).InvokeAction(message.Body.ToString());
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                ((HybridWebView)Element).Cleanup();
            }
            base.Dispose(disposing);
        }
    }
}

```

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:

- The renderer constructor creates a `WkWebViewConfiguration` object, and retrieves its `WKUserContentController` object. The `WKUserContentController` object allows posting messages and injecting user scripts into a web page.
- The renderer constructor creates a `WKUserScript` object, which injects the `invokeCSharpAction` JavaScript function into the web page after the web page is loaded.
- The renderer constructor calls the `WKUserContentController.AddUserScript` method to add the `WKUserScript` object to the content controller.
- The renderer constructor calls the `WKUserContentController.AddScriptMessageHandler` method to add a script message handler named `invokeAction` to the `WKUserContentController` object, which will cause the JavaScript function `window.webkit.messageHandlers.invokeAction.postMessage(data)` to be defined in all frames in all `WebView` instances that use the `WKUserContentController` object.
- 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.
- Resources are released when the element the renderer is attached to changes.
- The `Xamarin.Forms` element is cleaned up when the renderer is disposed of.

NOTE

The `WKWebView` class is only supported in iOS 8 and later.

In addition, `Info.plist` must be updated to include the following values:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

Create 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 : WebViewRenderer
    {
        const string JavascriptFunction = "function invokeCSharpAction(data){jsBridge.invokeAction(data);}";
        Context _context;

        public HybridWebViewRenderer(Context context) : base(context)
        {
            _context = context;
        }

        protected override void OnElementChanged(ElementChangedEventArgs<WebView> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                Control.RemoveJavascriptInterface("jsBridge");
                ((HybridWebView)Element).Cleanup();
            }
            if (e.NewElement != null)
            {
                Control.SetWebViewClient(new JavascriptWebViewClient(this, $"javascript:{JavascriptFunction}"));
                Control.AddJavascriptInterface(new JSBridge(this), "jsBridge");
                Control.LoadUrl($"file:///android_asset/Content/{{((HybridWebView)Element).Uri}}");
            }
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                ((HybridWebView)Element).Cleanup();
            }
            base.Dispose(disposing);
        }
    }
}
```

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 : FormsWebViewClient
{
    string _javascript;

    public JavascriptWebViewClient(HybridWebViewRenderer renderer, string javascript) : base(renderer)
    {
        _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 custom renderer is attached to a new Xamarin.Forms element:
 - The `SetWebViewClient` method sets a new `JavascriptWebViewClient` object as the implementation of `WebViewClient`.
 - 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.
- Resources are released when the element the renderer is attached to changes.
- The Xamarin.Forms element is cleaned up when the renderer is disposed of.

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))
        {
            ((HybridWebView)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.

IMPORTANT

Android 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](#).

Create 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 : WebViewRenderer
    {
        const string JavaScriptFunction = "function invokeCSharpAction(data)
{window.external.notify(data);}";

        protected override void OnElementChanged(ElementChangedEventArgs<Xamarin.Forms.WebView> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                Control.NavigationCompleted -= OnWebViewNavigationCompleted;
                Control.ScriptNotify -= OnWebViewScriptNotify;
            }
            if (e.NewElement != null)
            {
                Control.NavigationCompleted += OnWebViewNavigationCompleted;
                Control.ScriptNotify += OnWebViewScriptNotify;
                Control.Source = new Uri($"ms-appx-web:///Content//{((HybridWebView)Element).Uri}");
            }
        }

        async void OnWebViewNavigationCompleted(Windows.UI.Xaml.Controls.WebView sender,
        WebViewNavigationCompletedEventArgs args)
        {
            if (args.IsSuccess)
            {
                // Inject JS script
                await Control.InvokeScriptAsync("eval", new[] { JavaScriptFunction });
            }
        }

        void OnWebViewScriptNotify(object sender, NotifyEventArgs e)
        {
            ((HybridWebView)Element).InvokeAction(e.Value);
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                ((HybridWebView)Element).Cleanup();
            }
            base.Dispose(disposing);
        }
    }
}
```

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 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.
- Event are unsubscribed from when the element the renderer is attached to changes.
- The `Xamarin.Forms` element is cleaned up when the renderer is disposed of.

Related links

- [HybridWebView \(sample\)](#)

Implementing a View

8/4/2022 • 10 minutes to read • [Edit Online](#)

 [Download the sample](#)

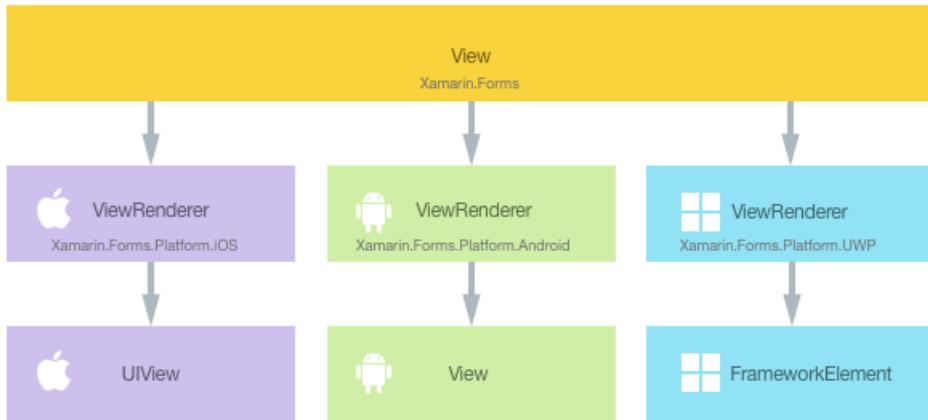
Xamarin.Forms custom user interface controls should derive from the `View` class, which is used to place layouts and controls on the screen. This article demonstrates how to create a custom renderer for a Xamarin.Forms custom control that's used to display a preview video stream from the device's camera.

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](#).

NOTE

Some controls on Android use fast renderers, which don't consume the `ViewRenderer` class. For more information about fast renderers, see [Xamarin.Forms Fast Renderers](#).

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 .NET Standard library 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 .NET Standard library 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"
    ...>
    <StackLayout>
        <Label Text="Camera Preview:" />
        <local:CameraPreview Camera="Rear"
            HorizontalOptions="FillAndExpand"
            VerticalOptions="FillAndExpand" />
    </StackLayout>
</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 on iOS and UWP 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.

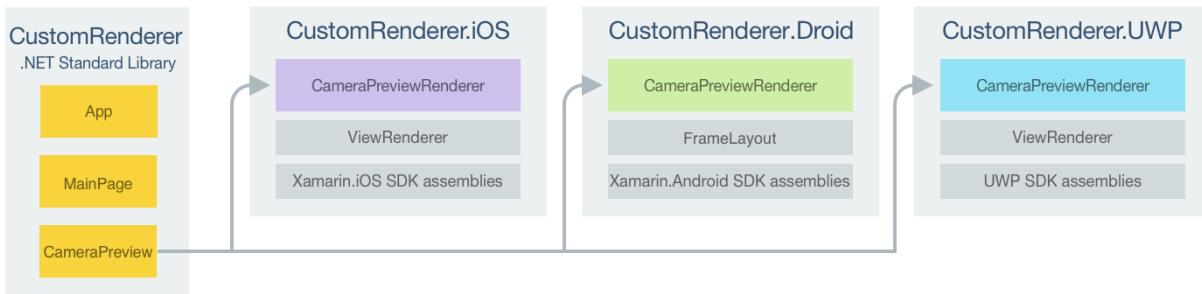
The process for creating the custom renderer class on Android, as a fast renderer, is as follows:

1. Create a subclass of the Android control that renders the custom control. In addition, specify that the subclass will implement the `IVisualElementRenderer` and `IViewRenderer` interfaces.
2. Implement the `IVisualElementRenderer` and `IViewRenderer` interfaces in the fast renderer class.
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.

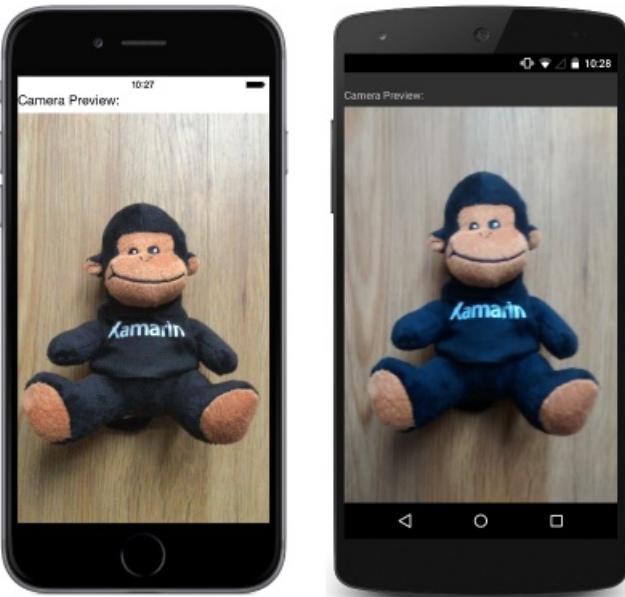
NOTE

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 derive from the `ViewRenderer` class on iOS and UWP, and from the `FrameLayout` class on Android. 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 (e.OldElement != null)
    {
        // Unsubscribe from event handlers and cleanup any resources
    }

    if (e.NewElement != null)
    {
        if (Control == null)
        {
            // Instantiate the native control and assign it to the Control property with
            // the SetNativeControl method
        }
        // Configure the control and subscribe to event handlers
    }
}
```

A new native control should only be instantiated once, when the `control` property is `null`. In addition, the control should only be created, 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.

IMPORTANT

The `SetNativeControl` method should only be called if `e.NewElement` is not `null`.

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 (e.OldElement != null) {
                // Unsubscribe
                uiCameraPreview.Tapped -= OnCameraPreviewTapped;
            }
            if (e.NewElement != null) {
                if (Control == null) {
                    uiCameraPreview = new UICameraPreview (e.NewElement.Camera);
                    SetNativeControl (uiCameraPreview);
                }
                // 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 fast renderer for the Android platform:

```
[assembly: ExportRenderer(typeof(CustomRenderer.CameraPreview), typeof(CameraPreviewRenderer))]
namespace CustomRenderer.Droid
{
    public class CameraPreviewRenderer : FrameLayout, IVisualElementRenderer, IViewRenderer
    {
        // ...
        CameraPreview element;
        VisualElementTracker visualElementTracker;
        VisualElementRenderer visualElementRenderer;
        FragmentManager fragmentManager;
        CameraFragment cameraFragment;

        FragmentManager FragmentManager => fragmentManager ??= Context.GetFragmentManager();
```

```

public event EventHandler<VisualElementChangedEventArgs> ElementChanged;
public event EventHandler<PropertyChangedEventArgs> ElementPropertyChanged;

CameraPreview Element
{
    get => element;
    set
    {
        if (element == value)
        {
            return;
        }

        var oldElement = element;
        element = value;
        OnElementChanged(new ElementChangedEventArgs<CameraPreview>(oldElement, element));
    }
}

public CameraPreviewRenderer(Context context) : base(context)
{
    visualElementRenderer = new VisualElementRenderer(this);
}

void OnElementChanged(ElementChangedEventArgs<CameraPreview> e)
{
    CameraFragment newFragment = null;

    if (e.OldElement != null)
    {
        e.OldElement.PropertyChanged -= OnElementPropertyChanged;
        cameraFragment.Dispose();
    }
    if (e.NewElement != null)
    {
        this.EnsureId();

        e.NewElement.PropertyChanged += OnElementPropertyChanged;

        ElevationHelper.SetElevation(this, e.NewElement);
        newFragment = new CameraFragment { Element = element };
    }

    FragmentManager.BeginTransaction()
        .Replace(Id, cameraFragment = newFragment, "camera")
        .Commit();
    ElementChanged?.Invoke(this, new VisualElementChangedEventArgs(e.OldElement, e.NewElement));
}

async void OnElementPropertyChanged(object sender, PropertyChangedEventArgs e)
{
    ElementPropertyChanged?.Invoke(this, e);

    switch (e.PropertyName)
    {
        case "Width":
            await cameraFragment.RetrieveCameraDevice();
            break;
    }
}
// ...
}

```

In this example, the `OnElementChanged` method creates a `CameraFragment` object, provided that the custom renderer is attached to a new Xamarin.Forms element. The `CameraFragment` type is a custom class that uses the

`Camera2` API to provide the preview stream from the camera. The `CameraFragment` object is disposed of when 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 (e.OldElement != null)
            {
                // Unsubscribe
                Tapped -= OnCameraPreviewTapped;
                ...
            }
            if (e.NewElement != null)
            {
                if (Control == null)
                {
                    ...
                    _captureElement = new CaptureElement();
                    _captureElement.Stretch = Stretch.UniformToFill;

                    SetupCamera();
                    SetNativeControl(_captureElement);
                }
                // 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 \(sample\)](#)

Xamarin.Forms Data Binding

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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. 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.

Relative Bindings

Use relative bindings to set the binding source relative to the position of the binding target.

Binding Fallbacks

Make data bindings more robust by defining fallback values to use if the binding process fails.

Multi-Bindings

Attach a collection of `Binding` objects to a single binding target property.

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

8/4/2022 • 9 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

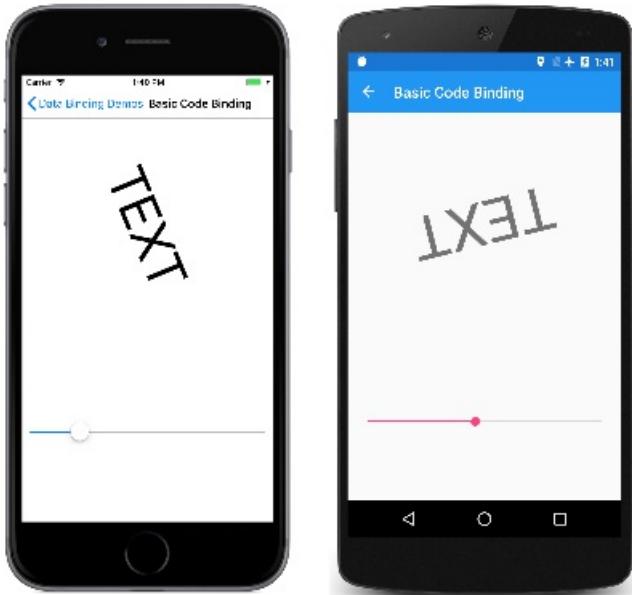
There is no such rule for the source property, which is specified as a string. Internally, reflection is used to access the actual property. In this particular case, however, the `Value` property is also backed by a bindable property.

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>
```

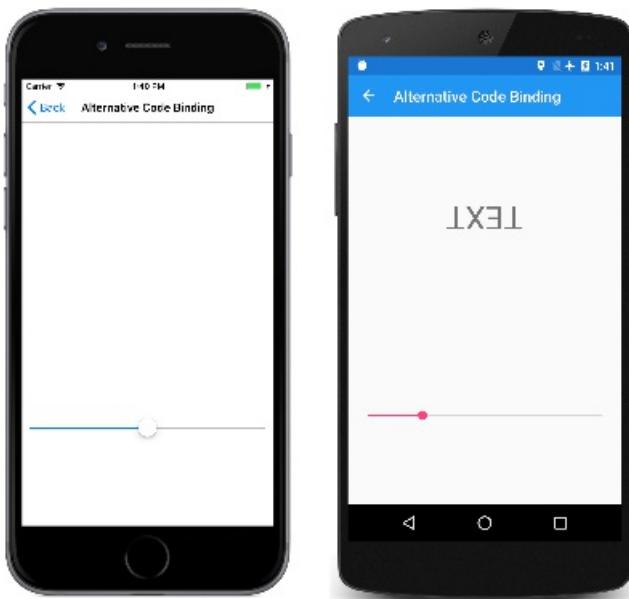
The code-behind file sets the binding with the `SetBinding` method defined by `BindableObject`. The argument is a `constructor` for the `Binding` class:

```
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 screenshot demonstrates, 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:

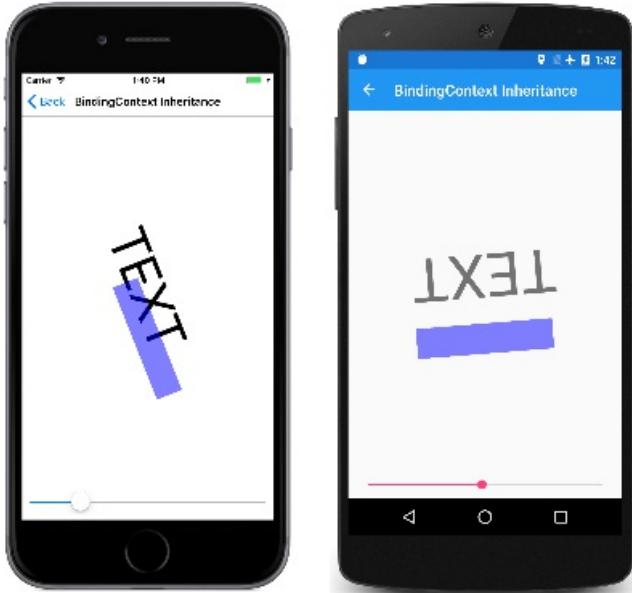
The setting of the `BindingContext` property is inherited through the visual tree.

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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Forms Binding Mode

8/4/2022 • 15 minutes to read • [Edit Online](#)

 [Download the sample](#)

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`:

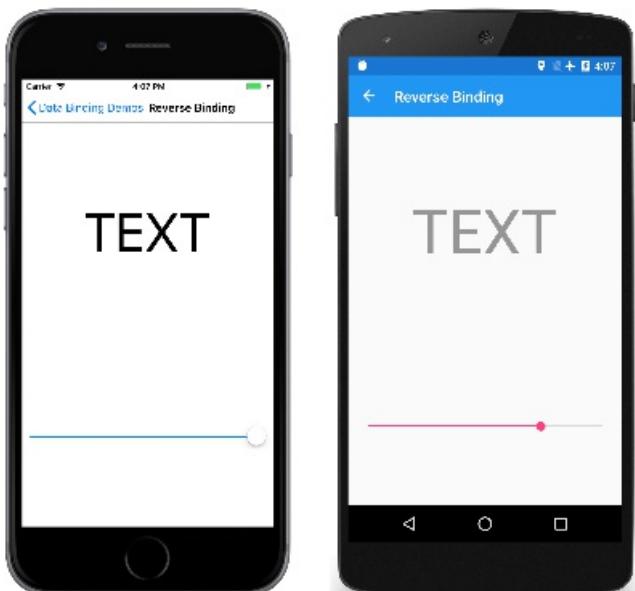
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataBindingDemos.ReverseBindingPage"
    Title="Reverse Binding">
    <StackLayout Padding="10, 0">

        <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 screenshot demonstrates. 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`, including the `IsTextPredictionEnabled` property of `Entry`.

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 event 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
{
    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}" />

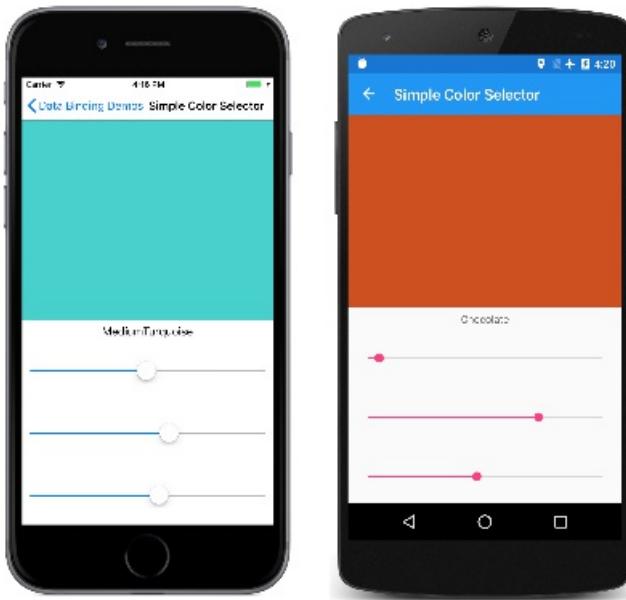
            <Slider Value="{Binding Luminosity}" />
        </StackLayout>
    </Grid>
</ContentPage>
```

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 screenshot.

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" />
        </StackLayout>
    </StackLayout>

```

```

        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" />

        <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}"
```

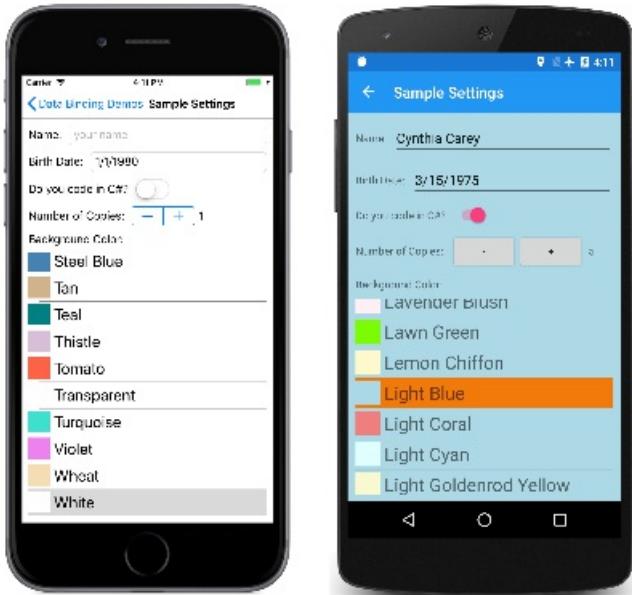
The default binding mode for `SelectedItem` is `OneWayToSource`, which sets the `ViewModel` property from the selected item. The `TwoWay` mode allows the `SelectedItem` to be initialized from the `ViewModel`.

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 screenshot shows 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

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

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=netstandard"
    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 `"` 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}" />
```

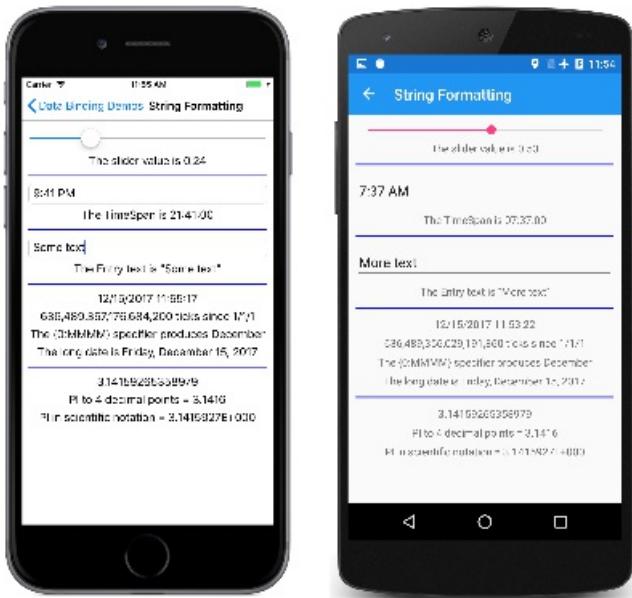
This simply displays the `DateTime` value of the `BindingContext` with default formatting. The second binding displays the `Ticks` property of `DateTime`, while the other two bindings display the `DateTime` itself with specific formatting. Notice this `StringFormat`:

```
<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}" />

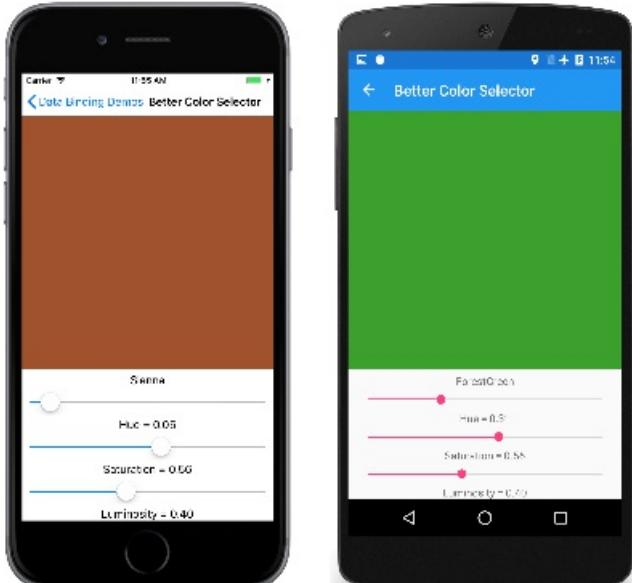
            <Slider Value="{Binding Hue}" />
            <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />

            <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

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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=netstandard"
    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>
```

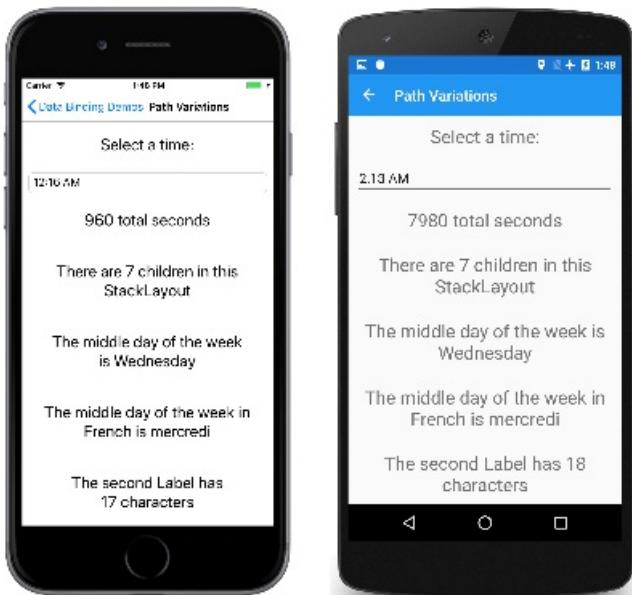
See [Passing Constructor Arguments](#) for more details on specifying constructor arguments in XAML.

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

8/4/2022 • 10 minutes to read • [Edit Online](#)



[Download the sample](#)

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" />

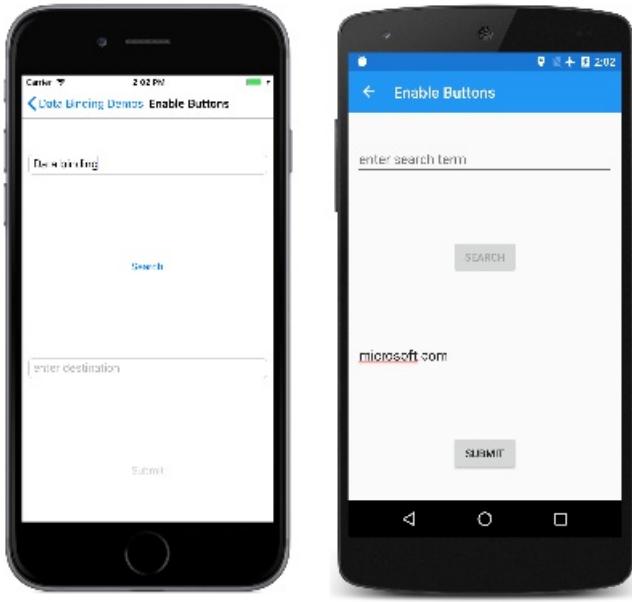
    <Button Text="Search"
        HorizontalOptions="Center"
        VerticalOptions="CenterAndExpand"
        IsEnabled="{Binding Source={x:Reference entry1},
            Path=Text.Length,
            Converter={StaticResource intToBool}}" />

    <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!"
                            FalseObject="No way!" />
                    </Binding.Converter>
                </Binding>
            </Label.Text>
        </Label>
    </StackLayout>

    <StackLayout Orientation="Horizontal"
        VerticalOptions="CenterAndExpand">
        <Label Text="Allow popups?" />
        <Switch x:Name="switch2" />
        <Label>
```

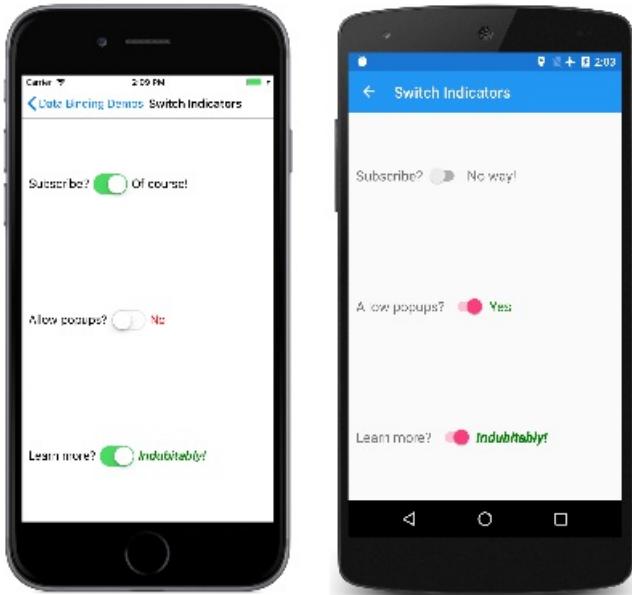
```

<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}'}" />

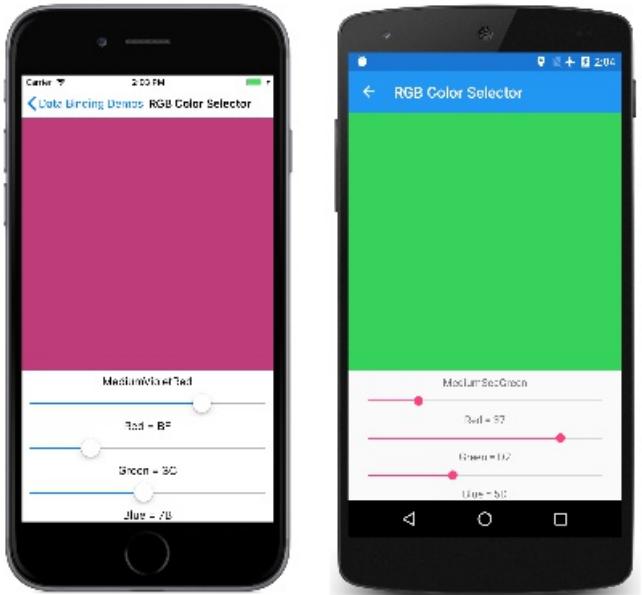
        <Slider Value="{Binding Green}" />
        <Label Text="{Binding Green,
                    Converter={StaticResource doubleToInt},
                    ConverterParameter=255,
                    StringFormat='Green = {0:X2}'}" />

        <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 Relative Bindings

8/4/2022 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

Relative bindings provide the ability to set the binding source relative to the position of the binding target. They are created with the `RelativeSource` markup extension, and set as the `Source` property of a binding expression.

The `RelativeSource` markup extension is supported by the `RelativeSourceExtension` class, which defines the following properties:

- `Mode`, of type `RelativeBindingSourceMode`, describes the location of the binding source relative to the position of the binding target.
- `AncestorLevel`, of type `int`, an optional ancestor level to look for, when the `Mode` property is `FindAncestor`. An `AncestorLevel` of `n` skips `n-1` instances of the `AncestorType`.
- `AncestorType`, of type `Type`, the type of ancestor to look for, when the `Mode` property is `FindAncestor`.

NOTE

The XAML parser allows the `RelativeSourceExtension` class to be abbreviated as `RelativeSource`.

The `Mode` property should be set to one of the `RelativeBindingSourceMode` enumeration members:

- `TemplatedParent` indicates the element to which the template, in which the bound element exists, is applied. For more information, see [Bind to a templated parent](#).
- `Self` indicates the element on which the binding is being set, allowing you to bind one property of that element to another property on the same element. For more information, see [Bind to self](#).
- `FindAncestor` indicates the ancestor in the visual tree of the bound element. This mode should be used to bind to an ancestor control represented by the `AncestorType` property. For more information, see [Bind to an ancestor](#).
- `FindAncestorBindingContext` indicates the `BindingContext` of the ancestor in the visual tree of the bound element. This mode should be used to bind to the `BindingContext` of an ancestor represented by the `AncestorType` property. For more information, see [Bind to an ancestor](#).

The `Mode` property is the content property of the `RelativeSourceExtension` class. Therefore, for XAML markup expressions expressed with curly braces, you can eliminate the `Mode=` part of the expression.

For more information about Xamarin.Forms markup extensions, see [XAML Markup Extensions](#).

Bind to self

The `Self` relative binding mode is used bind a property of an element to another property on the same element:

```
<BoxView Color="Red"
         WidthRequest="200"
         HeightRequest="{Binding Source={RelativeSource Self}, Path=WidthRequest}"
         HorizontalOptions="Center" />
```

In this example, the `BoxView` sets its `WidthRequest` property to a fixed size, and the `HeightRequest` property

binds to the `WidthRequest` property. Therefore, both properties are equal and so a square is drawn:



IMPORTANT

When binding a property of an element to another property on the same element, the properties must be the same type. Alternatively, you can specify a converter on the binding to convert the value.

A common use of this binding mode is set an object's `BindingContext` to a property on itself. The following code shows an example of this:

```
<ContentPage ...>
    <BindingContext>="{Binding Source={RelativeSource Self}, Path=DefaultViewModel}">
    <StackLayout>
        <ListView ItemsSource="{Binding Employees}">
            ...
        </ListView>
    </StackLayout>
</ContentPage>
```

In this example, the `BindingContext` of the page is set to the `DefaultViewModel` property of itself. This property is defined in the code-behind file for the page, and provides a viewmodel instance. The `ListView` binds to the `Employees` property of the.viewmodel.

Bind to an ancestor

The `FindAncestor` and `FindAncestorBindingContext` relative binding modes are used to bind to parent elements, of a certain type, in the visual tree. The `FindAncestor` mode is used to bind to a parent element, which derives from the `Element` type. The `FindAncestorBindingContext` mode is used to bind to the `BindingContext` of a parent element.

WARNING

The `AncestorType` property must be set to a `Type` when using the `FindAncestor` and `FindAncestorBindingContext` relative binding modes, otherwise a `XamlParseException` is thrown.

If the `Mode` property isn't explicitly set, setting the `AncestorType` property to a type that derives from `Element` will implicitly set the `Mode` property to `FindAncestor`. Similarly, setting the `AncestorType` property to a type that does not derive from `Element` will implicitly set the `Mode` property to `FindAncestorBindingContext`.

NOTE

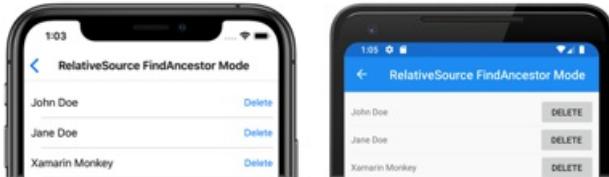
Relative bindings that use the `FindAncestorBindingContext` mode will be reapplied when the `BindingContext` of any ancestors change.

The following XAML shows an example where the `Mode` property will be implicitly set to

FindAncestorBindingContext :

```
<ContentPage ...>
    <StackLayout>
        <ListView ItemsSource="{Binding Employees}">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <StackLayout Orientation="Horizontal">
                            <Label Text="{Binding Fullname}" VerticalOptions="Center" />
                            <Button Text="Delete" Command="{Binding Source={RelativeSource AncestorType={x:Type local:PeopleViewModel}}, Path=DeleteEmployeeCommand}" CommandParameter="{Binding}" HorizontalOptions="EndAndExpand" />
                        </StackLayout>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

In this example, the `BindingContext` of the page is set to the `DefaultViewModel` property of itself. This property is defined in the code-behind file for the page, and provides a viewmodel instance. The `ListView` binds to the `Employees` property of the viewmodel. The `DataTemplate`, which defines the appearance of each item in the `ListView`, contains a `Button`. The button's `Command` property is bound to the `DeleteEmployeeCommand` in its parent's viewmodel. Tapping a `Button` deletes an employee:



In addition, the optional `AncestorLevel` property can help disambiguate ancestor lookup in scenarios where there is possibly more than one ancestor of that type in the visual tree:

```
<Label Text="{Binding Source={RelativeSource AncestorType={x:Type Entry}}, AncestorLevel=2}, Path=Text}" />
```

In this example, the `Label.Text` property binds to the `Text` property of the second `Entry` that's encountered on the upward path, starting at the target element of the binding.

NOTE

The `AncestorLevel` property should be set to 1 to find the ancestor nearest to the binding target element.

Bind to a templated parent

The `TemplatedParent` relative binding mode is used to bind from within a control template to the runtime object instance to which the template is applied (known as the templated parent). This mode is only applicable if the relative binding is within a control template, and is similar to setting a `TemplateBinding`.

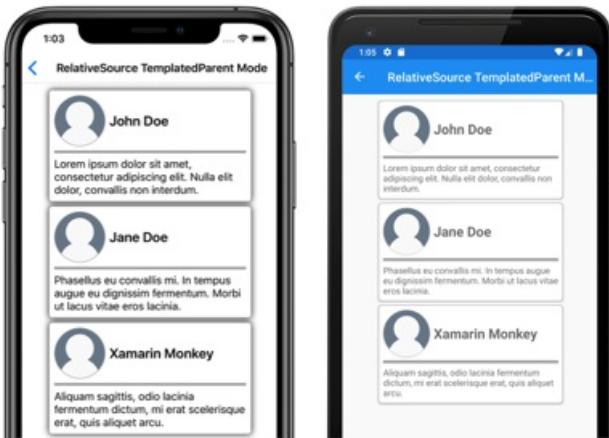
The following XAML shows an example of the `TemplatedParent` relative binding mode:

```

<ContentPage ...>
    <ContentPage.Resources>
        <ControlTemplate x:Key="CardViewControlTemplate">
            <Frame BindingContext="{Binding Source={RelativeSource TemplatedParent}}"
                BackgroundColor="{Binding CardColor}"
                BorderColor="{Binding BorderColor}"
                ...
            <Grid>
                ...
                <Label Text="{Binding CardTitle}"
                    ...
                <BoxView BackgroundColor="{Binding BorderColor}"
                    ...
                <Label Text="{Binding CardDescription}"
                    ...
                </Grid>
            </Frame>
        </ControlTemplate>
    </ContentPage.Resources>
    <StackLayout>
        <controls:CardView BorderColor="DarkGray"
            CardTitle="John Doe"
            CardDescription="Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla
            elit dolor, convallis non interdum."
            IconBackgroundColor="SlateGray"
            IconImageSource="user.png"
            ControlTemplate="{StaticResource CardViewControlTemplate}" />
        <controls:CardView BorderColor="DarkGray"
            CardTitle="Jane Doe"
            CardDescription="Phasellus eu convallis mi. In tempus augue eu dignissim
            fermentum. Morbi ut lacus vitae eros lacinia."
            IconBackgroundColor="SlateGray"
            IconImageSource="user.png"
            ControlTemplate="{StaticResource CardViewControlTemplate}" />
        <controls:CardView BorderColor="DarkGray"
            CardTitle="Xamarin Monkey"
            CardDescription="Aliquam sagittis, odio lacinia fermentum dictum, mi erat
            scelerisque erat, quis aliquet arcu."
            IconBackgroundColor="SlateGray"
            IconImageSource="user.png"
            ControlTemplate="{StaticResource CardViewControlTemplate}" />
    </StackLayout>
</ContentPage>

```

In this example, the `Frame`, which is the root element of the `ControlTemplate`, has its `BindingContext` set to the runtime object instance to which the template is applied. Therefore, the `Frame` and its children resolve their binding expressions against the properties of each `CardView` object:



For more information about control templates, see [Xamarin.Forms Control Templates](#).

Related links

- [Data Binding Demos \(sample\)](#)
- [XAML Markup Extensions](#)
- [Xamarin.Forms Control Templates](#)

Xamarin.Forms Binding Fallbacks

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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, multi-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.

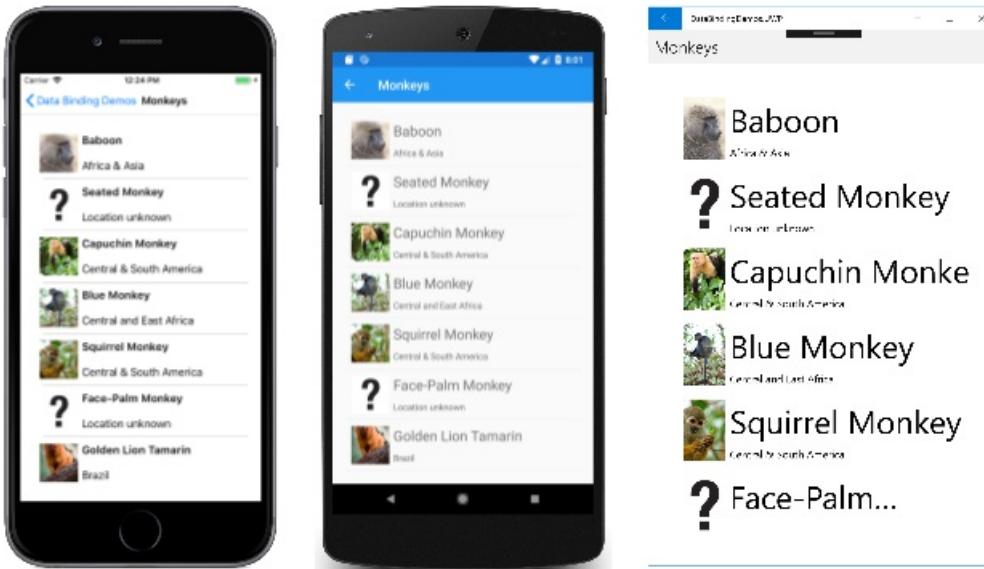
Rather than defining `TargetNullValue` property values inline, it's recommended to define them as resources in a `ResourceDictionary`. The advantage of this approach is that such values are defined once in a single location, and are more easily localizable. The resources can then be retrieved using the `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 \(sample\)](#)

Xamarin.Forms Multi-Bindings

8/4/2022 • 8 minutes to read • [Edit Online](#)

 [Download the sample](#)

Multi-bindings provide the ability to attach a collection of `Binding` objects to a single binding target property. They are created with the `MultiBinding` class, which evaluates all of its `Binding` objects, and returns a single value through a `IMultiValueConverter` instance provided by your application. In addition, `MultiBinding` reevaluates all of its `Binding` objects when any of the bound data changes.

The `MultiBinding` class defines the following properties:

- `Bindings`, of type `IList<BindingBase>`, which represents the collection of `Binding` objects within the `MultiBinding` instance.
- `Converter`, of type `IMultiValueConverter`, which represents the converter to use to convert the source values to or from the target value.
- `ConverterParameter`, of type `object`, which represents an optional parameter to pass to the `converter`.

The `Bindings` property is the content property of the `MultiBinding` class, and therefore does not need to be explicitly set from XAML.

In addition, the `MultiBinding` class inherits the following properties from the `BindingBase` class:

- `FallbackValue`, of type `object`, which represents the value to use when the multi-binding is unable to return a value.
- `Mode`, of type `BindingMode`, which indicates the direction of the data flow of the multi-binding.
- `StringFormat`, of type `string`, which specifies how to format the multi-binding result if it's displayed as a string.
- `TargetNullValue`, of type `object`, which represents the value that is used in the target when the value of the source is `null`.

A `MultiBinding` must use a `IMultiValueConverter` to produce a value for the binding target, based on the value of the bindings in the `Bindings` collection. For example, a `Color` might be computed from red, blue, and green values, which can be values from the same or different binding source objects. When a value moves from the target to the sources, the target property value is translated to a set of values that are fed back into the bindings.

IMPORTANT

Individual bindings in the `Bindings` collection can have their own value converters.

The value of the `Mode` property determines the functionality of the `MultiBinding`, and is used as the binding mode for all the bindings in the collection unless an individual binding overrides the property. For example, if the `Mode` property on a `MultiBinding` object is set to `TwoWay`, then all the bindings in the collection are considered `TwoWay` unless you explicitly set a different `Mode` value on one of the bindings.

Define a `IMultiValueConverter`

The `IMultiValueConverter` interface enables custom logic to be applied to a `MultiBinding`. To associate a converter with a `MultiBinding`, create a class that implements the `IMultiValueConverter` interface, and then

implement the `Convert` and `ConvertBack` methods:

```
public class AllTrueMultiConverter : IMultiValueConverter
{
    public object Convert(object[] values, Type targetType, object parameter, CultureInfo culture)
    {
        if (values == null || !targetType.IsAssignableFrom(typeof(bool)))
        {
            return false;
            // Alternatively, return BindableProperty.UnsetValue to use the binding FallbackValue
        }

        foreach (var value in values)
        {
            if (!(value is bool b))
            {
                return false;
                // Alternatively, return BindableProperty.UnsetValue to use the binding FallbackValue
            }
            else if (!b)
            {
                return false;
            }
        }
        return true;
    }

    public object[] ConvertBack(object value, Type[] targetTypes, object parameter, CultureInfo culture)
    {
        if (!(value is bool b) || targetTypes.Any(t => !t.IsAssignableFrom(typeof(bool))))
        {
            // Return null to indicate conversion back is not possible
            return null;
        }

        if (b)
        {
            return targetTypes.Select(t => (object)true).ToArray();
        }
        else
        {
            // Can't convert back from false because of ambiguity
            return null;
        }
    }
}
```

The `Convert` method converts source values to a value for the binding target. Xamarin.Forms calls this method when it propagates values from source bindings to the binding target. This method accepts four arguments:

- `values`, of type `object[]`, is an array of values that the source bindings in the `MultiBinding` produces.
- `targetType`, of type `Type`, is the type of the binding target property.
- `parameter`, of type `object`, is the converter parameter to use.
- `culture`, of type `CultureInfo`, is the culture to use in the converter.

The `Convert` method returns an `object` that represents a converted value. This method should return:

- `BindableProperty.UnsetValue` to indicate that the converter did not produce a value, and that the binding will use the `FallbackValue`.
- `Binding.DoNothing` to instruct Xamarin.Forms not to perform any action. For example, to instruct Xamarin.Forms not to transfer a value to the binding target, or not to use the `FallbackValue`.
- `null` to indicate that the converter cannot perform the conversion, and that the binding will use the

`TargetNullValue`.

IMPORTANT

A `MultiBinding` that receives `BindableProperty.UnsetValue` from a `Convert` method must define its `FallbackValue` property. Similarly, a `MultiBinding` that receives `null` from a `Convert` method must define its `TargetNullValue` property.

The `ConvertBack` method converts a binding target to the source binding values. This method accepts four arguments:

- `value`, of type `object`, is the value that the binding target produces.
- `targetTypes`, of type `Type[]`, is the array of types to convert to. The array length indicates the number and types of values that are suggested for the method to return.
- `parameter`, of type `object`, is the converter parameter to use.
- `culture`, of type `CultureInfo`, is the culture to use in the converter.

The `ConvertBack` method returns an array of values, of type `object[]`, that have been converted from the target value back to the source values. This method should return:

- `BindableProperty.UnsetValue` at position `i` to indicate that the converter is unable to provide a value for the source binding at index `i`, and that no value is to be set on it.
- `Binding.DoNothing` at position `i` to indicate that no value is to be set on the source binding at index `i`.
- `null` to indicate that the converter cannot perform the conversion or that it does not support conversion in this direction.

Consume a `IMultiValueConverter`

A `IMultiValueConverter` is consumed by instantiating it in a resource dictionary, and then referencing it using the `StaticResource` markup extension to set the `MultiBinding.Converter` property:

```
<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.MultiBindingConverterPage"
    Title="MultiBinding Converter demo">

    <ContentPage.Resources>
        <local:AllTrueMultiConverter x:Key="AllTrueConverter" />
        <local:InverterConverter x:Key="InverterConverter" />
    </ContentPage.Resources>

    <CheckBox>
        <CheckBox.IsChecked>
            <MultiBinding Converter="{StaticResource AllTrueConverter}">
                <Binding Path="Employee.IsOver16" />
                <Binding Path="Employee.HasPassedTest" />
                <Binding Path="Employee.IsSuspended" />
                <Binding Path="Employee.IsSuspended" Converter="{StaticResource InverterConverter}" />
            </MultiBinding>
        </CheckBox.IsChecked>
    </CheckBox>
</ContentPage>
```

In this example, the `MultiBinding` object uses the `AllTrueMultiConverter` instance to set the `CheckBox.IsChecked` property to `true`, provided that the three `Binding` objects evaluate to `true`. Otherwise, the `CheckBox.IsChecked` property is set to `false`.

By default, the `CheckBox.IsChecked` property uses a `TwoWay` binding. Therefore, the `ConvertBack` method of the `AllTrueMultiConverter` instance is executed when the `CheckBox` is unchecked by the user, which sets the source binding values to the value of the `CheckBox.IsChecked` property.

The equivalent C# code is shown below:

```
public class MultiBindingConverterCodePage : ContentPage
{
    public MultiBindingConverterCodePage()
    {
        BindingContext = new GroupViewModel();

        CheckBox checkBox = new CheckBox();
        checkBox.SetBinding(CheckBox.IsCheckedProperty, new MultiBinding
        {
            Bindings = new Collection<BindingBase>
            {
                new Binding("Employee1.IsOver16"),
                new Binding("Employee1.HasPassedTest"),
                new Binding("Employee1.IsSuspended", converter: new InverterConverter())
            },
            Converter = new AllTrueMultiConverter()
        });

        Title = "MultiBinding converter demo";
        Content = checkBox;
    }
}
```

Format strings

A `MultiBinding` can format any multi-binding result that's displayed as a string, with the `StringFormat` property. This property can be set to a standard .NET formatting string, with placeholders, that specifies how to format the multi-binding result:

```
<Label>
    <Label.Text>
        <MultiBinding StringFormat="{}{0} {1} {2}">
            <Binding Path="Employee1.Forename" />
            <Binding Path="Employee1.MiddleName" />
            <Binding Path="Employee1.Surname" />
        </MultiBinding>
    </Label.Text>
</Label>
```

In this example, the `StringFormat` property combines the three bound values into a single string that's displayed by the `Label`.

The equivalent C# code is shown below:

```
Label label = new Label();
label.SetBinding(Label.TextProperty, new MultiBinding
{
    Bindings = new Collection<BindingBase>
    {
        new Binding("Employee1.Forename"),
        new Binding("Employee1.MiddleName"),
        new Binding("Employee1.Surname")
    },
    StringFormat = "{0} {1} {2}"
});
```

IMPORTANT

The number of parameters in a composite string format can't exceed the number of child `Binding` objects in the `MultiBinding`.

When setting the `Converter` and `StringFormat` properties, the converter is applied to the data value first, and then the `StringFormat` is applied.

For more information about string formatting in Xamarin.Forms, see [Xamarin.Forms String Formatting](#).

Provide fallback values

Data bindings can be made more robust by defining fallback values to use if the binding process fails. This can be accomplished by optionally defining the `FallbackValue` and `TargetNullValue` properties on a `MultiBinding` object.

A `MultiBinding` will use its `FallbackValue` when the `Convert` method of an `IMultiValueConverter` instance returns `BindableProperty.UnsetValue`, which indicates that the converter did not produce a value. A `MultiBinding` will use its `TargetNullValue` when the `Convert` method of an `IMultiValueConverter` instance returns `null`, which indicates that the converter cannot perform the conversion.

For more information about binding fallbacks, see [Xamarin.Forms Binding Fallbacks](#).

Nest MultiBinding objects

`MultiBinding` objects can be nested so that multiple `MultiBinding` objects are evaluated to return a value through an `IMultiValueConverter` instance:

```
<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.NestedMultiBindingPage"
    Title="Nested MultiBinding demo">

    <ContentPage.Resources>
        <local:AllTrueMultiConverter x:Key="AllTrueConverter" />
        <local:AnyTrueMultiConverter x:Key="AnyTrueConverter" />
        <local:InverterConverter x:Key="InverterConverter" />
    </ContentPage.Resources>

    <CheckBox>
        <CheckBox.IsChecked>
            <MultiBinding Converter="{StaticResource AnyTrueConverter}">
                <MultiBinding Converter="{StaticResource AllTrueConverter}">
                    <Binding Path="Employee.IsOver16" />
                    <Binding Path="Employee.HasPassedTest" />
                    <Binding Path="Employee.IsSuspended" Converter="{StaticResource InverterConverter}" />
                </MultiBinding>
                <Binding Path="Employee.IsMonarch" />
            </MultiBinding>
        </CheckBox.IsChecked>
    </CheckBox>
</ContentPage>
```



In this example, the `MultiBinding` object uses its `AnyTrueMultiConverter` instance to set the `CheckBox.IsChecked` property to `true`, provided that all of the `Binding` objects in the inner `MultiBinding` object evaluate to `true`, or provided that the `Binding` object in the outer `MultiBinding` object evaluates to `true`. Otherwise, the `CheckBox.IsChecked` property is set to `false`.

Use a `RelativeSource` binding in a `MultiBinding`

`MultiBinding` objects support relative bindings, which provide the ability to set the binding source relative to the position of the binding target:

```

<ContentPage ...>
    xmlns:local="clr-namespace:DataBindingDemos"
    xmlns:xct="clr-namespace:Xamarin.CommunityToolkit.UI.Views;assembly=Xamarin.CommunityToolkit">
    <ContentPage.Resources>
        <local:AllTrueMultiConverter x:Key="AllTrueConverter" />

        <ControlTemplate x:Key="CardViewExpanderControlTemplate">
            <xct:Expander BindingContext="{Binding Source={RelativeSource TemplatedParent}}"
                IsExpanded="{Binding IsExpanded, Source={RelativeSource TemplatedParent}}"
                BackgroundColor="{Binding CardColor}">
                <xct:Expander.IsVisible>
                    <MultiBinding Converter="{StaticResource AllTrueConverter}">
                        <Binding Path="IsExpanded" />
                        <Binding Path=".IsEnabled" />
                    </MultiBinding>
                </xct:Expander.IsVisible>
                <xct:Expander.Header>
                    <Grid>
                        <!-- XAML that defines Expander header goes here -->
                    </Grid>
                </xct:Expander.Header>
                <Grid>
                    <!-- XAML that defines Expander content goes here -->
                </Grid>
            </xct:Expander>
        </ControlTemplate>
    </ContentPage.Resources>

    <StackLayout>
        <controls:CardViewExpander BorderColor="DarkGray"
            CardTitle="John Doe"
            CardDescription="Lorem ipsum dolor sit amet, consectetur adipiscing elit.
            Nulla elit dolor, convallis non interdum."
            IconBackgroundColor="SlateGray"
            IconImageSource="user.png"
            ControlTemplate="{StaticResource CardViewExpanderControlTemplate}"
            IsEnabled="True"
            IsExpanded="True" />
    </StackLayout>
</ContentPage>

```

NOTE

The `Expander` control is now part of the Xamarin Community Toolkit.

In this example, the `TemplatedParent` relative binding mode is used to bind from within a control template to the runtime object instance to which the template is applied. The `Expander`, which is the root element of the `ControlTemplate`, has its `BindingContext` set to the runtime object instance to which the template is applied. Therefore, the `Expander` and its children resolve their binding expressions, and `Binding` objects, against the properties of the `CardViewExpander` object. The `MultiBinding` uses the `AllTrueMultiConverter` instance to set the `Expander.IsVisible` property to `true` provided that the two `Binding` objects evaluate to `true`. Otherwise, the `Expander.IsVisible` property is set to `false`.

For more information about relative bindings, see [Xamarin.Forms Relative Bindings](#). For more information about control templates, see [Xamarin.Forms Control Templates](#).

Related links

- [Data Binding Demos \(sample\)](#)
- [Xamarin.Forms String Formatting](#)

- [Xamarin.Forms Binding Fallbacks](#)
- [Xamarin.Forms Relative Bindings](#)
- [Xamarin.Forms Control Templates](#)

The Xamarin.Forms Command Interface

8/4/2022 • 17 minutes to read • [Edit Online](#)



[Download the sample](#)

In the Model-View-ViewModel (MVVM) architecture, data bindings are defined between properties in the ViewModel, which is generally a class that derives from `IPropertyChanged`, 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 + " " + 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" />

```

```

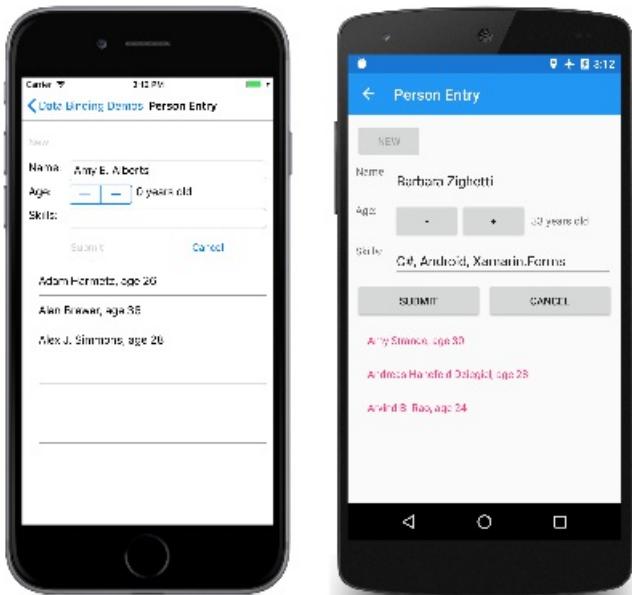
</Grid>

<!-- 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 screen shows 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;
                RefreshCanExecutes();
            },
            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=".&#xA0B7;"*
        Grid.Row="5" Grid.Column="2"
        Command="{Binding DigitCommand}"
        CommandParameter="." />

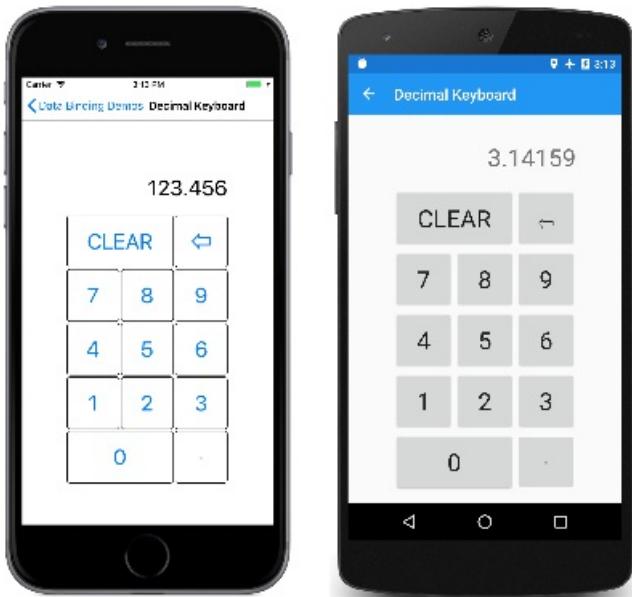
```

</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);
        }
        RefreshCanExecute();
    },
        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 `RefreshCanExecute`, 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.

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

8/4/2022 • 7 minutes to read • [Edit Online](#)



[Download the sample](#)

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

It's recommended to set the `x:DataType` attribute at the same level in the view hierarchy as the `BindingContext` is set. However, this attribute can be re-defined at any location in a view hierarchy.

To use compiled bindings, the `x:DataType` attribute must be set to a string literal, or a type using the `x:Type` markup extension. 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.

Use 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>

```

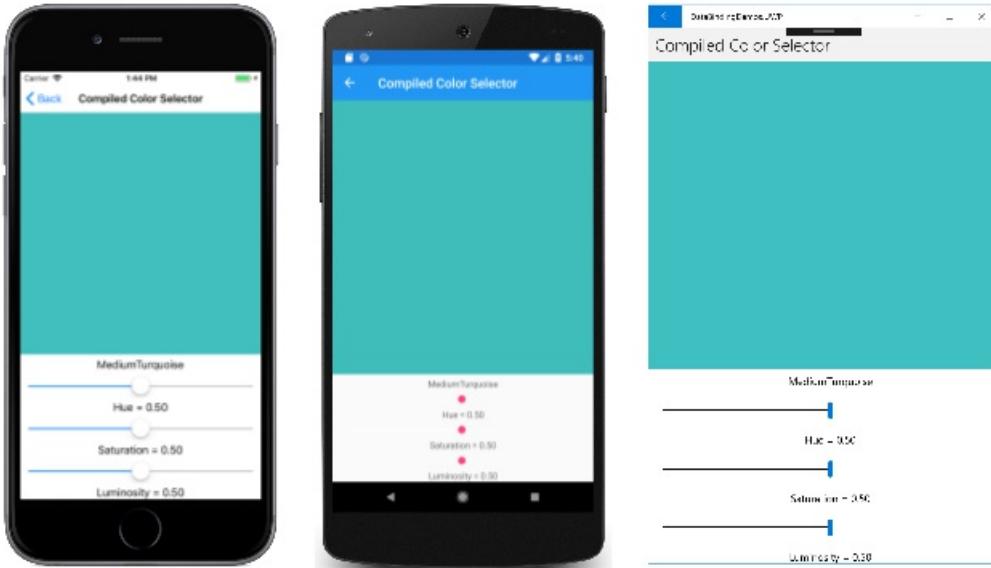
The root `StackLayout` instantiates the `HslColorViewModel` and initializes the `Color` property within property element tags for the `BindingContext` property. This root `StackLayout` also defines the `x:DataType` attribute as the viewmodel type, indicating that any binding expressions in the root `StackLayout` 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. While this example sets the `x:DataType` attribute to a string literal, it can also be set to a type with the `x:Type` markup extension. For more information about the `x:Type` markup extension, see [x:Type Markup Extension](#).

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](#).

Use 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">

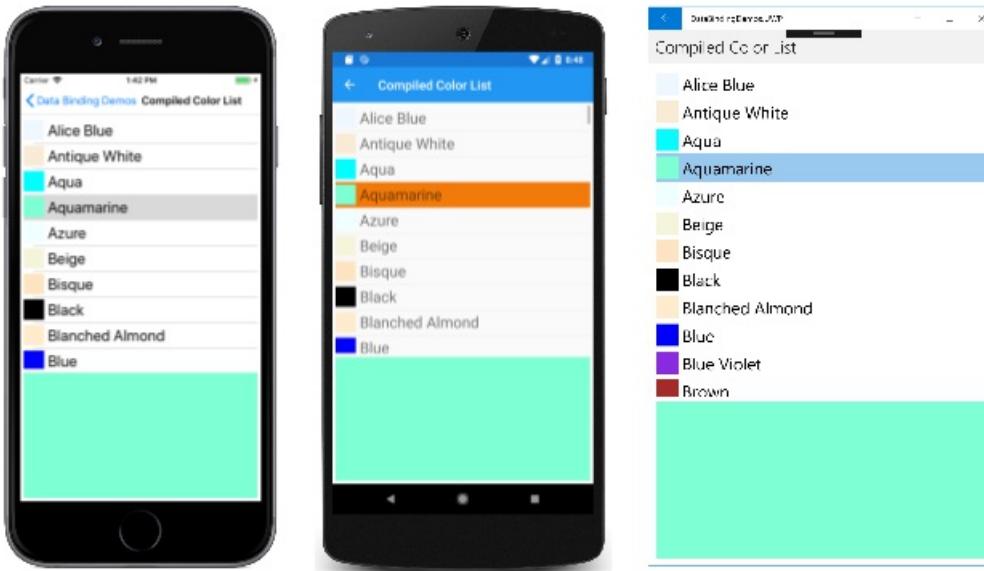
<Grid>
    ...
    <ListView x:Name="colorListView"
        ItemsSource="{x:Static local:NamedColor.All}"
        ... >
        <ListView.ItemTemplate>
            <DataTemplate x:DataType="local:NamedColor">
                <ViewCell>
                    <StackLayout Orientation="Horizontal">
                        <BoxView Color="{Binding Color}" ... />
                        <Label Text="{Binding FriendlyName}" ... />
                    </StackLayout>
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
    <!-- The BoxView doesn't use compiled bindings -->
    <BoxView Color="{Binding Source={x:Reference colorListView}, Path=SelectedItem.Color}" ... />
</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. While this example sets the `x:DataType` attribute to a string literal, it can also be set to a type with the `x>Type` markup extension. For more information about the `x>Type` markup extension, see [xType Markup Extension](#).

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`.

Combine 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

8/4/2022 • 2 minutes to read • [Edit Online](#)

Introduction

The `DependencyService` class is a service locator that enables Xamarin.Forms applications to invoke native platform functionality from shared code.

Registration and Resolution

Platform implementations must be registered with the `DependencyService`, and then resolved from shared code to invoke them.

Picking a Photo from the Library

This article explains how to use the Xamarin.Forms `DependencyService` class to pick a photo from the phone's picture library.

Xamarin.Forms DependencyService Introduction

8/4/2022 • 2 minutes to read • [Edit Online](#)

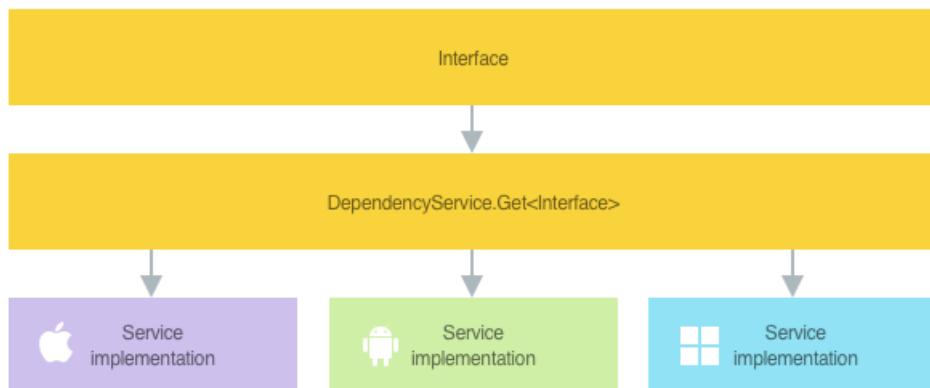
 [Download the sample](#)

The `DependencyService` class is a service locator that enables Xamarin.Forms applications to invoke native platform functionality from shared code.

The process for using the `DependencyService` to invoke native platform functionality is to:

1. Create an interface for the native platform functionality, in shared code. For more information, see [Create an interface](#).
2. Implement the interface in the required platform projects. For more information, see [Implement the interface on each platform](#).
3. Register the platform implementations with the `DependencyService`. This enables Xamarin.Forms to locate the platform implementations at runtime. For more information, see [Register the platform implementations](#).
4. Resolve the platform implementations from shared code, and invoke them. For more information, see [Resolve the platform implementations](#).

The following diagram shows how native platform functionality is invoked in a Xamarin.Forms application:



Create an interface

The first step in being able to invoke native platform functionality from shared code, is to create an interface that defines the API for interacting with the native platform functionality. This interface should be placed in your shared code project.

The following example shows an interface for an API that can be used to retrieve the orientation of a device:

```
public interface IDeviceOrientationService
{
    DeviceOrientation GetOrientation();
}
```

Implement the interface on each platform

After creating the interface that defines the API for interacting with the native platform functionality, the interface must be implemented in each platform project.

iOS

The following code example shows the implementation of the `IDeviceOrientationService` interface on iOS:

```
namespace DependencyServiceDemos.iOS
{
    public class DeviceOrientationService : IDeviceOrientationService
    {
        public DeviceOrientation GetOrientation()
        {
            UIInterfaceOrientation orientation = UIApplication.SharedApplication.StatusBarOrientation;

            bool isPortrait = orientation == UIInterfaceOrientation.Portrait ||
                orientation == UIInterfaceOrientation.PortraitUpsideDown;
            return isPortrait ? DeviceOrientation.Por
```

Android

The following code example shows the implementation of the `IDeviceOrientationService` interface on Android:

```
namespace DependencyServiceDemos.Droid
{
    public class DeviceOrientationService : IDeviceOrientationService
    {
        public DeviceOrientation GetOrientation()
        {
            IWindowManager windowManager =
Android.App.Application.Context.GetSystemService(Context.WindowService).JavaCast<IWindowManager>();

            SurfaceOrientation orientation = windowManager.DefaultDisplay.Rotation;
            bool isLandscape = orientation == SurfaceOrientation.Rotation90 ||
                orientation == SurfaceOrientation.Rotation270;
            return isLandscape ? DeviceOrientation.Landscape : DeviceOrientation.Por
```

Universal Windows Platform

The following code example shows the implementation of the `IDeviceOrientationService` interface on the Universal Windows Platform (UWP):

```
namespace DependencyServiceDemos.UWP
{
    public class DeviceOrientationService : IDeviceOrientationService
    {
        public DeviceOrientation GetOrientation()
        {
            ApplicationViewOrientation orientation = ApplicationView.GetForCurrentView().Orientation;
            return orientation == ApplicationViewOrientation.Landscape ? DeviceOrientation.Landscape :
DeviceOrientation.Por
        }
    }
}
```

Register the platform implementations

After implementing the interface in each platform project, the platform implementations must be registered with the `DependencyService`, so that Xamarin.Forms can locate them at runtime. This is typically performed with

the [DependencyAttribute](#), which indicates that the specified type provides an implementation of the interface.

The following example shows using the [DependencyAttribute](#) to register the iOS implementation of the [IDeviceOrientationService](#) interface:

```
using Xamarin.Forms;

[assembly: Dependency(typeof(DependencyServiceDemos.iOS.DeviceOrientationService))]
namespace DependencyServiceDemos.iOS
{
    public class DeviceOrientationService : IDeviceOrientationService
    {
        public DeviceOrientation GetOrientation()
        {
            ...
        }
    }
}
```

In this example, the [DependencyAttribute](#) registers the [DeviceOrientationService](#) with the [DependencyService](#). Similarly, the implementations of the [IDeviceOrientationService](#) interface on other platforms should be registered with the [DependencyAttribute](#).

For more information about registering platform implementations with the [DependencyService](#), see [Xamarin.Forms DependencyService Registration and Resolution](#).

Resolve the platform implementations

Following registration of platform implementations with the [DependencyService](#), the implementations must be resolved before being invoked. This is typically performed in shared code using the [DependencyService.Get<T>](#) method.

The following code shows an example of calling the [Get<T>](#) method to resolve the [IDeviceOrientationService](#) interface, and then invoking its [GetOrientation](#) method:

```
IDeviceOrientationService service = DependencyService.Get<IDeviceOrientationService>();
DeviceOrientation orientation = service.GetOrientation();
```

Alternatively, this code can be condensed into a single line:

```
DeviceOrientation orientation = DependencyService.Get<IDeviceOrientationService>().GetOrientation();
```

For more information about resolving platform implementations with the [DependencyService](#), see [Xamarin.Forms DependencyService Registration and Resolution](#).

Related links

- [DependencyService Demos \(sample\)](#)
- [Xamarin.Forms DependencyService Registration and Resolution](#)

Xamarin.Forms DependencyService Registration and Resolution

8/4/2022 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

When using the Xamarin.Forms `DependencyService` to invoke native platform functionality, platform implementations must be registered with the `DependencyService`, and then resolved from shared code to invoke them.

Register platform implementations

Platform implementations must be registered with the `DependencyService` so that Xamarin.Forms can locate them at runtime.

Registration can be performed with the `DependencyAttribute`, or with the `Register` and `RegisterSingleton` methods.

IMPORTANT

Release builds of UWP projects that use .NET native compilation should register platform implementations with the `Register` methods.

Registration by attribute

The `DependencyAttribute` can be used to register a platform implementation with the `DependencyService`. The attribute indicates that the specified type provides a concrete implementation of the interface.

The following example uses the `DependencyAttribute` to register the iOS implementation of the `IDeviceOrientationService` interface:

```
using Xamarin.Forms;

[assembly: Dependency(typeof(DeviceOrientationService))]
namespace DependencyServiceDemos.iOS
{
    public class DeviceOrientationService : IDeviceOrientationService
    {
        public DeviceOrientation GetOrientation()
        {
            ...
        }
    }
}
```

In this example, the `DependencyAttribute` registers the `DeviceOrientationService` with the `DependencyService`. This results in the concrete type being registered against the interface it implements.

Similarly, the implementations of the `IDeviceOrientationService` interface on other platforms should be registered with the `DependencyAttribute`.

NOTE

Registration with the `DependencyAttribute` is performed at the namespace level.

Registration by method

The `DependencyService.Register` methods, and the `RegisterSingleton` method, can be used to register a platform implementation with the `DependencyService`.

The following example uses the `Register` method to register the iOS implementation of the `IDeviceOrientationService` interface:

```
[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());
        DependencyService.Register<IDeviceOrientationService, DeviceOrientationService>();
        return base.FinishedLaunching(app, options);
    }
}
```

In this example, the `Register` method registers the concrete type, `DeviceOrientationService`, against the `IDeviceOrientationService` interface. Alternatively, an overload of the `Register` method can be used to register a platform implementation with the `DependencyService`:

```
DependencyService.Register<DeviceOrientationService>();
```

In this example, the `Register` method registers the `DeviceOrientationService` with the `DependencyService`. This results in the concrete type being registered against the interface it implements.

Alternatively, an existing object instance can be registered as a singleton with the `RegisterSingleton` method:

```
var service = new DeviceOrientationService();
DependencyService.RegisterSingleton<IDeviceOrientationService>(service);
```

In this example, the `RegisterSingleton` method registers the `DeviceOrientationService` object instance against the `IDeviceOrientationService` interface, as a singleton.

Similarly, the implementations of the `IDeviceOrientationService` interface on other platforms can be registered with the `Register` methods, or the `RegisterSingleton` method.

IMPORTANT

Registration with the `Register` and `RegisterSingleton` methods must be performed in platform projects, before the functionality provided by the platform implementation is invoked from shared code.

Resolve the platform implementations

Platform implementations must be resolved before being invoked. This is typically performed in shared code using the `DependencyService.Get<T>` method. However, it can also be accomplished with the `DependencyService.Resolve<T>` method.

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](#).

IMPORTANT

Invoking a platform implementation that hasn't been registered with the `DependencyService` will result in a `NullReferenceException` being thrown.

Resolve using the `Get<T>` method

The `Get<T>` method retrieves the platform implementation of interface `T` at runtime, and either:

- Creates an instance of it as a singleton.
- Returns an existing instance as a singleton, that was registered with the `DependencyService` by the `RegisterSingleton` method.

In both cases, the instance will live for the lifetime of the application, and any subsequent calls to resolve the same platform implementation will retrieve the same instance.

The following code shows an example of calling the `Get<T>` method to resolve the `IDeviceOrientationService` interface, and then invoking its `GetOrientation` method:

```
IDeviceOrientationService service = DependencyService.Get<IDeviceOrientationService>();  
DeviceOrientation orientation = service.GetOrientation();
```

Alternatively, this code can be condensed into a single line:

```
DeviceOrientation orientation = DependencyService.Get<IDeviceOrientationService>().GetOrientation();
```

NOTE

The `Get<T>` method returns an instance of the platform implementation of interface `T` as a singleton, by default. However, this behavior can be changed. For more information, see [Manage the lifetime of resolved objects](#).

Resolve using the `Resolve<T>` method

The `Resolve<T>` method retrieves the platform implementation of interface `T` at runtime, using a dependency resolution method that's been injected into Xamarin.Forms with the `DependencyResolver` class. If a dependency resolution method hasn't been injected into Xamarin.Forms, the `Resolve<T>` method will fallback to calling the `Get<T>` method to retrieve the platform implementation. For more information about injecting a dependency resolution method into Xamarin.Forms, see [Dependency resolution in Xamarin.Forms](#).

The following code shows an example of calling the `Resolve<T>` method to resolve the `IDeviceOrientationService` interface, and then invoking its `GetOrientation` method:

```
IDeviceOrientationService service = DependencyService.Resolve<IDeviceOrientationService>();  
DeviceOrientation orientation = service.GetOrientation();
```

Alternatively, this code can be condensed into a single line:

```
DeviceOrientation orientation = DependencyService.Resolve<IDeviceOrientationService>().GetOrientation();
```

NOTE

When the `Resolve<T>` method falls back to calling the `Get<T>` method, it returns an instance of the platform implementation of interface `T` as a singleton, by default. However, this behavior can be changed. For more information, see [Manage the lifetime of resolved objects](#).

Manage the lifetime of resolved objects

The default behavior of the `DependencyService` class is to resolve platform implementations as singletons. Therefore, platform implementations will live for the lifetime of an application.

This behavior is specified with the `DependencyFetchTarget` optional argument on the `Get<T>` and `Resolve<T>` methods. The `DependencyFetchTarget` enumeration defines two members:

- `GlobalInstance`, which returns the platform implementation as a singleton.
- `NewInstance`, which returns a new instance of the platform implementation. The application is then responsible for managing the lifetime of the platform implementation instance.

The `Get<T>` and `Resolve<T>` methods both set their optional arguments to `DependencyFetchTarget.GlobalInstance`, and so platform implementations are always resolved as singletons. This behavior can be changed, so that new instances of platform implementations are created, by specifying `DependencyFetchTarget.NewInstance` as arguments to the `Get<T>` and `Resolve<T>` methods:

```
ITextToSpeechService service = DependencyService.Get<ITextToSpeechService>()
(DependencyFetchTarget.NewInstance);
```

In this example, the `DependencyService` creates a new instance of the platform implementation for the `ITextToSpeechService` interface. Any subsequent calls to resolve the `ITextToSpeechService` will also create new instances.

The consequence of always creating a new instance of a platform implementation is that the application becomes responsible for managing the instances' lifetime. This means that if you subscribe to an event defined in a platform implementation, you should unsubscribe from the event when the platform implementation is no longer required. In addition, it means that it may be necessary for platform implementations to implement `IDisposable`, and cleanup their resources in `Dispose` methods. The sample application demonstrates this scenario in its `TextToSpeechService` platform implementations.

When an application finishes using a platform implementation that implements `IDisposable`, it should call the object's `Dispose` implementation. One way of accomplishing this is with a `using` statement:

```
ITextToSpeechService service = DependencyService.Get<ITextToSpeechService>()
(DependencyFetchTarget.NewInstance);
using (service as IDisposable)
{
    await service.SpeakAsync("Hello world");
}
```

In this example, after the `SpeakAsync` method is invoked, the `using` statement automatically disposes of the platform implementation object. This results in the object's `Dispose` method being invoked, which performs the required cleanup.

For more information about calling an object's `Dispose` method, see [Using objects that implement IDisposable](#).

Related links

- [DependencyService Demos \(sample\)](#)
- [Dependency resolution in Xamarin.Forms](#)

Picking a Photo from the Picture Library

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

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 [IPhotoPickerService](#) interface in the .NET Standard library of the sample code:

```
namespace DependencyServiceDemos
{
    public interface IPhotoPickerService
    {
        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 [IPhotoPickerService](#) 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 [PhotoPickerController](#) 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 [IPhotoPickerService](#) interface:

```

[assembly: Dependency (typeof (PhotoPickerController))]
namespace DependencyServiceDemos.iOS
{
    public class PhotoPickerController : IPhotoPickerController
    {
        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 += OnImagePickerControllerFinishedPickingMedia;
            imagePicker.Canceled += OnImagePickerControllerCancelled;

            // Present UIImagePickerController;
            UIWindow window = UIApplication.SharedApplication.KeyWindow;
            var viewController = window.RootViewController;
            viewController.PresentViewController(imagePicker, true, null);

            // 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 `PresentViewController` method 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 an in memory PNG or JPEG file 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 DependencyServiceDemos.iOS
{
    public class PhotoPickerController : IPhotoPickerController
    {
        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;
                if (args.ReferenceUrl.PathExtension.Equals("PNG") ||
                    args.ReferenceUrl.PathExtension.Equals("png"))
                {
                    data = image.AsPNG();
                }
                else
                {
                    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.DismissModalViewControllerAnimated(true);
    }

    void OnImagePickerCancelled(object sender, EventArgs args)
    {
        UnregisterEventHandlers();
        taskCompletionSource.SetResult(null);
        imagePicker.DismissModalViewControllerAnimated(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
{
    internal static MainActivity Instance { get; private set; }

    protected override void OnCreate(Bundle savedInstanceState)
    {
        // ...
        Instance = this;
    }
    // ...
    // 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 `PhotoPickerService` class in the Android project. This is the class with the `GetImageStreamAsync` method:

```
[assembly: Dependency(typeof(PhotoPickerService))]
namespace DependencyServiceDemos.Droid
{
    public class PhotoPickerService : IPhotoPickerService
    {
        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 `PhotoPickerService` 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(PhotoPickerService))]
namespace DependencyServiceDemos.UWP
{
    public class PhotoPickerService : IPhotoPickerService
    {
        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 shared code in the .NET Standard library can take advantage of it.

The UI includes a `Button` that can be clicked to choose a photo:

```
<Button Text="Pick Photo"
       Clicked="OnPickPhotoButtonClicked" />
```

The `Clicked` event handler uses the `DependencyService` class to call `GetImageStreamAsync`. This results in a call to the platform project. If the method returns a `Stream` object, then the handler sets the `Source` property of the `image` object to the `Stream` data:

```
async void OnPickPhotoButtonClicked(object sender, EventArgs e)
{
    (sender as Button).IsEnabled = false;

    Stream stream = await DependencyService.Get<IPhotoPickerService>().GetImageStreamAsync();
    if (stream != null)
    {
        image.Source = ImageSource.FromStream(() => stream);
    }

    (sender as Button).IsEnabled = true;
}
```

Related links

- [DependencyService \(sample\)](#)
- [Choose a Photo from the Gallery \(iOS\)](#)
- [Select an Image \(Android\)](#)

Xamarin.Forms dual-screen

8/4/2022 • 2 minutes to read • [Edit Online](#)

Dual-screen devices like the Microsoft Surface Duo facilitate new user-experience possibilities for your applications. Xamarin.Forms includes `TwoPaneView` and `DualScreenInfo` classes so you can develop apps for dual-screen devices.

Get started

Follow these steps to add dual-screen capabilities to a Xamarin.Forms app:

1. Open the **NuGet Package Manager** dialog for your solution.
2. Under the **Browse** tab, search for `Xamarin.Forms.DualScreen`.
3. Install the `Xamarin.Forms.DualScreen` package to your solution.
4. Add the following initialization method call to the Android project's `MainActivity` class, in the `OnCreate` event:

```
Xamarin.Forms.DualScreen.DualScreenService.Init(this);
```

This method is required for the app to be able to detect changes in the app's state, such as being spanned across two screens.

5. Update the `Activity` attribute on the Android project's `MainActivity` class, so that it includes *all* these `ConfigurationChanges` options:

```
ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation  
| ConfigChanges.ScreenLayout | ConfigChanges.SmallestScreenSize | ConfigChanges.UiMode
```

These values are required so that configuration changes and span state can be more reliably reported. By default only two are added to Xamarin.Forms projects, so remember to add the rest for reliable dual-screen support.

Troubleshooting

If the `DualScreenInfo` class or `TwoPaneView` layout aren't working as expected, double-check the set-up instructions on this page. Omitting or misconfiguring the `Init` method or the `ConfigurationChanges` attribute values are common causes of errors.

Review the [Xamarin.Forms dual-screen samples](#) for additional guidance and reference implementation.

Next steps

Once you've added the NuGet, add dual-screen features to your app with the following guidance:

- [Dual-screen design patterns](#) - When considering how to best utilize multiple screens on a dual-screen device, refer to this pattern guidance to find the best fit for your application interface.
- [TwoPaneView layout](#) - The Xamarin.Forms `TwoPaneView` class, inspired by the UWP control of the same name, is a cross-platform layout optimized for dual-screen devices.

- **DualScreenInfo helper class** - The `DualScreenInfo` class enables you to determine which pane your view is on, how big it is, what posture the device is in, the angle of the hinge, and more.
- **Dual-screen triggers** - The `Xamarin.Forms.DualScreen` namespace includes two state triggers that trigger a `VisualState` change when the view mode of the attached layout, or window, changes.

Visit the [dual-screen developer docs](#) for more information.

Xamarin.Forms dual-screen design patterns

8/4/2022 • 2 minutes to read • [Edit Online](#)

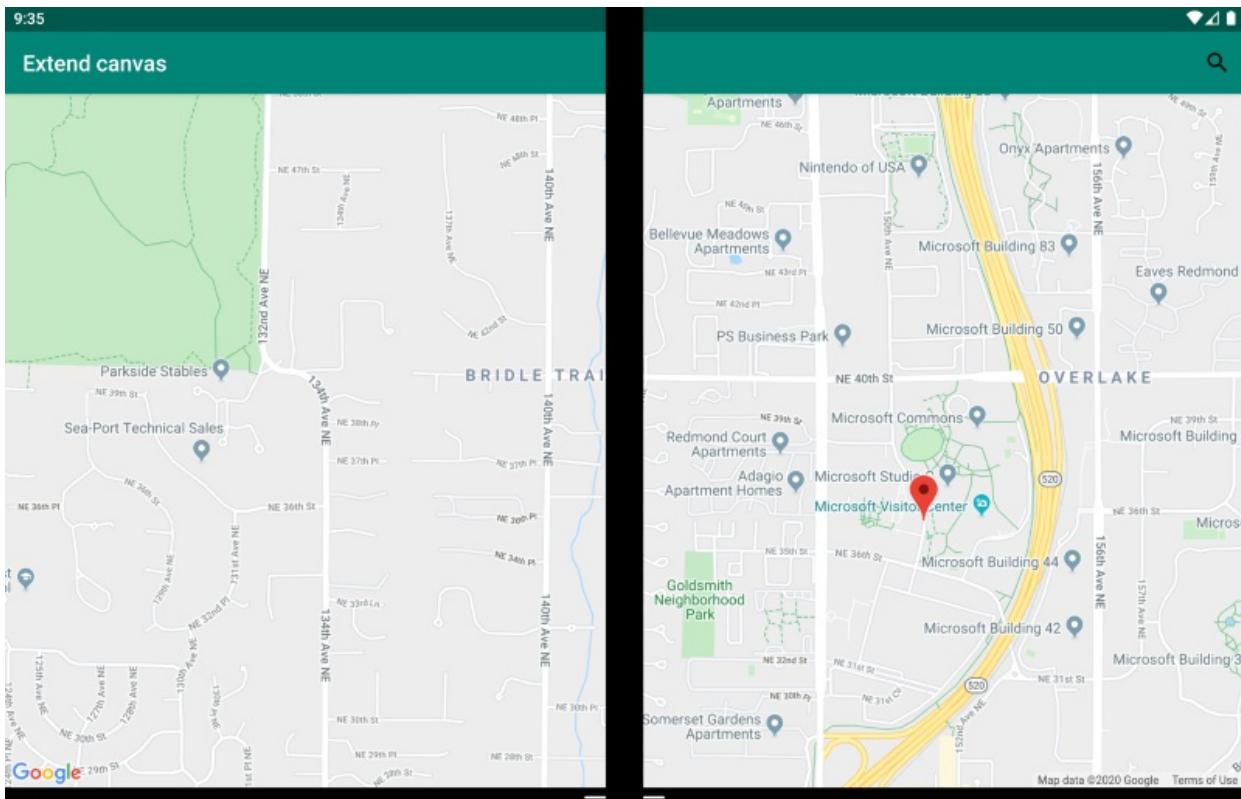


[Download the sample](#)

This guide introduces our recommended design patterns for dual-screen devices with code and samples to assist you in creating interfaces that provide engaging and useful user experiences.

Extended canvas pattern

The extended canvas pattern treats both screens as one large canvas for displaying a map, image, spreadsheet, or other such content that benefits from spreading to consume the maximum space:

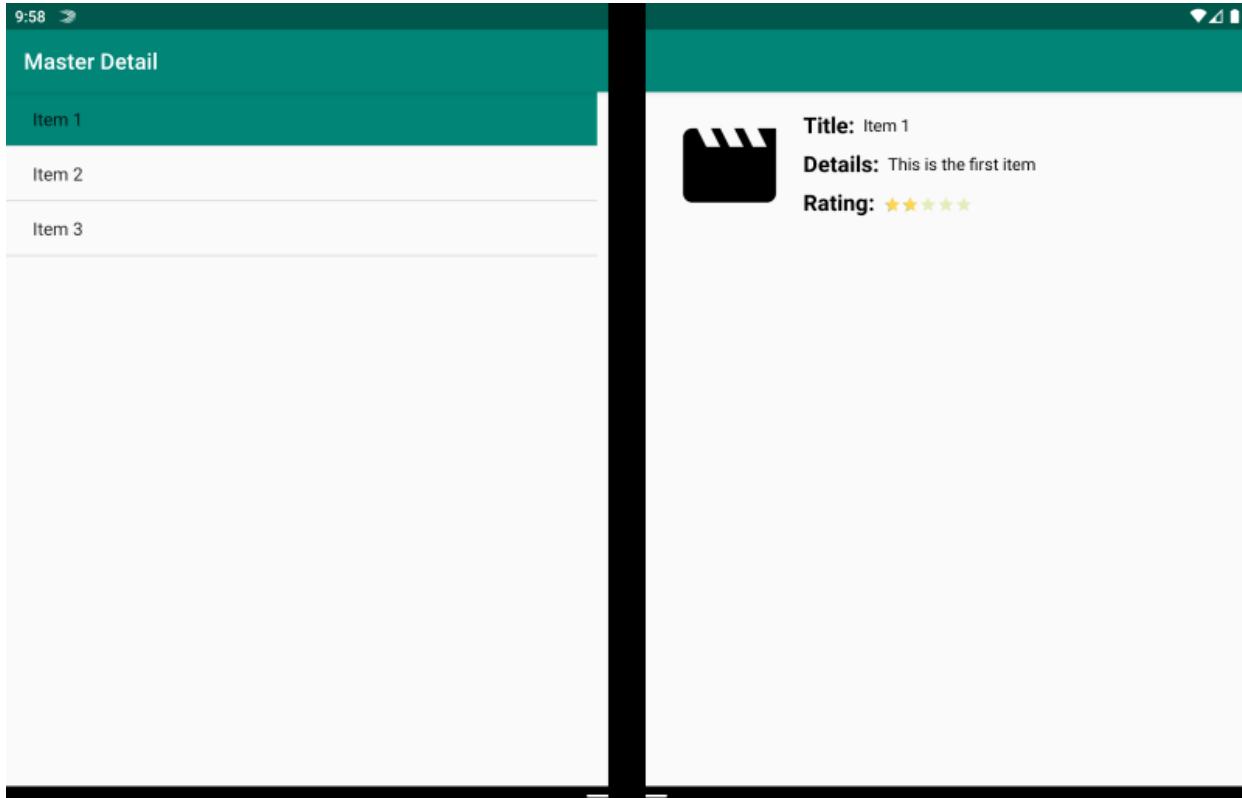


```
<ContentPage xmlns:local="clr-namespace:Xamarin.Duo.Forms.Samples"
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:d="http://xamarin.com/schemas/2014/forms/design"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    x:Class="Xamarin.Duo.Forms.Samples.ExtendCanvas">
    <Grid>
        <WebView x:Name="webView"
            HorizontalOptions="FillAndExpand"
            VerticalOptions="FillAndExpand" />
        <SearchBar x:Name="searchBar"
            Placeholder="Find a place..." 
            BackgroundColor="DarkGray"
            Opacity="0.8"
            HorizontalOptions="FillAndExpand"
            VerticalOptions="Start" />
    </Grid>
</ContentPage>
```

In this example, the `Grid` and inner content will expand to consume all of the screen available, whether displayed on a single screen, or spanned across two screens.

Master-detail pattern

The master-detail pattern is for when the master view, typically a list on the left, provides content from which a user selects to view details about that item on the right:

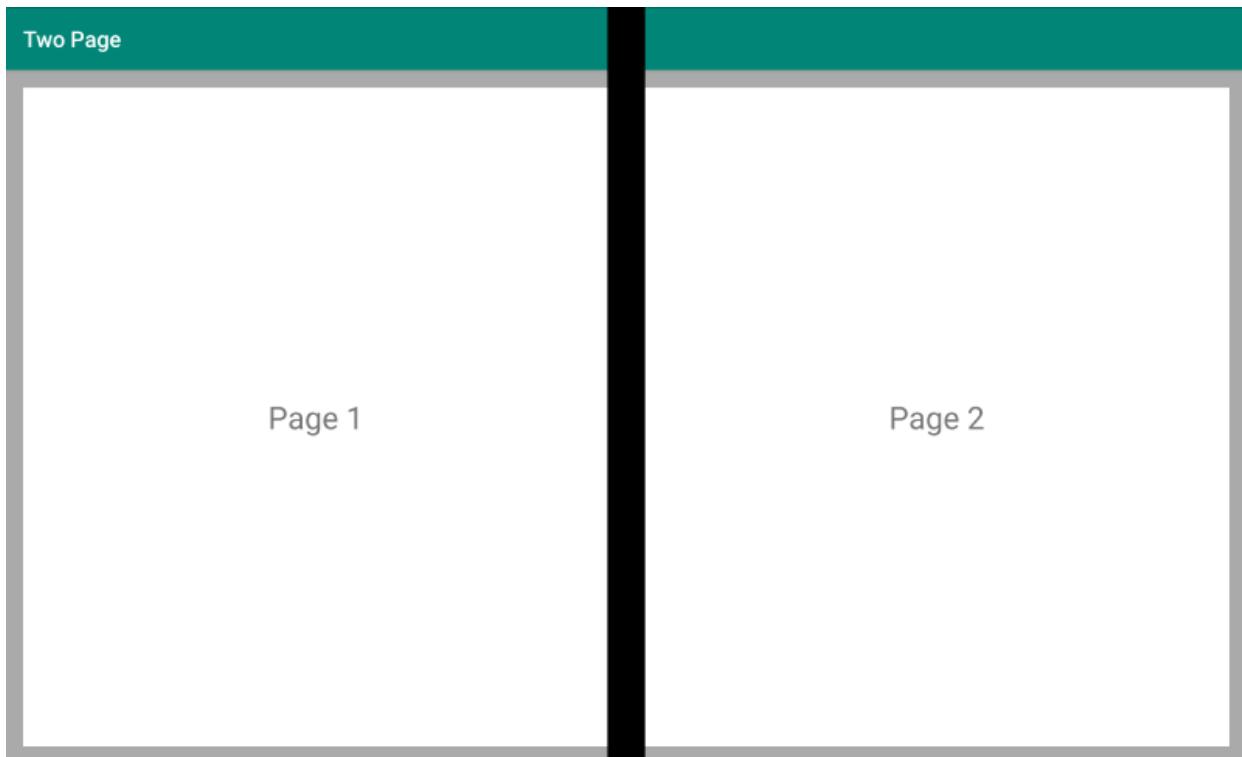


```
<ContentPage xmlns:local="clr-namespace:Xamarin.Duo.Forms.Samples"
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:dualScreen="clr-namespace:Xamarin.Forms.DualScreen;assembly=Xamarin.Forms.DualScreen"
    x:Class="Xamarin.Duo.Forms.Samples.MasterDetail">
    <dualScreen:TwoPaneView MinWideModeWidth="4000"
        MinTallModeHeight="4000">
        <dualScreen:TwoPaneView.Pane1>
            <local:Master x:Name="masterPage" />
        </dualScreen:TwoPaneView.Pane1>
        <dualScreen:TwoPaneView.Pane2>
            <local:Details x:Name="detailsPage" />
        </dualScreen:TwoPaneView.Pane2>
    </dualScreen:TwoPaneView>
</ContentPage>
```

In this example, you can make use of `TwoPaneView` to set a list on one pane, and a detail view on the other.

Two page pattern

The two page pattern is ideal for content that lends itself to a two-up layout, such as a document reader, notes, or an art-board:



```

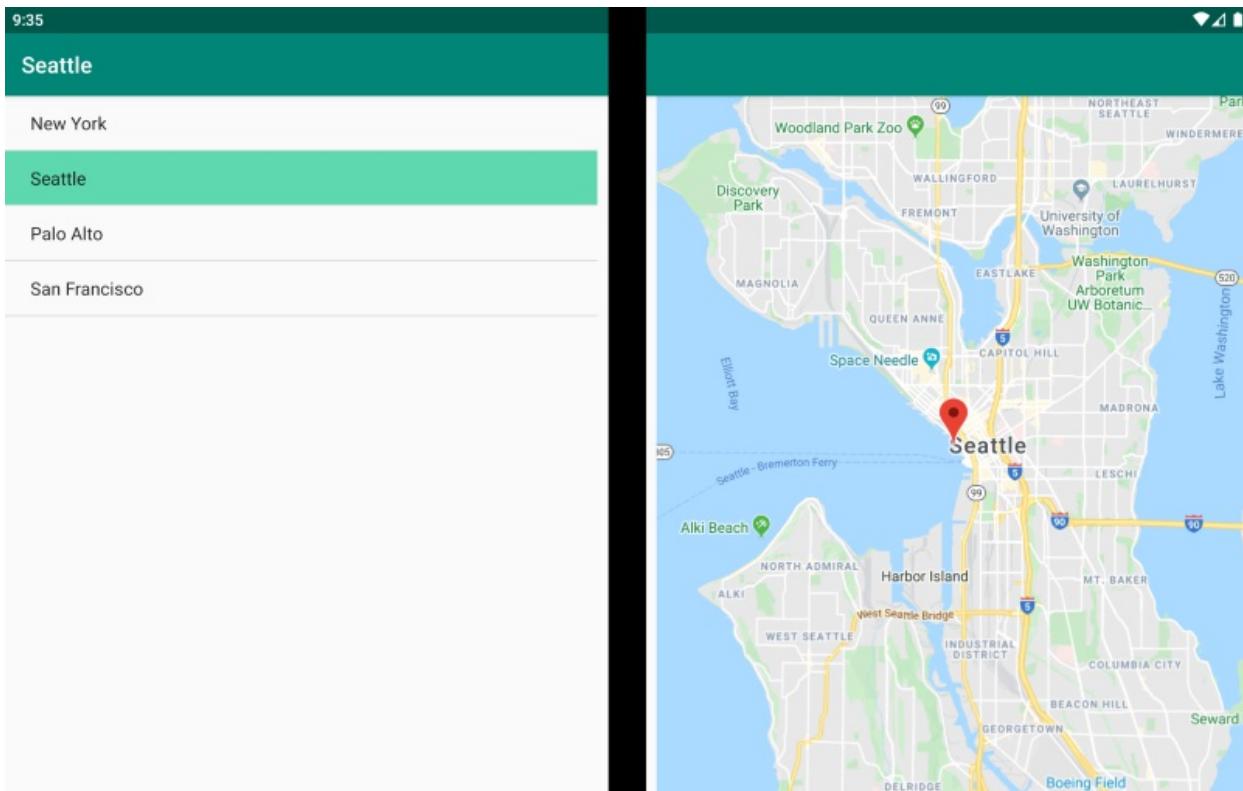
<Grid x:Name="layout">
    <CollectionView x:Name="cv"
        BackgroundColor="LightGray">
        <CollectionView.ItemsLayout>
            <GridItemsLayout SnapPointsAlignment="Start"
                SnapPointsType="MandatorySingle"
                Orientation="Horizontal"
                HorizontalItemSpacing="{Binding Source={x:Reference mainPage},
Path=HingeWidth}" />
        </CollectionView.ItemsLayout>
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <Frame BackgroundColor="LightGray"
                    Padding="0"
                    Margin="0"
                    WidthRequest="{Binding Source={x:Reference mainPage}, Path=ContentWidth}"
                    HeightRequest="{Binding Source={x:Reference mainPage}, Path=ContentHeight}">
                    <Frame Margin="20"
                        BackgroundColor="White">
                        <Label FontSize="Large"
                            Text="{Binding .}"
                            VerticalTextAlignment="Center"
                            HorizontalTextAlignment="Center"
                            HorizontalOptions="Center"
                            VerticalOptions="Center" />
                    </Frame>
                </Frame>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</Grid>

```

The `CollectionView`, with a grid layout that splits on the hinge width, makes for an ideal approach to deliver this dual-screen experience.

Dual view pattern

The dual view pattern may look just like the "Two page" view, but the distinction is in the content and user scenario. In this pattern, you are comparing content side by side, perhaps to edit a document or photo, to compare different restaurant menus, or to diff a merge conflict for code files:



```

<ContentPage xmlns:local="clr-namespace:Xamarin.Duo.Forms.Samples"
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:dualScreen="clr-namespace:Xamarin.Forms.DualScreen;assembly=Xamarin.Forms.DualScreen"
    x:Class="Xamarin.Duo.Forms.Samples.DualViewListPage">
    <dualScreen:TwoPaneView>
        <dualScreen:TwoPaneView.Pane1>
            <CollectionView x:Name="mapList"
                SelectionMode="Single">
                <CollectionView.ItemTemplate>
                    <DataTemplate>
                        <Grid Padding="10,5,10,5">
                            <Frame Visual="Material"
                                BorderColor="LightGray">
                                <StackLayout Padding="5">
                                    <Label FontSize="Title"
                                        Text="{Binding Title}" />
                                </StackLayout>
                            </Frame>
                        </Grid>
                    </DataTemplate>
                </CollectionView.ItemTemplate>
            </CollectionView>
        </dualScreen:TwoPaneView.Pane1>
        <dualScreen:TwoPaneView.Pane2>
            <local:DualViewMap x:Name="mapPage" />
        </dualScreen:TwoPaneView.Pane2>
    </dualScreen:TwoPaneView>
</ContentPage>

```

Companion pattern

The companion pattern demonstrates how you might use the second screen to provide a second level of content related to the primary view, like in the case of a drawing app, a game, or media editing:

Companion Pane

Slide Content 1



```
<ContentPage xmlns:local="clr-namespace:Xamarin.Duo.Forms.Samples"
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:dualscreen="clr-namespace:Xamarin.Forms.DualScreen;assembly=Xamarin.Forms.DualScreen"
    x:Name="mainPage"
    x:Class="Xamarin.Duo.Forms.Samples.CompanionPane"
    BackgroundColor="LightGray"
    Visual="Material">
    <dualscreen:TwoPaneView x:Name="twoPaneView"
        MinWideModeWidth="4000"
        MinTallModeHeight="4000">
        <dualscreen:TwoPaneView.Pane1>
            <CarouselView x:Name="cv"
                BackgroundColor="LightGray"
                IsScrollAnimated="False" >
                <CarouselView.ItemTemplate>
                    <DataTemplate>
                        <Frame BackgroundColor="LightGray"
                            Padding="0"
                            Margin="0"
                            WidthRequest="{Binding Source={x:Reference twoPaneView}, Path=Pane1.Width}"
                            HeightRequest="{Binding Source={x:Reference twoPaneView},
Path=Pane1.Height}">
                            <Frame Margin="20"
                                BackgroundColor="White">
                                <Label FontSize="Large"
                                    Text="{Binding ., StringFormat='Slide Content {0}'}"
                                    VerticalTextAlignment="Center"
                                    HorizontalTextAlignment="Center"
                                    HorizontalOptions="Center"
                                    VerticalOptions="Center" />
                            </Frame>
                        </Frame>
                    </DataTemplate>
                </CarouselView.ItemTemplate>
            </CarouselView>
        </dualscreen:TwoPaneView.Pane1>
        <dualscreen:TwoPaneView.Pane2>
            <CollectionView x:Name="indicators"
                SelectionMode="Single"
                Margin="20, 20, 20, 20"
                BackgroundColor="LightGray"
```

```

        WidthRequest="{Binding Source={x:Reference twoPaneView}, Path=Pane2.Width}"
        ItemsSource="{Binding Source={x:Reference cv}, Path=ItemsSource}">
    <CollectionView.Resources>
        <ResourceDictionary>
            <Style TargetType="Frame">
                <Setter Property="VisualStateManager.VisualStateGroups">
                    <VisualStateGroupList>
                        <VisualStateGroup x:Name="CommonStates">
                            <VisualState x:Name="Normal">
                                <VisualState.Setters>
                                    <Setter Property="Padding"
                                           Value="0" />
                                </VisualState.Setters>
                            </VisualState>
                            <VisualState x:Name="Selected">
                                <VisualState.Setters>
                                    <Setter Property="BorderColor"
                                           Value="Green" />
                                    <Setter Property="Padding"
                                           Value="1" />
                                </VisualState.Setters>
                            </VisualState>
                        </VisualStateGroup>
                    </VisualStateGroupList>
                </Setter>
            </Style>
        </ResourceDictionary>
    </CollectionView.Resources>
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical"
                           ItemSpacing="10" />
    </CollectionView.ItemsLayout>
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Frame WidthRequest="{Binding Source={x:Reference twoPaneView}, Path=Pane2.Width}"
                   CornerRadius="10"
                   HeightRequest="60"
                   BackgroundColor="White"
                   Margin="0">
                <StackLayout HorizontalOptions="Fill"
                            VerticalOptions="Fill"
                            Orientation="Horizontal">
                    <Label FontSize="Micro"
                           Padding="20,0,20,0"
                           VerticalTextAlignment="Center"
                           WidthRequest="140" Text="{Binding .., StringFormat='Slide Content
{0}'}" />
                    <Label FontSize="Small"
                           Padding="20,0,20,0"
                           VerticalTextAlignment="Center"
                           HorizontalOptions="FillAndExpand"
                           BackgroundColor="DarkGray"
                           Grid.Column="1"
                           Text="{Binding .., StringFormat='Slide {0}'}" />
                </StackLayout>
            </Frame>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
</dualscreen:TwoPaneView.Pane2>
</dualscreen:TwoPaneView>
</ContentPage>

```

Related links

- [DualScreen \(sample\)](#)

- Create apps for dual screen devices

Xamarin.Forms TwoPaneView layout

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `TwoPaneView` class represents a container with two views that size and position content in the available space, either side-by-side or top-to-bottom. `TwoPaneView` inherits from `Grid` so the easiest way to think about these properties is as if they are being applied to a grid.

Set up TwoPaneView

Follow these instructions to create a dual-screen layout in your app:

1. Follow the [get started](#) instructions to add the NuGet and configure the Android `MainActivity` class.
2. Start with a basic `TwoPaneView` using the following XAML:

```
<ContentPage
    xmlns:dualScreen="clr-namespace:Xamarin.Forms.DualScreen;assembly=Xamarin.Forms.DualScreen">
    <dualScreen:TwoPaneView>
        <dualScreen:TwoPaneView.Pane1>
            <StackLayout>
                <Label Text="Pane1 Content" />
            </StackLayout>
        </dualScreen:TwoPaneView.Pane1>
        <dualScreen:TwoPaneView.Pane2>
            <StackLayout>
                <Label Text="Pane2 Content" />
            </StackLayout>
        </dualScreen:TwoPaneView.Pane2>
    </dualScreen:TwoPaneView>
</ContentPage>
```

TIP

The above XAML omits many common attributes from the `ContentPage` element. When adding a `TwoPaneView` to your app, remember to declare the `xmlns:dualScreen` namespace as shown.

Understand TwoPaneView modes

Only one of these modes can be active:

- `SinglePane` only one pane is currently visible.
- `Wide` the two panes are laid out horizontally. One pane is on the left and the other is on the right. When on two screens this is the mode when the device is portrait.
- `Tall` the two panes are laid out vertically. One pane is on top and the other is on bottom. When on two screens this is the mode when the device is landscape.

Control TwoPaneView when it's only on one screen

The following properties apply when the `TwoPaneView` is occupying a single screen:

- `MinTallModeHeight` indicates the minimum height the control must be to enter tall mode.
- `MinWideModeWidth` indicates the minimum width the control must be to enter wide mode.
- `Pane1Length` sets the width of Pane1 in Wide mode, the height of Pane1 in Tall mode, and has no effect in SinglePane mode.
- `Pane2Length` sets the width of Pane2 in Wide mode, the height of Pane2 in Tall mode, and has no effect in SinglePane mode.

IMPORTANT

If the `TwoPaneView` is spanned across two screens these properties have no effect.

Properties that apply when on one screen or two

The following properties apply when the `TwoPaneView` is occupying a single screen or two screens:

- `TallModeConfiguration` indicates, when in tall mode, the Top/Bottom arrangement or if you only want a single pane visible as defined by the `TwoPaneViewPriority`.
- `WideModeConfiguration` indicates, when in wide mode, the Left/Right arrangement or if you only want a single pane visible as defined by the `TwoPaneViewPriority`.
- `PanePriority` determines whether to show Pane1 or Pane2 if in SinglePane mode.

Related links

- [DualScreen \(sample\)](#)

Xamarin.Forms DualScreenInfo helper class

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

The `DualScreenInfo` class enables you to determine which pane your view is on, how big it is, what posture the device is in, the angle of the hinge, and more.

Configure DualScreenInfo

Follow these instructions to create a dual-screen layout in your app:

1. Follow the [get started](#) instructions to add the NuGet and configure the Android `MainActivity` class.
2. Add `using Xamarin.Forms.DualScreen;` to your class file.
3. Use the `DualScreenInfo.Current` class in your app.

Properties

- `SpanningBounds` returns, when spanned across two screens, two rectangles indicating the bounds of each visible area. If the window isn't spanned this will return an empty array.
- `HingeBounds` indicates the position of the hinge on the screen.
- `IsLandscape` indicates if the device is landscape. This is useful because native orientation APIs don't report orientation correctly when an application is spanned.
- `SpanMode` indicates if the layout is in tall, wide, or single pane mode.

In addition, the `PropertyChanged` event fires when any properties change, and the `HingeAngleChanged` event fires when the hinge angle changes.

Poll hinge angle on Android and UWP

The following method is available when accessing `DualScreenInfo` from Android and UWP platform projects:

- `GetHingeAngleAsync` retrieves the current angle of the device hinge. When using the simulator the `HingeAngle` can be set by modifying the Pressure sensor.

This method can be invoked from custom renderers on Android and UWP. The following code shows an Android custom renderer example:

```

public class HingeAngleLabelRenderer : Xamarin.Forms.Platform.Android.FastRenderers.LabelRenderer
{
    System.Timers.Timer _hingeTimer;
    public HingeAngleLabelRenderer(Context context) : base(context)
    {
    }

    async void OnTimerElapsed(object sender, System.Timers.ElapsedEventArgs e)
    {
        if (_hingeTimer == null)
            return;

        _hingeTimer.Stop();
        var hingeAngle = await DualScreenInfo.Current.GetHingeAngleAsync();

        Device.BeginInvokeOnMainThread(() =>
        {
            if (_hingeTimer != null)
                Element.Text = hingeAngle.ToString();
        });
    }

    if (_hingeTimer != null)
        _hingeTimer.Start();
}

protected override void OnElementChanged(ElementChangedEventArgs<Label> e)
{
    base.OnElementChanged(e);

    if (_hingeTimer == null)
    {
        _hingeTimer = new System.Timers.Timer(100);
        _hingeTimer.Elapsed += OnTimerElapsed;
        _hingeTimer.Start();
    }
}

protected override void Dispose(bool disposing)
{
    if (_hingeTimer != null)
    {
        _hingeTimer.Elapsed -= OnTimerElapsed;
        _hingeTimer.Stop();
        _hingeTimer = null;
    }

    base.Dispose(disposing);
}
}

```

Access DualScreenInfo in your application window

The following code shows how to access `DualScreenInfo` for your application window:

```

DualScreenInfo currentWindow = DualScreenInfo.Current;

// Retrieve absolute position of the hinge on the screen
var hingeBounds = currentWindow.HingeBounds;

// check if app window is spanned across two screens
if(currentWindow.SpanMode == TwoPaneViewMode.SinglePane)
{
    // window is only on one screen
}
else if(currentWindow.SpanMode == TwoPaneViewMode.Tall)
{
    // window is spanned across two screens and oriented top-bottom
}
else if(currentWindow.SpanMode == TwoPaneViewMode.Wide)
{
    // window is spanned across two screens and oriented side-by-side
}

// Detect if any of the properties on DualScreenInfo change.
// This is useful to detect if the app window gets spanned
// across two screens or put on only one
currentWindow.PropertyChanged += OnDualScreenInfoChanged;

```

Apply DualScreenInfo to layouts

The `DualScreenInfo` class has a constructor that can take a layout and will give you information about the layout relative to the devices two screens:

```

<Grid x:Name="grid" ColumnSpacing="0">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="{Binding Column1Width}" />
        <ColumnDefinition Width="{Binding Column2Width}" />
        <ColumnDefinition Width="{Binding Column3Width}" />
    </Grid.ColumnDefinitions>
    <Label FontSize="Large"
        VerticalOptions="Center"
        HorizontalOptions="End"
        Text="I should be on the left side of the hinge" />
    <Label FontSize="Large"
        VerticalOptions="Center"
        HorizontalOptions="Start"
        Grid.Column="2"
        Text="I should be on the right side of the hinge" />
</Grid>

```

```

public partial class GridUsingDualScreenInfo : ContentPage
{
    public DualScreenInfo DualScreenInfo { get; }
    public double Column1Width { get; set; }
    public double Column2Width { get; set; }
    public double Column3Width { get; set; }

    public GridUsingDualScreenInfo()
    {
        InitializeComponent();
        DualScreenInfo = new DualScreenInfo(grid);
        BindingContext = this;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
        DualScreenInfo.PropertyChanged += OnInfoPropertyChanged;
        UpdateColumns();
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        DualScreenInfo.PropertyChanged -= OnInfoPropertyChanged;
    }

    void UpdateColumns()
    {
        // Check if grid is on two screens
        if (DualScreenInfo.SpanningBounds.Length > 0)
        {
            // set the width of the first column to the width of the layout
            // that's on the left screen
            Column1Width = DualScreenInfo.SpanningBounds[0].Width;

            // set the middle column to the width of the hinge
            Column2Width = DualScreenInfo.HingeBounds.Width;

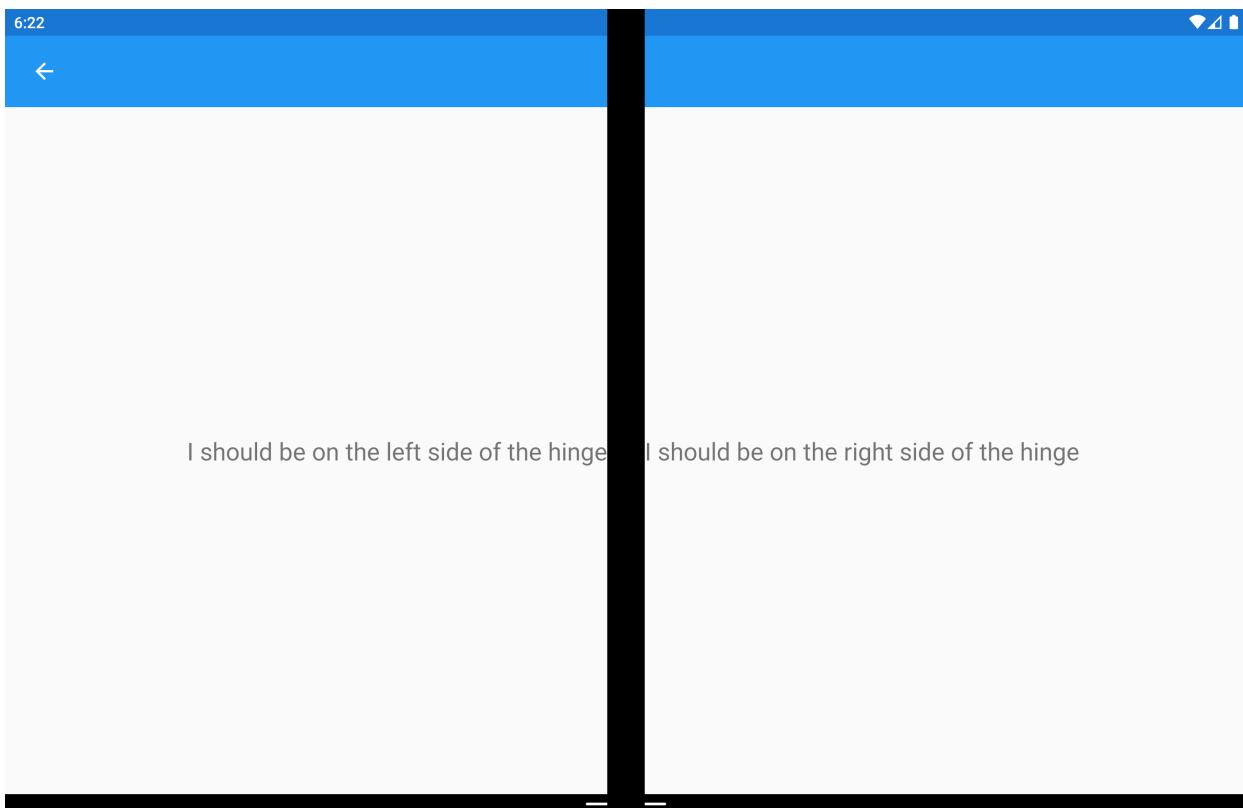
            // set the width of the third column to the width of the layout
            // that's on the right screen
            Column3Width = DualScreenInfo.SpanningBounds[1].Width;
        }
        else
        {
            Column1Width = 100;
            Column2Width = 0;
            Column3Width = 100;
        }

        OnPropertyChanged(nameof(Column1Width));
        OnPropertyChanged(nameof(Column2Width));
        OnPropertyChanged(nameof(Column3Width));
    }

    void OnInfoPropertyChanged(object sender, System.ComponentModel.PropertyChangedEventArgs e)
    {
        UpdateColumns();
    }
}

```

The following screenshot shows the resulting layout:



Related links

- [DualScreen \(sample\)](#)

Xamarin.Forms dual-screen triggers

8/4/2022 • 2 minutes to read • [Edit Online](#)

The `Xamarin.Forms.DualScreen` namespace includes two state triggers:

- `SpanModeStateTrigger` triggers a `VisualState` change when the view mode of the attached layout changes.
- `WindowSpanModeStateTrigger` triggers a `VisualState` change when the view mode of the window changes.

For more information about state triggers, see [State triggers](#).

Span mode state trigger

A `SpanModeStateTrigger` triggers a `VisualState` change when the span mode of the attached layout changes.

This trigger has a single bindable property:

- `SpanMode`, of type `TwoPaneViewMode`, which indicates the span mode to which the `VisualState` should be applied.

NOTE

The `SpanModeStateTrigger` derives from the `StateTriggerBase` class and can therefore attach an event handler to the `IsActiveChanged` event.

The following XAML example shows a `Grid` that includes `SpanModeStateTrigger` objects:

```

<Grid>
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup>
            <VisualState x:Name="GridSingle">
                <VisualState.StateTriggers>
                    <dualScreen:SpanModeStateTrigger SpanMode="SinglePane"/>
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Green" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="GridWide">
                <VisualState.StateTriggers>
                    <dualScreen:SpanModeStateTrigger SpanMode="Wide" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Red" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="GridTall">
                <VisualState.StateTriggers>
                    <dualScreen:SpanModeStateTrigger SpanMode="Tall" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Purple" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    ...
</Grid>

```

In this example, visual states are set on a `Grid` object. The background color of the `Grid` is green when only one pane is shown, is red when panes are shown side by side, and is purple when panes are shown top-bottom.

Window span mode state trigger

A `WindowSpanModeStateTrigger` triggers a `VisualState` change when the span mode of the window changes. This trigger has a single bindable property:

- `SpanMode`, of type `TwoPaneViewMode`, which indicates the span mode to which the `VisualState` should be applied.

NOTE

The `WindowSpanModeStateTrigger` derives from the `StateTriggerBase` class and can therefore attach an event handler to the `IsActiveChanged` event.

The following XAML example shows a `Grid` that includes `WindowSpanModeStateTrigger` objects:

```
<Grid>
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup>
            <VisualState x:Name="NotSpanned">
                <VisualState.StateTriggers>
                    <dualScreen:WindowSpanModeStateTrigger SpanMode="SinglePane"/>
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Red" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="Spanned">
                <VisualState.StateTriggers>
                    <dualScreen:WindowSpanModeStateTrigger SpanMode="Wide" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Green" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="Tall">
                <VisualState.StateTriggers>
                    <dualScreen:WindowSpanModeStateTrigger SpanMode="Tall" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Yellow" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    ...
</Grid>
```

In this example, visual states are set on a `Grid` object. The background color of the `Grid` is red when only one pane is shown, is green when panes are shown side by side, and is yellow when panes are shown top-bottom.

Related links

- [Xamarin.Forms Triggers](#)
- [Xamarin.Forms Visual State Manager](#)

Xamarin.Forms Effects

8/4/2022 • 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.

Reusable RoundEffect

RoundEffect is a reusable effect that can be applied to any control deriving from VisualElement to render the control as a circle. This effect can be used to create circular images, circular buttons, or other circular controls.

Introduction to Effects

8/4/2022 • 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

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

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:

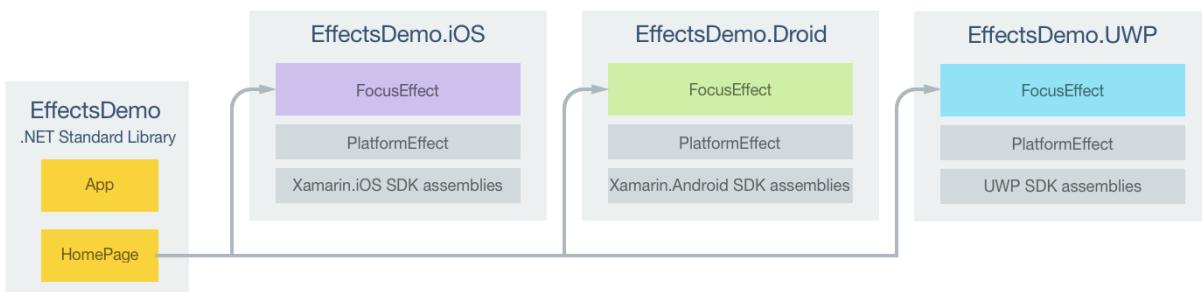
1. Create a subclass of the `PlatformEffect` class.
2. Override the `OnAttached` method and write logic to customize the control.
3. Override the `OnDetached` method and write logic to clean up the control customization, if required.
4. Add a `ResolutionGroupName` attribute to the effect class. This 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(EffectsDemo.iOS.FocusEffect), nameof(EffectsDemo.iOS.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 System;
using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

[assembly: ResolutionGroupName("MyCompany")]
[assembly: ExportEffect(typeof(EffectsDemo.Droid.FocusEffect), nameof(EffectsDemo.Droid.FocusEffect))]
namespace EffectsDemo.Droid
{
    public class FocusEffect : PlatformEffect
    {
        Android.Graphics.Color originalBackgroundColor = new Android.Graphics.Color(0, 0, 0, 0);
        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(originalBackgroundColor);
                    }
                    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(EffectsDemo.UWP.FocusEffect), nameof(EffectsDemo.UWP.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 the effect is attached to 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.{nameof(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.{nameof(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

8/4/2022 • 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

8/4/2022 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

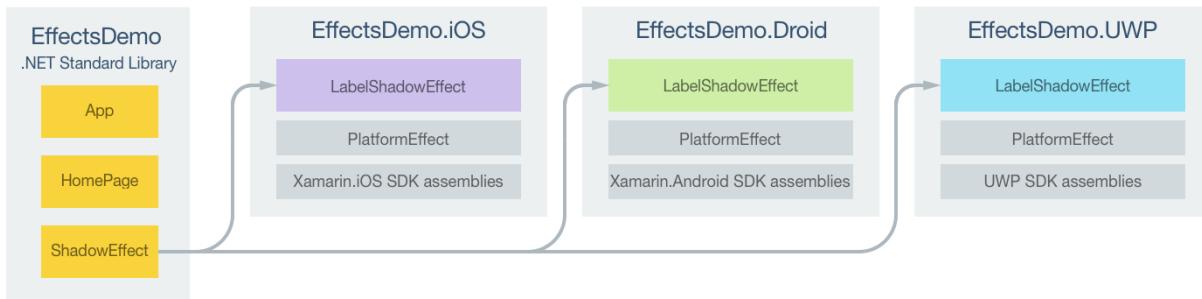
Common Language Runtime (CLR) properties can be used to define effect parameters that don't respond to runtime property changes. This article demonstrates using CLR properties to pass parameters to an effect.

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.ShadowRadius = 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

8/4/2022 • 10 minutes to read • [Edit Online](#)

 [Download the sample](#)

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:

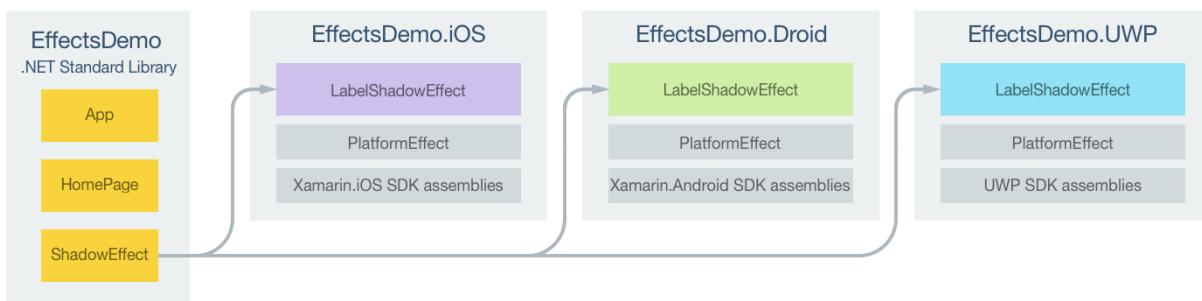
1. Create a `static` class that contains an attached property for each parameter to be passed to the effect.
2. Add an additional attached property to the class that will be used to control the addition or removal of the effect to the control that the class will be attached to. Ensure that this attached 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.ShadowRadius = (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

8/4/2022 • 21 minutes to read • [Edit Online](#)



[Download the sample](#)

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 pressed. 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 `Touch ActionType` enumeration for the type of touch events:

```
public enum Touch ActionType
{
    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, TouchEventArgs 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, TouchActionEventArgs> 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 TouchActionEventArgs(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.

IMPORTANT

Many of the views in `UIKit` do not have touch enabled by default. Touch can be enabled by adding `view.UserInteractionEnabled = true;` to the `OnAttached` override in the `TouchEvent` class in the iOS project. This should occur after the `UIView` is obtained that corresponds to the element the effect is attached to.

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 `TouchEvent` 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 += OnTouchEventAction;
    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;

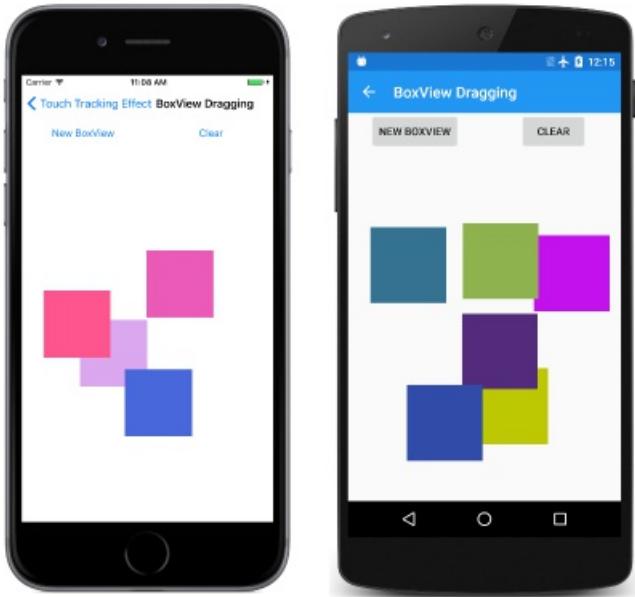
        case TouchActionType.Moved:
            if (dragDictionary.ContainsKey(boxView) && dragDictionary[boxView].Id == args.Id)
            {
                Rectangle rect = AbsoluteLayout.GetLayoutBounds(boxView);
                Point initialLocation = dragDictionary[boxView].PressPoint;
                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;
        }
    }
}

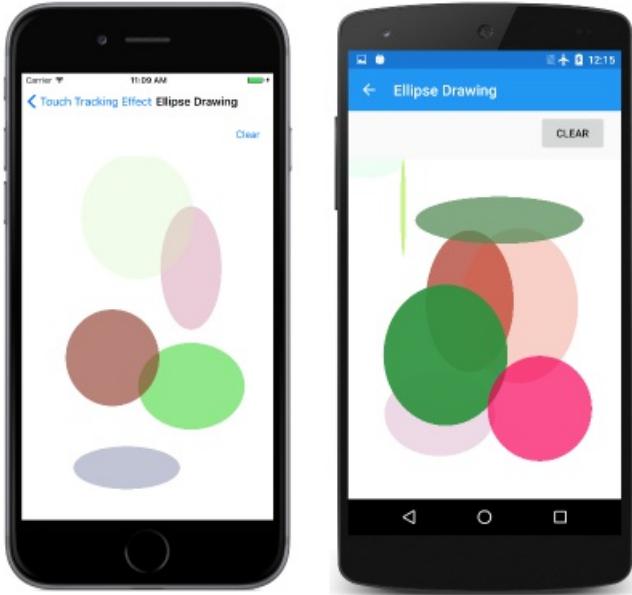
```

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;
                }
            }

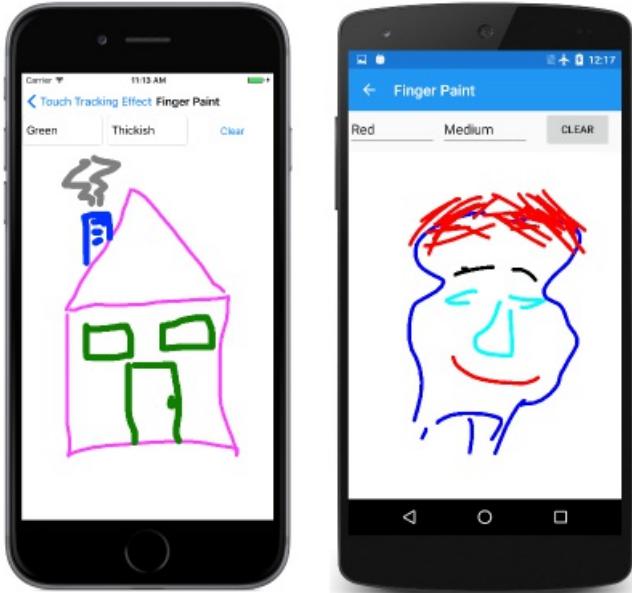
            if (isDragOperation)
            {
                fig.LastFingerLocation = args.Location;
                draggingFigures.Add(args.Id, fig);
                break;
            }
        }
    }

    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 \(sample\)](#)

Xamarin.Forms Reusable RoundEffect

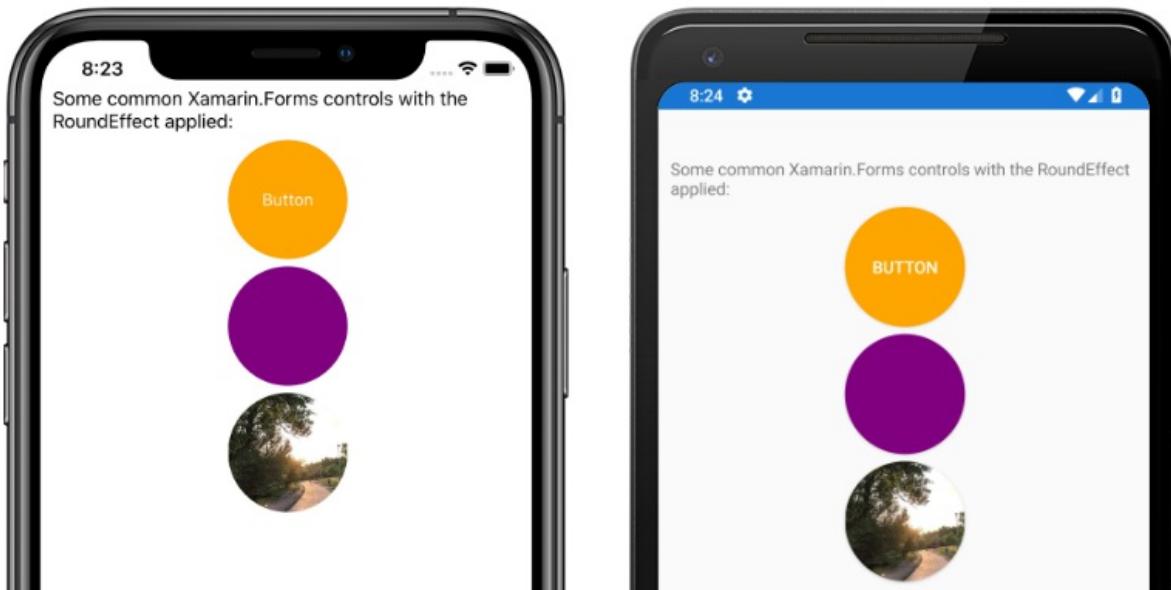
8/4/2022 • 4 minutes to read • [Edit Online](#)

 [Download the sample](#)

IMPORTANT

It's no longer necessary to use a `RoundEffect` to render a control as a circle. The latest recommended approach is to clip the control using an `EllipseGeometry`. For more information, see [Clip with a Geometry](#).

The `RoundEffect` simplifies rendering any control that derives from `VisualElement` as a circle. This effect can be used to create circular images, buttons, and other controls:



Create a shared RoutingEffect

An effect class must be created in the shared project to create a cross-platform effect. The sample application creates an empty `RoundEffect` class that derives from the `RoutingEffect` class:

```
public class RoundEffect : RoutingEffect
{
    public RoundEffect() : base($"Xamarin.{nameof(RoundEffect)}")
    {
    }
}
```

This class allows the shared project to resolve the references to the effect in code or XAML but does not provide any functionality. The effect must have implementations for each platform.

Implement the Android effect

The Android platform project defines a `RoundEffect` class that derives from `PlatformEffect`. This class is tagged with `assembly` attributes that allow Xamarin.Forms to resolve the effect class:

```
[assembly: ResolutionGroupName("Xamarin")]
[assembly: ExportEffect(typeof(RoundEffectDemo.Droid.RoundEffect),
nameof(RoundEffectDemo.Droid.RoundEffect))]
namespace RoundEffectDemo.Droid
{
    public class RoundEffect : PlatformEffect
    {
        // ...
    }
}
```

The Android platform uses the concept of an `OutlineProvider` to define the edges of a control. The sample project includes a `CornerRadiusProvider` class that derives from the `ViewOutlineProvider` class:

```
class CornerRadiusOutlineProvider : ViewOutlineProvider
{
    Element element;

    public CornerRadiusOutlineProvider(Element formsElement)
    {
        element = formsElement;
    }

    public override void GetOutline(Android.Views.View view, Outline outline)
    {
        float scale = view.Resources.DisplayMetrics.Density;
        double width = (double)element.GetValue(VisualElement.WidthProperty) * scale;
        double height = (double)element.GetValue(VisualElement.HeightProperty) * scale;
        float minDimension = (float)Math.Min(height, width);
        float radius = minDimension / 2f;
        Rect rect = new Rect(0, 0, (int)width, (int)height);
        outline.SetRoundRect(rect, radius);
    }
}
```

This class uses the `Width` and `Height` properties of the `Xamarin.Forms Element` instance to calculate a radius that is half of the shortest dimension.

Once an outline provider is defined the `RoundEffect` class can consume it to implement the effect:

```

public class RoundEffect : PlatformEffect
{
    ViewOutlineProvider originalProvider;
    Android.Views.View effectTarget;

    protected override void OnAttached()
    {
        try
        {
            effectTarget = Control ?? Container;
            originalProvider = effectTarget.OutlineProvider;
            effectTarget.OutlineProvider = new CornerRadiusOutlineProvider(Element);
            effectTarget.ClipToOutline = true;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Failed to set corner radius: {ex.Message}");
        }
    }

    protected override void OnDetached()
    {
        if(effectTarget != null)
        {
            effectTarget.OutlineProvider = originalProvider;
            effectTarget.ClipToOutline = false;
        }
    }
}

```

The `OnAttached` method is called when the effect is attached to an element. The existing `OutlineProvider` object is saved so it can be restored when the effect is detached. A new instance of the `CornerRadiusOutlineProvider` is used as the `OutlineProvider` and `ClipToOutline` is set to true to clip overflowing elements to the outline borders.

The `OnDetached` method is called when the effect is removed from an element and restores the original `OutlineProvider` value.

NOTE

Depending on the element type, the `Control` property may or may not be null. If the `Control` property is not null, the rounded corners can be applied directly to the control. However, if it is null the rounded corners must be applied to the `Container` object. The `effectTarget` field allows the effect to be applied to the appropriate object.

Implement the iOS effect

The iOS platform project defines a `RoundEffect` class that derives from `PlatformEffect`. This class is tagged with `assembly` attributes that allow Xamarin.Forms to resolve the effect class:

```

[assembly: ResolutionGroupName("Xamarin")]
[assembly: ExportEffect(typeof(RoundEffectDemo.iOS.RoundEffect), nameof(RoundEffectDemo.iOS.RoundEffect))]
namespace RoundEffectDemo.iOS
{
    public class RoundEffect : PlatformEffect
    {
        // ...
    }
}

```

On iOS, controls have a `Layer` property, which has a `CornerRadius` property. The `RoundEffect` class

implementation on iOS calculates the appropriate corner radius and updates the layer's `CornerRadius` property:

```
public class RoundEffect : PlatformEffect
{
    nfloat originalRadius;
    UIKit.UIView effectTarget;

    protected override void OnAttached()
    {
        try
        {
            effectTarget = Control ?? Container;
            originalRadius = effectTarget.Layer.CornerRadius;
            effectTarget.ClipsToBounds = true;
            effectTarget.Layer.CornerRadius = CalculateRadius();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Failed to set corner radius: {ex.Message}");
        }
    }

    protected override void OnDetached()
    {
        if (effectTarget != null)
        {
            effectTarget.ClipsToBounds = false;
            if (effectTarget.Layer != null)
            {
                effectTarget.Layer.CornerRadius = originalRadius;
            }
        }
    }

    float CalculateRadius()
    {
        double width = (double)Element.GetValue(VisualElement.WidthRequestProperty);
        double height = (double)Element.GetValue(VisualElement.HeightRequestProperty);
        float minDimension = (float)Math.Min(height, width);
        float radius = minDimension / 2f;

        return radius;
    }
}
```

The `CalculateRadius` method calculates a radius based on the minimum dimension of the `Xamarin.Forms Element`. The `OnAttached` method is called when the effect is attached to a control, and updates the layer's `CornerRadius` property. It sets the `clipToBounds` property to `true` so overflowing elements are clipped to the borders of the control. The `OnDetached` method is called when the effect is removed from a control and reverses these changes, restoring the original corner radius.

NOTE

Depending on the element type, the `Control` property may or may not be null. If the `Control` property is not null, the rounded corners can be applied directly to the control. However, if it is null the rounded corners must be applied to the `Container` object. The `effectTarget` field allows the effect to be applied to the appropriate object.

Consume the effect

Once the effect is implemented across platforms, it can be consumed by `Xamarin.Forms` controls. A common application of the `RoundEffect` is making an `Image` object circular. The following XAML shows the effect being

applied to an `Image` instance:

```
<Image Source="outdoors"
      HeightRequest="100"
      WidthRequest="100">
  <Image.Effects>
    <local:RoundEffect />
  </Image.Effects>
</Image>
```

The effect can also be applied in code:

```
var image = new Image
{
  Source = ImageSource.FromFile("outdoors"),
  HeightRequest = 100,
  WidthRequest = 100
};
image.Effects.Add(new RoundEffect());
```

The `RoundEffect` class can be applied to any control that derives from `VisualElement`.

NOTE

For the effect to calculate the correct radius, the control it's applied to must have explicit sizing. Therefore, the `HeightRequest` and `WidthRequest` properties should be defined. If the affected control appears in a `StackLayout`, its `HorizontalOptions` property should not use one of the `Expand` values such as `LayoutOptions.CenterAndExpand` or it will not have accurate dimensions.

Related links

- [RoundEffect sample application](#)
- [Introduction to Effects](#)
- [Creating an Effect](#)

Xamarin.Forms gestures

8/4/2022 • 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, swipe, and drag and drop gestures on [View](#) instances.

Add a tap gesture recognizer

A tap gesture is used for tap detection and is recognized with the [TapGestureRecognizer](#) class.

Add a pinch gesture recognizer

A pinch gesture is used for performing interactive zoom and is recognized with the [PinchGestureRecognizer](#) class.

Add 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.

Add 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.

Add a drag and drop gesture recognizer

A drag and drop gesture enables items, and their associated data packages, to be dragged from one onscreen location to another location using a continuous gesture. Drag gestures are recognized with the [DragGestureRecognizer](#) class, and drop gestures are recognized with the [DropGestureRecognizer](#) class.

Add a tap gesture recognizer

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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](#)

Add a pinch gesture recognizer

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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](#)

Add a pan gesture recognizer

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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
}
```

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](#)

Add a swipe gesture recognizer

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

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 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](#)

Add drag and drop gesture recognizers

8/4/2022 • 11 minutes to read • [Edit Online](#)



[Download the sample](#)

A drag and drop gesture enables items, and their associated data packages, to be dragged from one onscreen location to another location using a continuous gesture. Drag and drop can take place in a single application, or it can start in one application and end in another.

IMPORTANT

Recognition of drag and drop gestures is supported on iOS, Android, and the Universal Windows Platform (UWP). However, on iOS a minimum platform of iOS 11 is required.

The *drag source*, which is the element on which the drag gesture is initiated, can provide data to be transferred by populating a data package object. When the drag source is released, drop occurs. The *drop target*, which is the element under the drag source, then processes the data package.

The process for enabling drag and drop in an application is as follows:

1. Enable drag on an element by adding a `DragGestureRecognizer` object to its `GestureRecognizers` collection. For more information, see [Enable drag](#).
2. [optional] Build a data package. Xamarin.Forms automatically populates the data package for image and text controls, but for other content you'll need to construct your own data package. For more information, see [Build a data package](#).
3. Enable drop on an element by adding a `DropGestureRecognizer` object to its `GestureRecognizers` collection. For more information, see [Enable drop](#).
4. [optional] Handle the `DropGestureRecognizer.DragOver` event to indicate the type of operation allowed by the drop target. For more information, see [Handle the DragOver event](#).
5. [optional] Process the data package to receive the dropped content. Xamarin.Forms will automatically retrieve image and text data from the data package, but for other content you'll need to process the data package. For more information, see [Process the data package](#).

NOTE

Dragging items to and from a `CollectionView` is currently unsupported.

Enable drag

In Xamarin.Forms, drag gesture recognition is provided by the `DragGestureRecognizer` class. This class defines the following properties:

- `CanDrag`, of type `bool`, which indicates whether the element the gesture recognizer is attached to can be a drag source. The default value of this property is `true`.
- `DragStartingCommand`, of type `ICommand`, which is executed when a drag gesture is first recognized.
- `DragStartingCommandParameter`, of type `object`, which is the parameter that's passed to the `DragStartingCommand`.
- `DropCompletedCommand`, of type `ICommand`, which is executed when the drag source is dropped.

- `DropCompletedCommandParameter`, of type `object`, which is the parameter that's passed to the `DropCompletedCommand`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `DragGestureRecognizer` class also defines `DragStarting` and `DropCompleted` events that fire provided that the `CanDrag` property is `true`. When a `DragGestureRecognizer` object detects a drag gesture, it executes the `DragStartingCommand` and invokes the `DragStarting` event. Then, when the `DragGestureRecognizer` object detects the completion of a drop gesture, it executes the `DropCompletedCommand` and invokes the `DropCompleted` event.

The `DragStartingEventArgs` object that accompanies the `DragStarting` event defines the following properties:

- `Handled`, of type `bool`, indicates whether the event handler has handled the event or whether Xamarin.Forms should continue its own processing.
- `Cancel`, of type `bool`, indicates whether the event should be canceled.
- `Data`, of type `DataPackage`, indicates the data package that accompanies the drag source. This is a read-only property.

The following XAML example shows a `DragGestureRecognizer` attached to an `Image`:

```
<Image Source="monkeyface.png">
    <Image.GestureRecognizers>
        <DragGestureRecognizer />
    </Image.GestureRecognizers>
</Image>
```

In this example, a drag gesture can be initiated on the `Image`.

TIP

On iOS, Android, and UWP, a drag gesture is initiated with a long-press followed by a drag.

For an example of using `DragGestureRecognizer` commands, see the [sample](#).

Build a data package

Xamarin.Forms will automatically build a data package for you, when a drag is initiated, for the following controls:

- Text controls. Text values can be dragged from `CheckBox`, `DatePicker`, `Editor`, `Entry`, `Label`, `RadioButton`, `Switch`, and `TimePicker` objects.
- Image controls. Images can be dragged from `Button`, `Image`, and `ImageButton` controls.

The following table shows the properties that are read, and any conversion that's attempted, when a drag is initiated on a text control:

CONTROL	PROPERTY	CONVERSION
<code>CheckBox</code>	<code>IsChecked</code>	<code>bool</code> converted to a <code>string</code> .
<code>DatePicker</code>	<code>Date</code>	<code>DateTime</code> converted to a <code>string</code> .
<code>Editor</code>	<code>Text</code>	

CONTROL	PROPERTY	CONVERSION
Entry	Text	
Label	Text	
RadioButton	IsChecked	bool converted to a string.
Switch	IsToggled	bool converted to a string.
TimePicker	Time	TimeSpan converted to a string.

For content other than text and images, you'll need to build a data package yourself.

Data packages are represented by the `DataPackage` class, which defines the following properties:

- `Properties`, of type `DataPackagePropertySet`, which is a collection of properties that comprise the data contained in the `DataPackage`. This property is a read-only property.
- `Image`, of type `ImageSource`, which is the image contained in the `DataPackage`.
- `Text`, of type `string`, which is the text contained in the `DataPackage`.
- `View`, of type `DataPackageView`, which is a read-only version of the `DataPackage`.

The `DataPackagePropertySet` class represents a property bag stored as a `Dictionary<string,object>`. For information about the `DataPackageView` class, see [Process the data package](#).

Store image or text data

Image or text data can be associated with a drag source by storing the data in the `DataPackage.Image` or `DataPackage.Text` property. This can be accomplished in the handler for the `DragStarting` event.

The following XAML example shows a `DragGestureRecognizer` that registers a handler for the `DragStarting` event:

```
<Path Stroke="Black"
      StrokeThickness="4">
    <Path.GestureRecognizers>
        <DragGestureRecognizer DragStarting="OnDragStarting" />
    </Path.GestureRecognizers>
    <Path.Data>
        <!-- PathGeometry goes here -->
    </Path.Data>
</Path>
```

In this example, the `DragGestureRecognizer` is attached to a `Path` object. The `DragStarting` event is fired when a drag gesture is detected on the `Path`, which executes the `OnDragStarting` event handler:

```
void OnDragStarting(object sender, DragStartingEventArgs e)
{
    e.Data.Text = "My text data goes here";
}
```

The `DragStartingEventArgs` object that accompanies the `DragStarting` event has a `Data` property, of type `DataPackage`. In this example, the `Text` property of the `DataPackage` object is set to a `string`. The `DataPackage` can then be accessed on drop, to retrieve the `string`.

Store data in the property bag

Any data, including images and text, can be associated with a drag source by storing the data in the `DataPackage.Properties` collection. This can be accomplished in the handler for the `DragStarting` event.

The following XAML example shows a `DragGestureRecognizer` that registers a handler for the `DragStarting` event:

```
<Rectangle Stroke="Red"
           Fill="DarkBlue"
           StrokeThickness="4"
           HeightRequest="200"
           WidthRequest="200">
    <Rectangle.GestureRecognizers>
        <DragGestureRecognizer DragStarting="OnDragStarting" />
    </Rectangle.GestureRecognizers>
</Rectangle>
```

In this example, the `DragGestureRecognizer` is attached to a `Rectangle` object. The `DragStarting` event is fired when a drag gesture is detected on the `Rectangle`, which executes the `OnDragStarting` event handler:

```
void OnDragStarting(object sender, DragStartingEventArgs e)
{
    Shape shape = (sender as Element).Parent as Shape;
    e.Data.Properties.Add("Square", new Square(shape.Width, shape.Height));
}
```

The `DragStartingEventArgs` object that accompanies the `DragStarting` event has a `Data` property, of type `DataPackage`. The `Properties` collection of the `DataPackage` object, which is a `Dictionary<string, object>` collection, can be modified to store any required data. In this example, the `Properties` dictionary is modified to store a `Square` object, that represents the size of the `Rectangle`, against a "Square" key.

Enable drop

In Xamarin.Forms, drop gesture recognition is provided by the `DropGestureRecognizer` class. This class defines the following properties:

- `AllowDrop`, of type `bool`, which indicates whether the element the gesture recognizer is attached to can be a drop target. The default value of this property is `true`.
- `DragOverCommand`, of type `ICommand`, which is executed when the drag source is dragged over the drop target.
- `DragOverCommandParameter`, of type `object`, which is the parameter that's passed to the `DragOverCommand`.
- `DragLeaveCommand`, of type `ICommand`, which is executed when the drag source is dragged off the drop target.
- `DragLeaveCommandParameter`, of type `object`, which is the parameter that's passed to the `DragLeaveCommand`.
- `DropCommand`, of type `ICommand`, which is executed when the drag source is dropped over the drop target.
- `DropCommandParameter`, of type `object`, which is the parameter that's passed to the `DropCommand`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `DropGestureRecognizer` class also defines `DragOver`, `DragLeave`, and `Drop` events that fire provided that the `AllowDrop` property is `true`. When a `DropGestureRecognizer` recognizes a drag source over the drop target, it executes the `DragOverCommand` and invokes the `DragOver` event. Then, if the drag source is dragged off the drop target, the `DropGestureRecognizer` executes the `DragLeaveCommand` and invokes the `DragLeave` event. Finally, when the `DropGestureRecognizer` recognizes a drop gesture over the drop target, it executes the `DropCommand` and invokes the `Drop` event.

The `DragEventArgs` class, that accompanies the `DragOver` and `DragLeave` events, defines the following

properties:

- `Data`, of type `DataPackage`, which contains the data associated with the drag source. This property is read-only.
- `AcceptedOperation`, of type `DataPackageOperation`, which specifies which operations are allowed by the drop target.

For information about the `DataPackageOperation` enumeration, see [Handle the DragOver event](#).

The `DropEventArgs` class that accompanies the `Drop` event defines the following properties:

- `Data`, of type `DataPackageView`, which is a read-only version of the data package.
- `Handled`, of type `bool`, indicates whether the event handler has handled the event or whether Xamarin.Forms should continue its own processing.

The following XAML example shows a `DropGestureRecognizer` attached to an `Image`:

```
<Image BackgroundColor="Silver"
       HeightRequest="300"
       WidthRequest="250">
    <Image.GestureRecognizers>
        <DropGestureRecognizer />
    </Image.GestureRecognizers>
</Image>
```

In this example, when a drag source is dropped on the `Image` drop target, the drag source will be copied to the drop target, provided that the drag source is an `ImageSource`. This occurs because Xamarin.Forms automatically copies dragged images, and text, to compatible drop targets.

For an example of using `DropGestureRecognizer` commands, see the [sample](#).

Handle the DragOver event

The `DropGestureRecognizer.DragOver` event can be optionally handled to indicate which type of operations are allowed by the drop target. This can be accomplished by setting the `AcceptedOperation` property, of type `DataPackageOperation`, of the `DragEventArgs` object that accompanies the `DragOver` event.

The `DataPackageOperation` enumeration defines the following members:

- `None`, indicates that no action will be performed.
- `Copy`, indicates that the drag source content will be copied to the drop target.

IMPORTANT

When a `DragEventArgs` object is created, the `AcceptedOperation` property defaults to `DataPackageOperation.Copy`.

The following XAML example shows a `DropGestureRecognizer` that registers a handler for the `DragOver` event:

```
<Image BackgroundColor="Silver"
       HeightRequest="300"
       WidthRequest="250">
    <Image.GestureRecognizers>
        <DropGestureRecognizer DragOver="OnDragOver" />
    </Image.GestureRecognizers>
</Image>
```

In this example, the `DropGestureRecognizer` is attached to an `Image` object. The `DragOver` event is fired when a drag source is dragged over the drop target, but hasn't been dropped, which executes the `OnDragOver` event handler:

```
void OnDragOver(object sender, DragEventArgs e)
{
    e.AcceptedOperation = DataPackageOperation.None;
}
```

In this example, the `AcceptedOperation` property of the `DragEventArgs` object is set to `DataPackageOperation.None`. This ensures that no action is taken when a drag source is dropped over the drop target.

Process the data package

The `Drop` event is fired when a drag source is released over a drop target. When this occurs, Xamarin.Forms will automatically attempt to retrieve data from the data package, when a drag source is dropped onto the following controls:

- Text controls. Text values can be dropped onto `CheckBox`, `DatePicker`, `Editor`, `Entry`, `Label`, `RadioButton`, `Switch`, and `TimePicker` objects.
- Image controls. Images can be dropped onto `Button`, `Image`, and `ImageButton` controls.

The following table shows the properties that are set, and any conversion that's attempted, when a text-based drag source is dropped on a text control:

CONTROL	PROPERTY	CONVERSION
<code>CheckBox</code>	<code>IsChecked</code>	<code>string</code> is converted to a <code>bool</code> .
<code>DatePicker</code>	<code>Date</code>	<code>string</code> is converted to a <code>DateTime</code> .
<code>Editor</code>	<code>Text</code>	
<code>Entry</code>	<code>Text</code>	
<code>Label</code>	<code>Text</code>	
<code>RadioButton</code>	<code>IsChecked</code>	<code>string</code> is converted to a <code>bool</code> .
<code>Switch</code>	<code>IsToggled</code>	<code>string</code> is converted to a <code>bool</code> .
<code>TimePicker</code>	<code>Time</code>	<code>string</code> is converted to a <code> TimeSpan</code> .

For content other than text and images, you'll need to process the data package yourself.

The `DropEventArgs` class that accompanies the `Drop` event defines a `Data` property, of type `DataPackageView`. This property represents a read-only version of the data package.

Retrieve image or text data

Image or text data can be retrieved from a data package in the handler for the `Drop` event, using methods defined in the `DataPackageView` class.

The `DataPackageView` class includes `GetImageAsync` and `GetTextAsync` methods. The `GetImageAsync` method

retrieves an image from the data package, that was stored in the `DataPackage.Image` property, and returns `Task<ImageSource>`. Similarly, the `GetTextAsync` method retrieves text from the data package, that was stored in the `DataPackage.Text` property, and returns `Task<string>`.

The following example shows a `Drop` event handler that retrieves text from the data package for a `Path`:

```
async void OnDrop(object sender, DropEventArgs e)
{
    string text = await e.Data.GetTextAsync();

    // Perform logic to take action based on the text value.
}
```

In this example, text data is retrieved from the data package using the `GetTextAsync` method. An action based on the text value can then be taken.

Retrieve data from the property bag

Any data can be retrieved from a data package in the handler for the `Drop` event, by accessing the `Properties` collection of the data package.

The `DataPackageView` class defines a `Properties` property, of type `DataPackagePropertySetView`. The `DataPackagePropertySetView` class represents a read-only property bag stored as a `Dictionary<string, object>`.

The following example shows a `Drop` event handler that retrieves data from the property bag of a data package for a `Rectangle`:

```
void OnDrop(object sender, DropEventArgs e)
{
    Square square = (Square)e.Data.Properties["Square"];

    // Perform logic to take action based on retrieved value.
}
```

In this example, the `Square` object is retrieved from the property bag of the data package, by specifying the "Square" dictionary key. An action based on the retrieved value can then be taken.

Related links

- [Drag and drop gesture \(sample\)](#)

Local notifications in Xamarin.Forms

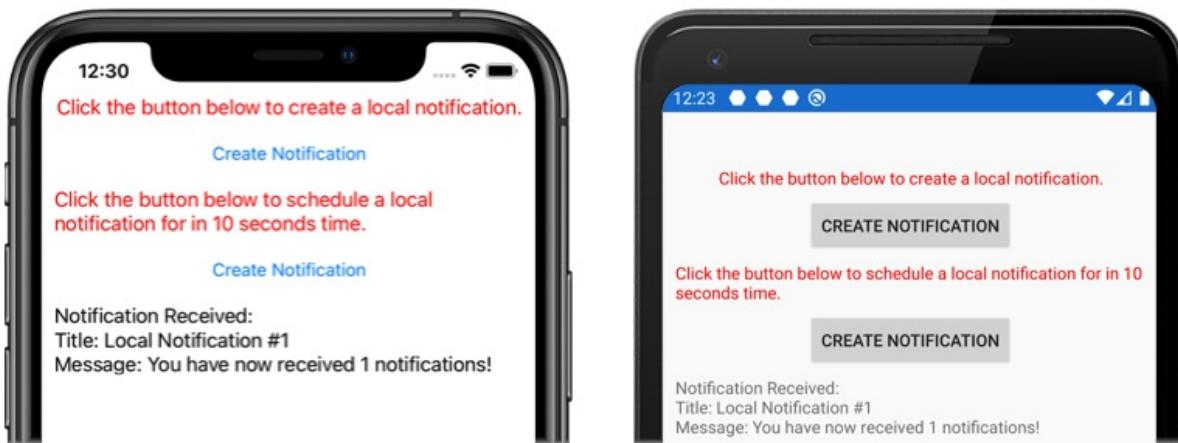
8/4/2022 • 9 minutes to read • [Edit Online](#)

 [Download the sample](#)

Local notifications are alerts sent by applications installed on a mobile device. Local notifications are often used for features such as:

- Calendar events
- Reminders
- Location-based triggers

Each platform handles the creation, display, and consumption of local notifications differently. This article explains how to create a cross-platform abstraction to send, schedule, and receive local notifications with Xamarin.Forms.



Create a cross-platform interface

The Xamarin.Forms application should create and consume notifications without concern for the underlying platform implementations. The following `INotificationManager` interface is implemented in the shared code library, and defines a cross-platform API that the application can use to interact with notifications:

```
public interface INotificationManager
{
    event EventHandler NotificationReceived;
    void Initialize();
    void SendNotification(string title, string message, DateTime? notifyTime = null);
    void ReceiveNotification(string title, string message);
}
```

This interface will be implemented in each platform project. The `NotificationReceived` event allows the application to handle incoming notifications. The `Initialize` method should perform any native platform logic needed to prepare the notification system. The `SendNotification` method should send a notification, at an optional `DateTime`. The `ReceiveNotification` method should be called by the underlying platform when a message is received.

Consume the interface in Xamarin.Forms

Once an interface has been created, it can be consumed in the shared Xamarin.Forms project even though platform implementations haven't been created yet. The sample application contains a `ContentPage` called `MainPage.xaml` with the following content:

```
<StackLayout Margin="0,35,0,0"
    x:Name="stackLayout">
    <Label Text="Click the button below to create a local notification."
        TextColor="Red"
        HorizontalOptions="Center"
        VerticalOptions="Start" />
    <Button Text="Create Notification"
        HorizontalOptions="Center"
        VerticalOptions="Start"
        Clicked="OnSendClick" />
    <Label Text="Click the button below to schedule a local notification for in 10 seconds time."
        TextColor="Red"
        HorizontalOptions="Center"
        VerticalOptions="Start" />
    <Button Text="Create Notification"
        HorizontalOptions="Center"
        VerticalOptions="Start"
        Clicked="OnScheduleClick" />
</StackLayout>
```

The layout contains `Label` elements that explain instructions, and `Button` elements that send or schedule a notification when tapped.

The `MainPage` class code-behind handles the sending and receiving of notifications:

```

public partial class MainPage : ContentPage
{
    INotificationManager notificationManager;
    int notificationNumber = 0;

    public MainPage()
    {
        InitializeComponent();

        notificationManager = DependencyService.Get<INotificationManager>();
        notificationManager.NotificationReceived += (sender, eventArgs) =>
        {
            var evtData = (NotificationEventArgs)eventArgs;
            ShowNotification(evtData.Title, evtData.Message);
        };
    }

    void OnSendClick(object sender, EventArgs e)
    {
        notificationNumber++;
        string title = $"Local Notification #{notificationNumber}";
        string message = $"You have now received {notificationNumber} notifications!";
        notificationManager.SendNotification(title, message);
    }

    void OnScheduleClick(object sender, EventArgs e)
    {
        notificationNumber++;
        string title = $"Local Notification #{notificationNumber}";
        string message = $"You have now received {notificationNumber} notifications!";
        notificationManager.SendNotification(title, message, DateTime.Now.AddSeconds(10));
    }

    void ShowNotification(string title, string message)
    {
        Device.BeginInvokeOnMainThread(() =>
        {
            var msg = new Label()
            {
                Text = $"Notification Received:\nTitle: {title}\nMessage: {message}"
            };
            stackLayout.Children.Add(msg);
        });
    }
}

```

The `MainPage` class constructor uses the `Xamarin.Forms DependencyService` to retrieve a platform-specific instance of the `INotificationManager`. The `OnSendClick` and `OnScheduleClick` methods use the `INotificationManager` instance to send and schedule new notifications. The `ShowNotification` method is called from the event handler attached to the `NotificationReceived` event, and will insert a new `Label` into the page when the event is invoked.

The `NotificationReceived` event handler casts its event arguments to `NotificationEventArgs`. This type is defined in the shared `Xamarin.Forms` project:

```

public class NotificationEventArgs : EventArgs
{
    public string Title { get; set; }
    public string Message { get; set; }
}

```

For more information about the `Xamarin.Forms DependencyService`, see [Xamarin.Forms DependencyService](#).

Create the Android interface implementation

For the Xamarin.Forms application to send and receive notifications on Android, the application must provide an implementation of the `INotificationManager` interface.

Create the `AndroidNotificationManager` class

The `AndroidNotificationManager` class implements the `INotificationManager` interface:

```
using System;
using Android.App;
using Android.Content;
using Android.Graphics;
using Android.OS;
using AndroidX.Core.App;
using Xamarin.Forms;
using AndroidApp = Android.App.Application;

[assembly: Dependency(typeof(LocalNotifications.Droid.AndroidNotificationManager))]
namespace LocalNotifications.Droid
{
    public class AndroidNotificationManager : INotificationManager
    {
        const string channelId = "default";
        const string channelName = "Default";
        const string channelDescription = "The default channel for notifications.";

        public const string TitleKey = "title";
        public const string MessageKey = "message";

        bool channelInitialized = false;
        int messageId = 0;
        int pendingIntentId = 0;

        NotificationManager manager;

        public event EventHandler NotificationReceived;

        public static AndroidNotificationManager Instance { get; private set; }

        public AndroidNotificationManager() => Initialize();

        public void Initialize()
        {
            if (Instance == null)
            {
                CreateNotificationChannel();
                Instance = this;
            }
        }

        public void SendNotification(string title, string message, DateTime? notifyTime = null)
        {
            if (!channelInitialized)
            {
                CreateNotificationChannel();
            }

            if (notifyTime != null)
            {
                Intent intent = new Intent(AndroidApp.Context, typeof(AlarmHandler));
                intent.PutExtra(TitleKey, title);
                intent.PutExtra(MessageKey, message);

                PendingIntent pendingIntent = PendingIntent.GetBroadcast(AndroidApp.Context,
                pendingIntentId++, intent, PendingIntentFlags.CancelCurrent);
                long triggerTime = GetNotifyTime(notifyTime.Value);
            }
        }
    }
}
```

```

        AlarmManager alarmManager = AndroidApp.Context.GetService(Context.AlarmService) as
AlarmManager;
        alarmManager.Set(AlarmType.RtcWakeup, triggerTime, pendingIntent);
    }
    else
    {
        Show(title, message);
    }
}

public void ReceiveNotification(string title, string message)
{
    var args = new NotificationEventArgs()
    {
        Title = title,
        Message = message,
    };
    NotificationReceived?.Invoke(null, args);
}

public void Show(string title, string message)
{
    Intent intent = new Intent(AndroidApp.Context, typeof(MainActivity));
    intent.PutExtra>TitleKey, title);
    intent.PutExtra(MessageKey, message);

    PendingIntent pendingIntent = PendingIntent.GetActivity(AndroidApp.Context, pendingIntentId++,
intent, PendingIntentFlags.UpdateCurrent);

    NotificationCompat.Builder builder = new NotificationCompat.Builder(AndroidApp.Context,
channelId)
        .SetContentIntent(pendingIntent)
        .SetContentTitle(title)
        .SetContentText(message)
        .SetLargeIcon(BitmapFactory.DecodeResource(AndroidApp.Context.Resources,
Resource.Drawable.xamagonBlue))
        .SetSmallIcon(Resource.Drawable.xamagonBlue)
        .SetDefaults((int)NotificationDefaults.Sound | (int)NotificationDefaults.Vibrate);

    Notification notification = builder.Build();
    manager.Notify(messageId++, notification);
}

void CreateNotificationChannel()
{
    manager =
(NotificationManager)AndroidApp.Context.GetService(AndroidApp.NotificationService);

    if (Build.VERSION.SdkInt >= BuildVersionCodes.O)
    {
        var channelNameJava = new Java.Lang.String(channelName);
        var channel = new NotificationChannel(channelId, channelNameJava,
NotificationImportance.Default)
        {
            Description = channelDescription
        };
        manager.CreateNotificationChannel(channel);
    }

    channelInitialized = true;
}

long GetNotifyTime(DateTime notifyTime)
{
    DateTime utcTime = TimeZoneInfo.ConvertTimeToUtc(notifyTime);
    double epochDiff = (new DateTime(1970, 1, 1) - DateTime.MinValue).TotalSeconds;
    long utcAlarmTime = utcTime.AddSeconds(-epochDiff).Ticks / 10000;
    return utcAlarmTime; // milliseconds
}

```

```
    }  
}
```

The `assembly` attribute above the namespace registers the `INotificationManager` interface implementation with the `DependencyService`.

Android allows applications to define multiple channels for notifications. The `Initialize` method creates a basic channel the sample application uses to send notifications. The `SendNotification` method defines the platform-specific logic required to create and send a notification. The `ReceiveNotification` method is called by the Android OS when a message is received, and invokes the event handler.

The `SendNotification` method creates a local notification immediately, or at an exact `DateTime`. A notification can be scheduled for an exact `DateTime` using the `AlarmManager` class, and the notification will be received by an object that derives from the `BroadcastReceiver` class:

```
[BroadcastReceiver(Enabled = true, Label = "Local Notifications Broadcast Receiver")]  
public class AlarmHandler : BroadcastReceiver  
{  
    public override void OnReceive(Context context, Intent intent)  
    {  
        if (intent?.Extras != null)  
        {  
            string title = intent.GetStringExtra(AndroidNotificationManager.TitleKey);  
            string message = intent.GetStringExtra(AndroidNotificationManager.MessageKey);  
  
            AndroidNotificationManager manager = AndroidNotificationManager.Instance ?? new  
AndroidNotificationManager();  
            manager.Show(title, message);  
        }  
    }  
}
```

IMPORTANT

By default, notifications scheduled using the `AlarmManager` class will not survive device restart. However, you can design your application to automatically reschedule notifications if the device is restarted. For more information, [Start an alarm when the device restarts](#) in [Schedule repeating alarms](#) on developer.android.com, and the [sample](#). For information about background processing on Android, see [Guide to background processing](#) on developer.android.com.

For more information about broadcast receivers, see [Broadcast Receivers in Xamarin.Android](#).

Handle incoming notifications on Android

The `MainActivity` class must detect incoming notifications and notify the `AndroidNotificationManager` instance. The `Activity` attribute on the `MainActivity` class should specify a `LaunchMode` value of `LaunchMode.SingleTop`:

```
[Activity(  
    //...  
    LaunchMode = LaunchMode.SingleTop]  
public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity  
{  
    // ...  
}
```

The `SingleTop` mode prevents multiple instances of an `Activity` from being started while the application is in the foreground. This `LaunchMode` may not be appropriate for applications that launch multiple activities in more complex notification scenarios. For more information about `LaunchMode` enumeration values, see [Android Activity LaunchMode](#).

In the `MainActivity` class is modified to receive incoming notifications:

```
protected override void OnCreate(Bundle savedInstanceState)
{
    // ...

    global::Xamarin.Forms.Forms.Init(this, savedInstanceState);
    LoadApplication(new App());
    CreateNotificationFromIntent(Intent);
}

protected override void OnNewIntent(Intent intent)
{
    CreateNotificationFromIntent(intent);
}

void CreateNotificationFromIntent(Intent intent)
{
    if (intent?.Extras != null)
    {
        string title = intent.GetStringExtra(AndroidNotificationManager.TitleKey);
        string message = intent.GetStringExtra(AndroidNotificationManager.MessageKey);
        DependencyService.Get<INotificationManager>().ReceiveNotification(title, message);
    }
}
```

The `CreateNotificationFromIntent` method extracts notification data from the `intent` argument and provides it to the `AndroidNotificationManager` using the `ReceiveNotification` method. The `CreateNotificationFromIntent` method is called from both the `OnCreate` method and the `OnNewIntent` method:

- When the application is started by notification data, the `Intent` data will be passed to the `OnCreate` method.
- If the application is already in the foreground, the `Intent` data will be passed to the `OnNewIntent` method.

Android offers many advanced options for notifications. For more information, see [Notifications in Xamarin.Android](#).

Create the iOS interface implementation

For the `Xamarin.Forms` application to send and receive notifications on iOS, the application must provide an implementation of the `INotificationManager`.

Create the `iOSNotificationManager` class

The `iOSNotificationManager` class implements the `INotificationManager` interface:

```
using System;
using Foundation;
using UserNotifications;
using Xamarin.Forms;

[assembly: Dependency(typeof(LocalNotifications.iOS.iOSNotificationManager))]
namespace LocalNotifications.iOS
{
    public class iOSNotificationManager : INotificationManager
    {
        int messageId = 0;
        bool hasNotificationsPermission;
        public event EventHandler NotificationReceived;

        public void Initialize()
        {
            // request the permission to use local notifications
            UNUserNotificationCenter.Current.RequestAuthorization(UNAuthorizationOptions.Alert, (approved,
```

```

    err) =>
    {
        hasNotificationsPermission = approved;
    });
}

public void SendNotification(string title, string message, DateTime? notifyTime = null)
{
    // EARLY OUT: app doesn't have permissions
    if (!hasNotificationsPermission)
    {
        return;
    }

    messageId++;

    var content = new UNMutableNotificationContent()
    {
        Title = title,
        Subtitle = "",
        Body = message,
        Badge = 1
    };

    UNNotificationTrigger trigger;
    if (notifyTime != null)
    {
        // Create a calendar-based trigger.
        trigger = UNCalendarNotificationTrigger.CreateTrigger(GetNSDateComponents(notifyTime.Value),
false);
    }
    else
    {
        // Create a time-based trigger, interval is in seconds and must be greater than 0.
        trigger = UNTimeIntervalNotificationTrigger.CreateTrigger(0.25, false);
    }

    var request = UNNotificationRequest.FromIdentifier(messageId.ToString(), content, trigger);
    UNUserNotificationCenter.Current.AddNotificationRequest(request, (err) =>
    {
        if (err != null)
        {
            throw new Exception($"Failed to schedule notification: {err}");
        }
    });
}

public void ReceiveNotification(string title, string message)
{
    var args = new NotificationEventArgs()
    {
        Title = title,
        Message = message
    };
    NotificationReceived?.Invoke(null, args);
}

NSDateComponents GetNSDateComponents(DateTime dateTime)
{
    return new NSDateComponents
    {
        Month = dateTime.Month,
        Day = dateTime.Day,
        Year = dateTime.Year,
        Hour = dateTime.Hour,
        Minute = dateTime.Minute,
        Second = dateTime.Second
    };
}

```

```
    }  
}
```

The `assembly` attribute above the namespace registers the `INotificationManager` interface implementation with the `DependencyService`.

On iOS, you must request permission to use notifications before attempting to schedule a notification. The `Initialize` method requests authorization to use local notifications. The `SendNotification` method defines the logic required to create and send a notification. The `ReceiveNotification` method will be called by iOS when a message is received, and invokes the event handler.

NOTE

The `SendNotification` method creates a local notification immediately, using a `UNTimeIntervalNotificationTrigger` object, or at an exact `DateTime` using a `UNCalendarNotificationTrigger` object.

Handle incoming notifications on iOS

On iOS, you must create a delegate that subclasses `UNUserNotificationCenterDelegate` to handle incoming messages. The sample application defines an `iOSNotificationReceiver` class:

```
public class iOSNotificationReceiver : UNUserNotificationCenterDelegate  
{  
    public override void WillPresentNotification(UNUserNotificationCenter center, UNNotification notification, Action<UNNotificationPresentationOptions> completionHandler)  
    {  
        ProcessNotification(notification);  
        completionHandler(UNNotificationPresentationOptions.Alert);  
    }  
  
    void ProcessNotification(UNNotification notification)  
    {  
        string title = notification.Request.Content.Title;  
        string message = notification.Request.Content.Body;  
  
        DependencyService.Get<INotificationManager>().ReceiveNotification(title, message);  
    }  
}
```

This class uses the `DependencyService` to get an instance of the `iOSNotificationManager` class and provides incoming notification data to the `ReceiveNotification` method.

The `AppDelegate` class must specify an `iOSNotificationReceiver` object as the `UNUserNotificationCenter` delegate during application startup. This occurs in the `FinishedLaunching` method:

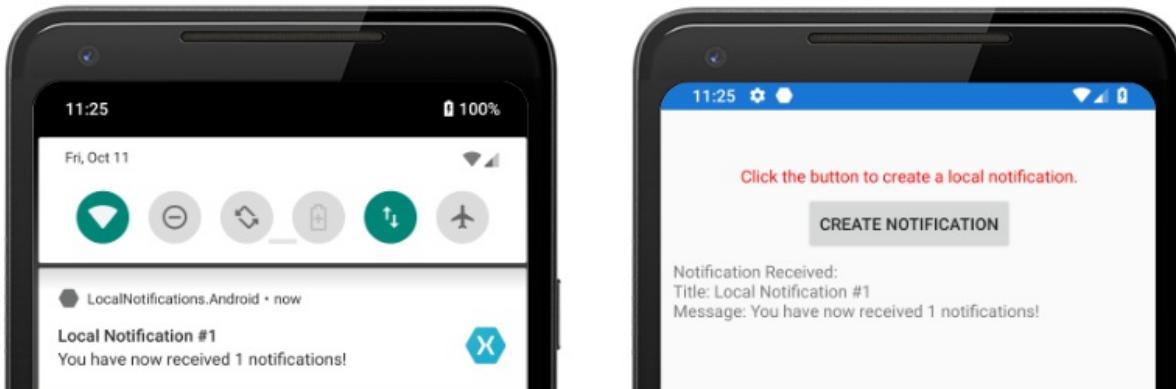
```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)  
{  
    global::Xamarin.Forms.Forms.Init();  
  
    UNUserNotificationCenter.Current.Delegate = new iOSNotificationReceiver();  
  
    LoadApplication(new App());  
    return base.FinishedLaunching(app, options);  
}
```

iOS offers many advanced options for notifications. For more information, see [Notifications in Xamarin.iOS](#).

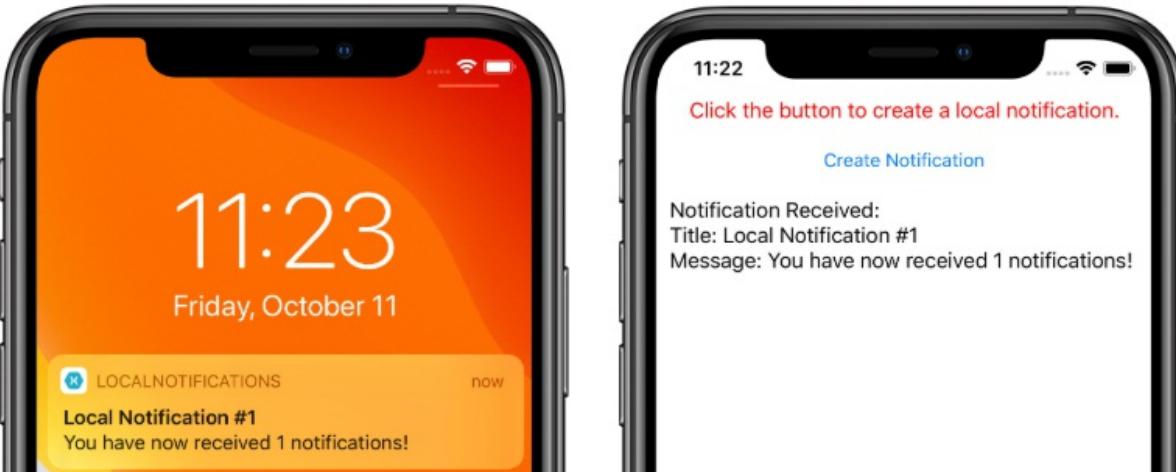
Test the application

Once the platform projects contain a registered implementation of the `INotificationManager` interface, the application can be tested on both platforms. Run the application and click either of the **Create Notification** buttons to create a notification.

On Android, notifications will appear in the notification area. When the notification is tapped, the application receives the notification and displays a message:



On iOS, incoming notifications are automatically received by the application without requiring user input. The application receives the notification and displays a message:



Related links

- [Sample project](#)
- [Notifications in Xamarin.Android](#)
- [Broadcast Receivers in Xamarin.Android](#)
- [Notifications in Xamarin.iOS](#)
- [Xamarin.Forms DependencyService](#)

Xamarin.Forms Localization

8/4/2022 • 2 minutes to read • [Edit Online](#)

The built-in .NET localization framework can be used to build cross-platform multilingual applications with Xamarin.Forms.

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

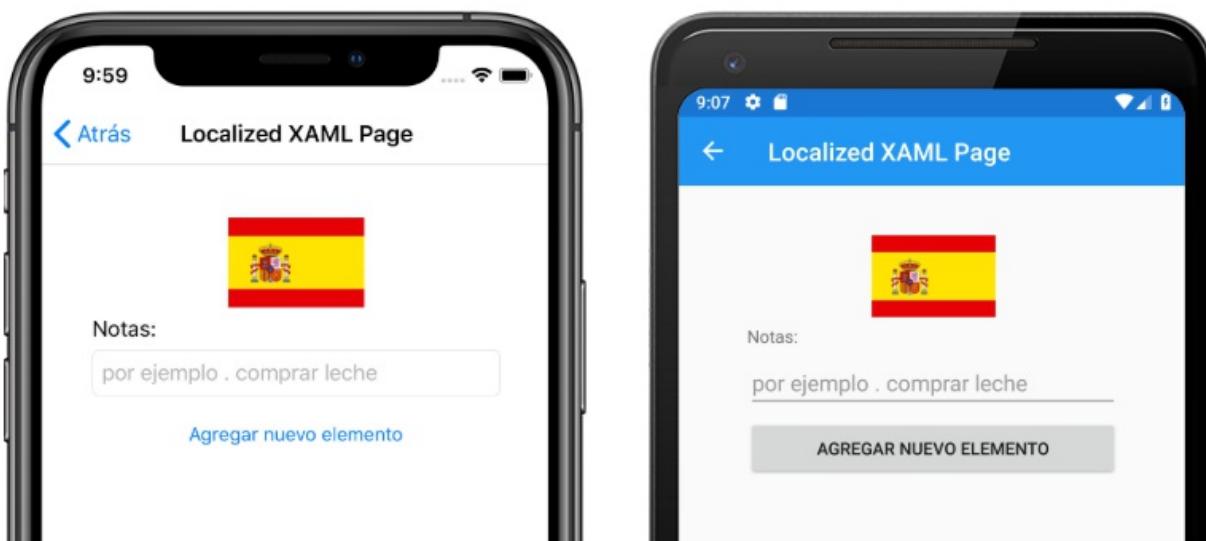
Flow direction is the direction in which the UI elements on the page are scanned by the eye. Right-to-left localization adds support for right-to-left flow direction to Xamarin.Forms applications.

Xamarin.Forms String and Image Localization

8/4/2022 • 9 minutes to read • [Edit Online](#)

 [Download the sample](#)

Localization is the process of adapting an application to meet the specific language or cultural requirements of a target market. To accomplish localization, the text and images in an application may need to be translated into multiple languages. A localized application automatically displays translated text based on the culture settings of the mobile device:



The .NET framework includes a built-in mechanism for localizing applications using [Resx resource files](#). A resource file stores text and other content as name/value pairs that allow the application to retrieve content for a provided key. Resource files allow localized content to be separated from application code.

Using resource files to localize Xamarin.Forms applications requires you to perform the following steps:

1. [Create Resx files](#) containing translated text.
2. [Specify the default culture](#) in the shared project.
3. [Localize text in Xamarin.Forms](#).
4. [Localize images](#) based on culture settings for each platform.
5. [Localize the application name](#) on each platform.
6. [Test localization](#) on each platform.

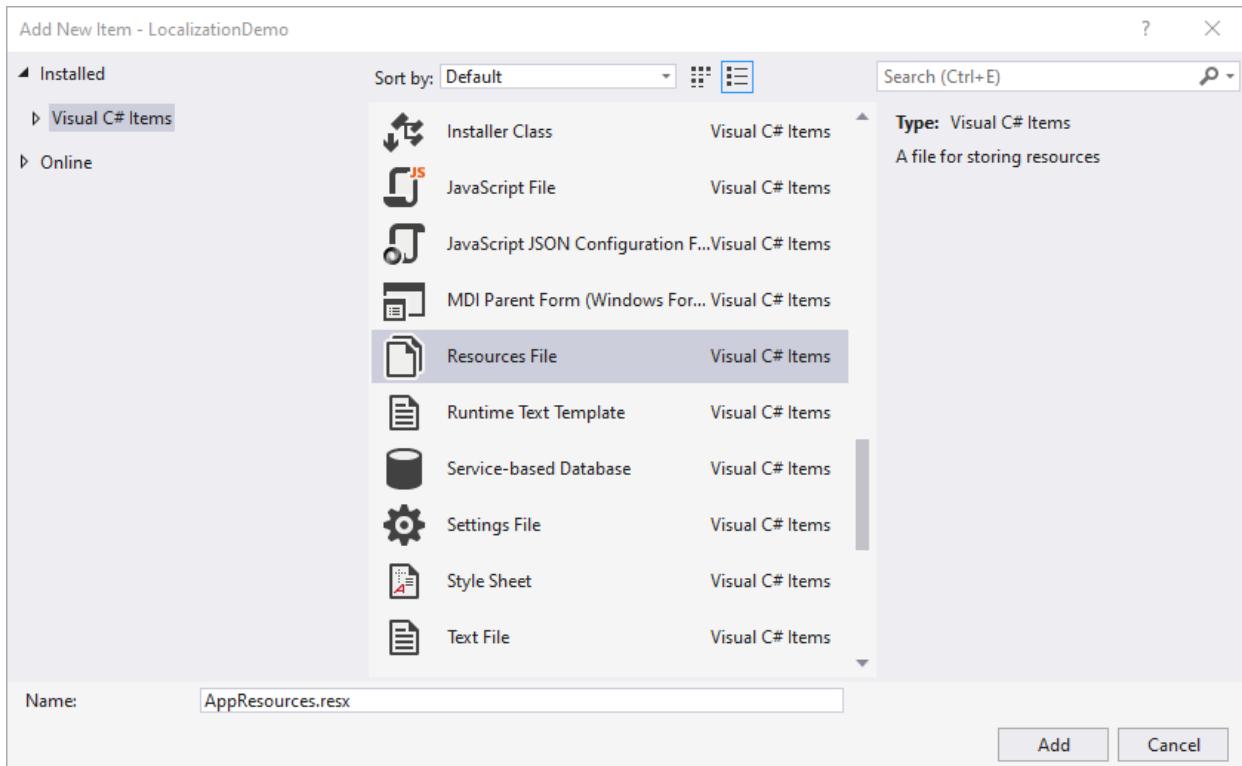
Create Resx files

Resource files are XML files with a `.resx` extension that are compiled into binary resource (`.resources`) files during the build process. Visual Studio 2019 generates a class that provides an API used to retrieve resources. A localized application typically contains a default resource file with all strings used in the application, as well as resource files for each supported language. The sample application has a `Resx` folder in the shared project that contains the resource files, and its default resource file called `AppResources.resx`.

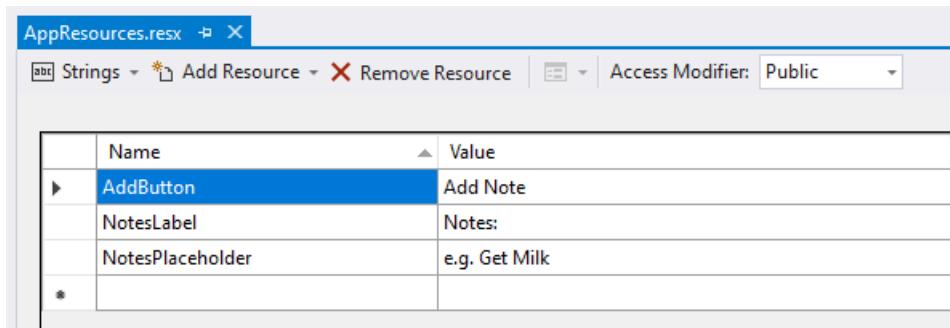
Resource files contain the following information for each item:

- **Name** specifies the key used to access the text in code.
- **Value** specifies the translated text.
- **Comment** is an optional field containing additional information.

A resource file is added with the **Add New Item** dialog in Visual Studio 2019:



Once the file is added, rows can be added for each text resource:



The **Access Modifier** drop down setting determines how Visual Studio generates the class used to access resources. Setting the Access Modifier to **Public** or **Internal** results in a generated class with the specified accessibility level. Setting the Access Modifier to **No code generation** does not generate a class file. The default resource file should be configured to generate a class file, which results in a file with the **.designer.cs** extension being added to the project.

Once the default resource file is created, additional files can be created for each culture the application supports. Each additional resource file should include the translation culture in the filename and should have the **Access Modifier** set to **No code generation**.

At runtime, the application attempts to resolve a resource request in order of specificity. For example, if the device culture is **en-US** the application looks for resource files in this order:

1. AppResources.en-US.resx
2. AppResources.en.resx
3. AppResources.resx (default)

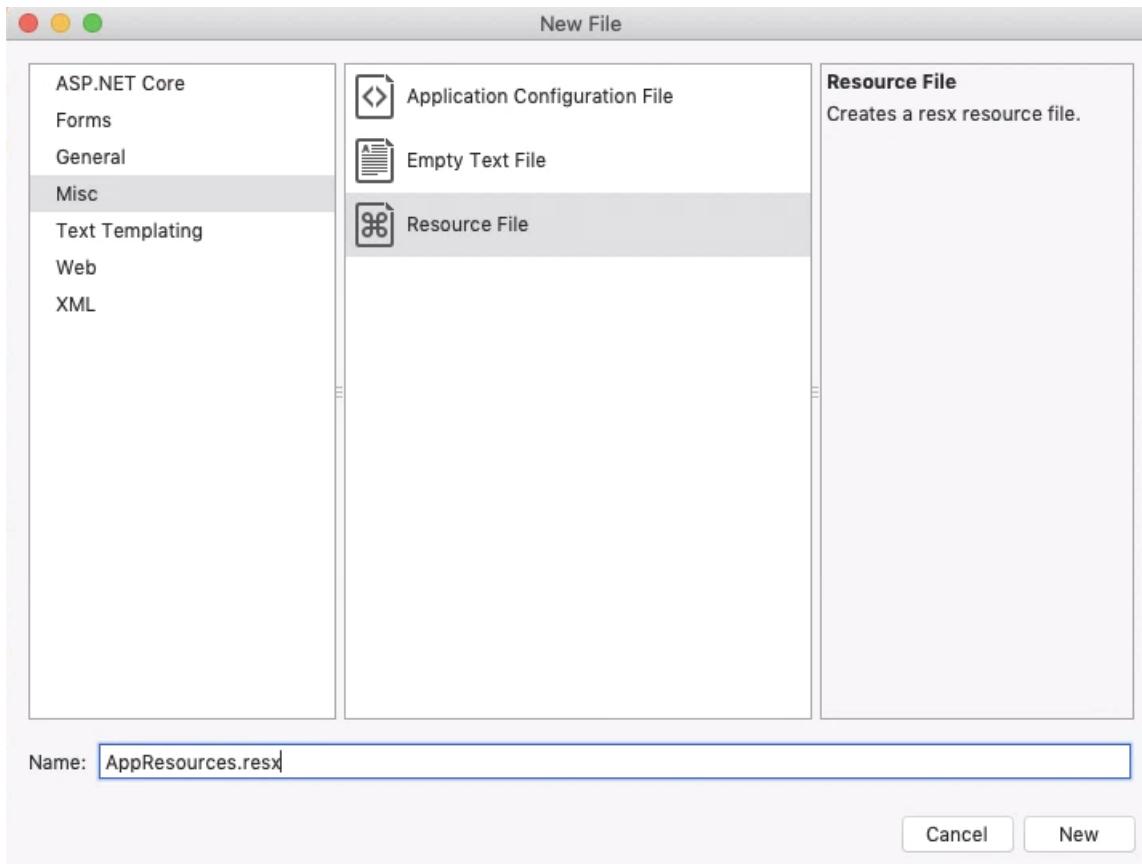
The following screenshot shows a Spanish translation file named **AppResources.es.resx**:

AppResources.es.resx

	Name	Value
▶	AddButton	Agregar nuevo elemento
	NotesLabel	Notas:
	NotesPlaceholder	por ejemplo . comprar leche
*		

The translation file uses the same **Name** values specified in the default file but contains Spanish language strings in the **Value** column. Additionally, the **Access Modifier** is set to **No code generation**.

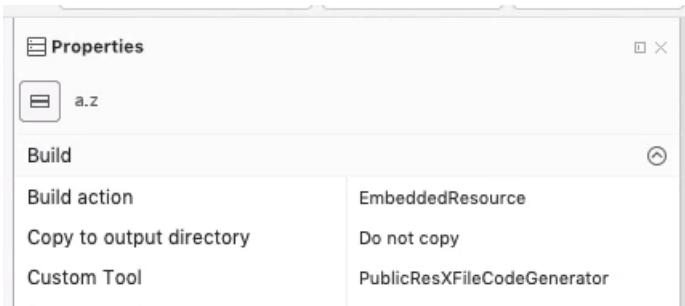
A resource file is added with the **Add New File** dialog in Visual Studio 2019 for Mac:



Once a default resource file has been created, text can be added by creating `<data>` elements within the `<root>` element in the resource file:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  ...
  <data name="AddButton" xml:space="preserve">
    <value>Add Note</value>
  </data>
  <data name="NotesLabel" xml:space="preserve">
    <value>Notes:</value>
  </data>
  <data name="NotesPlaceholder" xml:space="preserve">
    <value>e.g. Get Milk</value>
  </data>
</root>
```

A `.designer.cs` class file can be created by setting a **Custom Tool** property in the resource file options:



Setting the **Custom Tool** to **PublicResXFileCodeGenerator** will result in generated class with **public** access. Setting the **Custom Tool** to **InternalResXFileCodeGenerator** will result in a generated class with **internal** access. An empty **Custom Tool** value will not generate a class. The generated class name will match the resource file name. For example, the **AppResources.resx** file will result in the creation of an **AppResources** class in a file called **AppResources.designer.cs**.

Additional resource files can be created for each supported culture. Each language file should include the translation culture in the filename so a file targeting **es-MX** should be named **AppResources.es-MX.resx**.

At runtime, the application attempts to resolve a resource request in order of specificity. For example, if the device culture is **en-US** the application looks for resource files in this order:

1. AppResources.en-US.resx
2. AppResources.en.resx
3. AppResources.resx (default)

Language translation files should have the same **Name** values specified as the default file. The following XML shows the Spanish translation file named **AppResources.es.resx**:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  ...
  <data name="NotesLabel" xml:space="preserve">
    <value>Notas:</value>
  </data>
  <data name="NotesPlaceholder" xml:space="preserve">
    <value>por ejemplo . comprar leche</value>
  </data>
  <data name="AddButton" xml:space="preserve">
    <value>Agregar nuevo elemento</value>
  </data>
</root>
```

Specify the default culture

For resource files to work correctly, the application must have an **NeutralResourcesLanguage** attribute specified. In the project containing the resource files, the **AssemblyInfo.cs** file should be customized to specify the default culture. The following code shows how to set the **NeutralResourcesLanguage** to **en-US** in the **AssemblyInfo.cs** file:

```
using System.Resources;

// The resources from the neutral language .resx file are stored directly
// within the library assembly. For that reason, changing en-US to a different
// language in this line will not by itself change the language shown in the
// app. See the discussion of UltimateResourceFallbackLocation in the
// documentation for additional information:
// https://docs.microsoft.com/dotnet/api/system.resources.neutralresourceslanguageattribute
[assembly: NeutralResourcesLanguage("en-US")]
```

WARNING

If you do not specify the `NeutralResourcesLanguage` attribute, the `ResourceManager` class returns `null` values for any cultures without a specific resource file. When the default culture is specified, the `ResourceManager` returns results from the default Resx file for unsupported cultures. Therefore, it is recommended that you always specify the `NeutralResourcesLanguage` so that text is displayed for unsupported cultures.

Once a default resource file has been created and the default culture specified in the `AssemblyInfo.cs` file, the application can retrieve localized strings at runtime.

For more information about resource files, see [Create resource files for .NET apps](#).

Specify supported languages on iOS

On iOS, you must declare all supported languages in the `Info.plist` file for your project. In the `Info.plist` file, use the `Source` view to set an array for the `CFBundleLocalizations` key, and provide values that correspond to the Resx files. In addition, ensure you set an expected language via the `CFBundleDevelopmentRegion` key:

▼ Localizations	▽ Array	(9 items)
	String	de
	String	es
	String	fr
	String	ja
	String	pt
	String	pt-PT
	String	ru
	String	zh-Hans
	String	zh-Hant

Add new entry

Localization native development region ▽ String en

Alternatively, open the `Info.plist` file in an XML editor and add the following:

```
<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>
```

NOTE

Apple treats Portuguese slightly differently than you might expect. For more information, see [Adding Languages](#) on developer.apple.com.

For more information, see [Specifying default and supported languages in Info.plist](#).

Specify supported languages on UWP

This is only necessary if you generate an App Bundle when you package the app for sideloading or the store. When you generate a UWP App Bundle, when the bundle is installed, it will only load the resources related to the install device's language settings. Therefore, if the device only has English, then only English resources will be installed with the app. For more information and instructions, see [Windows 8.1 Store apps: Ensure that resources are installed on a device regardless of whether a device requires them](#).

Localize text in Xamarin.Forms

Text is localized in Xamarin.Forms using the generated `AppResources` class. This class is named based on the default resource file name. Since the sample project resource file is named `AppResources.cs`, Visual Studio generates a matching class called `AppResources`. Static properties are generated in the `AppResources` class for each row in the resource file. The following static properties are generated in the sample application's `AppResources` class:

- `AddButton`
- `NotesLabel`
- `NotesPlaceholder`

Accessing these values as `x:Static` properties allows localized text to be displayed in XAML:

```
<ContentPage ...
    xmlns:resources="clr-namespace:LocalizationDemo.Resx">
    <Label Text="{x:Static resources:AppResources.NotesLabel}" />
    <Entry Placeholder="{x:Static resources:AppResources.NotesPlaceholder}" />
    <Button Text="{x:Static resources:AppResources.AddButton}" />
</ContentPage>
```

Localized text can also be retrieved in code:

```
public LocalizedCodePage()
{
    Label notesLabel = new Label
    {
        Text = AppResources.NotesLabel,
        // ...
    };

    Entry notesEntry = new Entry
    {
        Placeholder = AppResources.NotesPlaceholder,
        //...
    };

    Button addButton = new Button
    {
        Text = AppResources.AddButton,
        // ...
    };

    Content = new StackLayout
    {
        Children = {
            notesLabel,
            notesEntry,
            addButton
        }
    };
}
```

The properties in the `AppResources` class use the current value of the `System.Globalization.CultureInfo.CurrentCulture` to determine which culture resource file to retrieve values from.

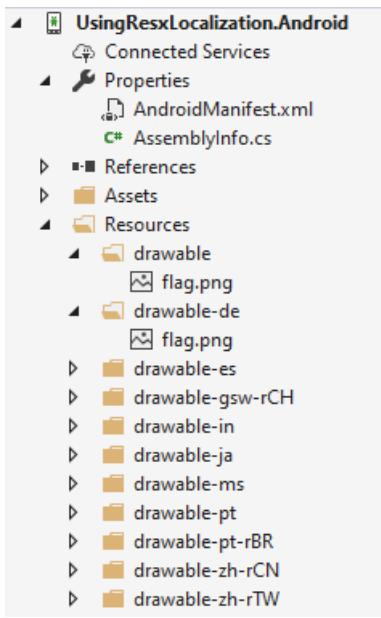
Localize images

In addition to storing text, Resx files are capable of storing more than just text, they can also store images and binary data. However, mobile devices have a range of screen sizes and densities and each mobile platform has functionality for displaying density-dependent images. Therefore, platform image localization functionality should be used instead of storing images in resource files.

Localize images on Android

On Android, localized drawables (images) are stored using a naming convention for folders in the `Resources` directory. Folders are named **drawable** with a suffix for the target language. For example, the Spanish-language folder is named **drawable-es**.

When a four-letter locale code is required, Android requires an additional **r** following the dash. For example, the Mexico locale (es-MX) folder should be named **drawable-es-rMX**. The image file names in each locale folder should be identical:

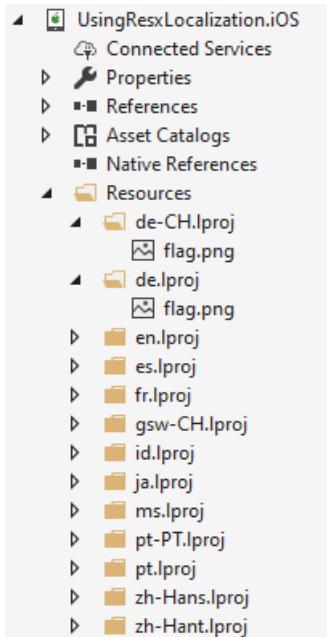


For more information, see [Android Localization](#).

Localize images on iOS

On iOS, localized images are stored using a naming convention for folders in the **Resources** directory. The default folder is named **Base.lproj**. Language-specific folders are named with the language or locale name, followed by **.lproj**. For example, the Spanish-language folder is named **es.lproj**.

Four-letter local codes work just like two-letter language codes. For example, the Mexico locale (es-MX) folder should be named **es-MX.lproj**. The image file names in each locale folder should be identical:



NOTE

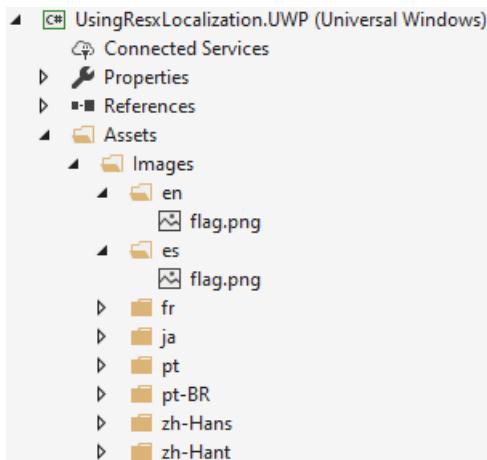
iOS supports creating a localized Asset Catalog instead of using the .lproj folder structure. However, these must be created and managed in Xcode.

For more information, see [iOS Localization](#).

Localize images on UWP

On UWP, localized images are stored using a naming convention for folders in the **Assets/Images** directory. Folders are named with the language or locale. For example, the Spanish-language folder is named **es** and the

Mexico locale folder should be named **es-MX**. The image file names in each locale folder should be identical:



For more information, see [UWP Localization](#).

Consume localized images

Since each platform stores images with a unique file structure, the XAML uses the `OnPlatform` class to set the `ImageSource` property based on the current platform:

```
<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>
```

NOTE

The `OnPlatform` markup extension offers a more concise way of specifying platform-specific values. For more information, see [OnPlatform markup extension](#).

The image source can be set based on the `Device.RuntimePlatform` property in code:

```
string imgSrc = Device.RuntimePlatform == Device.UWP ? "Assets/Images/flag.png" : "flag.png";
Image flag = new Image
{
    Source = ImageSource.FromFile(imgSrc),
    WidthRequest = 100
};
```

Localize the application name

The application name is specified per-platform and does not use Resx resource files. To localize the application name on Android, see [Localize app name on Android](#). To localize the application name on iOS, see [Localize app name on iOS](#). To localize the application name on UWP, see [Localize strings in the UWP package manifest](#).

Test localization

Testing localization is best accomplished by changing your device language. It is possible to set the value of `System.Globalization.CultureInfo.CurrentCulture` in code but behavior is inconsistent across platforms so this is not recommended for testing.

On iOS, in the settings app, you can set the language for each app specifically without changing your device language.

On Android, the language settings are detected and cached when the application starts. If you change languages, you may need to exit and restart the application to see the changes applied.

Related links

- [Localization Sample Project](#)
- [Create resource files for .NET apps](#)
- [Cross-Platform Localization](#)
- [Using the CultureInfo class \(MSDN\)](#)
- [Android Localization](#)
- [iOS Localization](#)
- [UWP Localization](#)
- [Locating and Using Resources for a Specific Culture \(MSDN\)](#)

Right-to-left localization

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

Right-to-left localization adds support for right-to-left flow direction to Xamarin.Forms applications.

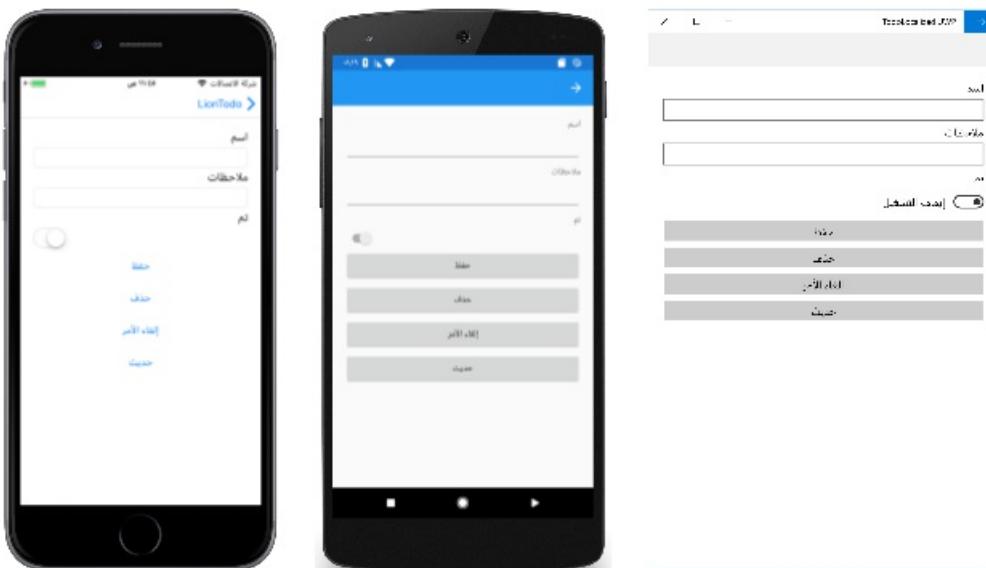
NOTE

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.
- `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 `FlyoutPage` children.
- `ContextActions` text alignment is controlled by the device locale, rather than the `FlowDirection` property.

Force right-to-left layout

Xamarin.iOS and Xamarin.Android applications can be forced to always use a right-to-left layout, regardless of device settings, by modifying the respective platform projects.

iOS

Xamarin.iOS applications can be forced to always use a right-to-left layout by modifying the `AppDelegate` class as follows:

1. Declare the `IntPtr_objc_msgSend` function as the first line in your `AppDelegate` class:

```
[System.Runtime.InteropServices.DllImport(ObjCRuntime.Constants.ObjectiveCLibrary, EntryPoint =
"objc_msgSend")]
internal extern static IntPtr IntPtr_objc_msgSend(IntPtr receiver, IntPtr selector,
UISemanticContentAttribute arg1);
```

2. Call the `IntPtr_objc_msgSend` function from the `FinishedLaunching` method, before returning from the `FinishedLaunching` method:

```
bool result = base.FinishedLaunching(app, options);

ObjCRuntime.Selector selector = new ObjCRuntime.Selector("setSemanticContentAttribute:");
IntPtr_objc_msgSend(UIView.Appearance.Handle, selector.Handle,
UISemanticContentAttribute.ForceRightToLeft);

return result;
```

This approach is useful for applications that always require a right-to-left layout, and removes the requirement to set the `FlowDirection` property.

For more information about the `IntPtr_objc_msgSend` method, see [Objective-C selectors in Xamarin.iOS](#).

Android

Xamarin.Android applications can be forced to always use a right-to-left layout by modifying the `MainActivity` class to include the following line:

```
Window.DecorView.LayoutDirection = LayoutDirection.Rtl;
```

NOTE

This approach requires that the application is setup to support right-to-left layout. For more information, see [Android platform setup](#).

This approach is useful for applications that always require a right-to-left layout, and removes the requirement to set the `FlowDirection` property for most controls. However, some controls, such as `CollectionView`, don't respect the `LayoutDirection` property and still require the `FlowDirection` property to be set.

Right to left language support with Xamarin.University

Related links

- [TodoLocalizedRTL Sample App](#)

Xamarin.Forms MessagingCenter

8/4/2022 • 4 minutes to read • [Edit Online](#)

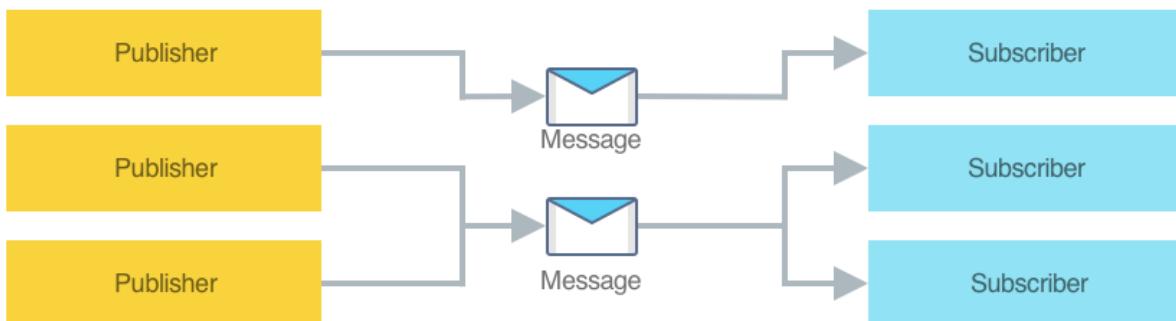
 [Download the sample](#)

The publish-subscribe pattern is a messaging pattern in which publishers send messages without having knowledge of any receivers, known as subscribers. Similarly, subscribers listen for specific messages, without having knowledge of any publishers.

Events in .NET implement the publish-subscribe pattern, and are the most simple and straightforward approach for a communication layer between components if loose coupling is not required, such as a control and the page that contains it. However, the publisher and subscriber lifetimes are coupled by object references to each other, and the subscriber type must have a reference to the publisher type. This can create memory management issues, especially when there are short lived objects that subscribe to an event of a static or long-lived object. If the event handler isn't removed, the subscriber will be kept alive by the reference to it in the publisher, and this will prevent or delay the garbage collection of the subscriber.

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 them.

The `MessagingCenter` class provides multicast publish-subscribe functionality. This means that there can be multiple publishers that publish a single message, and there can be multiple subscribers listening for the same message:



Publishers send messages using the `MessagingCenter.Send` method, while subscribers listen for messages using the `MessagingCenter.Subscribe` method. In addition, subscribers can also unsubscribe from message subscriptions, if required, with the `MessagingCenter.Unsubscribe` method.

IMPORTANT

Internally, the `MessagingCenter` class uses weak references. This means that it will not keep objects alive, and will allow them to be garbage collected. Therefore, it should only be necessary to unsubscribe from a message when a class no longer wishes to receive the message.

Publish a message

`MessagingCenter` messages are strings. Publishers notify subscribers of a message with one of the `MessagingCenter.Send` overloads. The following code example publishes a `Hi` message:

```
MessagingCenter.Send<MainPage>(this, "Hi");
```

In this example the `Send` method specifies a generic argument that represents the sender. To receive the message, a subscriber must also specify the same generic argument, indicating that they are listening for a message from that sender. In addition, this example specifies two method arguments:

- The first argument specifies the sender instance.
- The second argument specifies the message.

Payload data can also be sent with a message:

```
MessagingCenter.Send<MainPage, string>(this, "Hi", "John");
```

In this example, the `Send` method specifies two generic arguments. The first is the type that's sending the message, and the second is the type of the payload data being sent. To receive the message, a subscriber must also specify the same generic arguments. This enables multiple messages that share a message identity but send different payload data types to be received by different subscribers. In addition, this example specifies a third method argument that represents the payload data to be sent to the subscriber. In this case the payload data is a `string`.

The `Send` method will publish the message, and any payload data, using a fire-and-forget approach. Therefore, the message is sent even if there are no subscribers registered to receive the message. In this situation, the sent message is ignored.

Subscribe to a message

Subscribers can register to receive a message using one of the `MessagingCenter.Subscribe` overloads. The following code example shows an example of this:

```
MessagingCenter.Subscribe<MainPage> (this, "Hi", (sender) =>
{
    // Do something whenever the "Hi" message is received
});
```

In this example, the `Subscribe` method subscribes the `this` object to `Hi` messages that are sent by the `MainPage` type, and executes a callback delegate in response to receiving the message. The callback delegate, specified as a lambda expression, could be code that updates the UI, saves some data, or triggers some other operation.

NOTE

A subscriber might not need to handle every instance of a published message, and this can be controlled by the generic type arguments that are specified on the `Subscribe` method.

The following example shows how to subscribe to a message that contains payload data:

```
MessagingCenter.Subscribe<MainPage, string>(this, "Hi", async (sender, arg) =>
{
    await DisplayAlert("Message received", "arg=" + arg, "OK");
});
```

In this example, the `Subscribe` method subscribes to `Hi` messages that are sent by the `MainPage` type, whose

payload data is a `string`. A callback delegate is executed in response to receiving such a message, that displays the payload data in an alert.

IMPORTANT

The delegate that's executed by the `Subscribe` method will be executed on the same thread that publishes the message using the `Send` method.

Unsubscribe from a message

Subscribers can unsubscribe from messages they no longer want to receive. This is achieved with one of the `MessagingCenter.Unsubscribe` overloads:

```
MessagingCenter.Unsubscribe<MainPage>(this, "Hi");
```

In this example, the `Unsubscribe` method unsubscribes the `this` object from the `Hi` message sent by the `MainPage` type.

Messages containing payload data should be unsubscribed from using the `unsubscribe` overload that specifies two generic arguments:

```
MessagingCenter.Unsubscribe<MainPage, string>(this, "Hi");
```

In this example, the `Unsubscribe` method unsubscribes the `this` object from the `Hi` message sent by the `MainPage` type, whose payload data is a `string`.

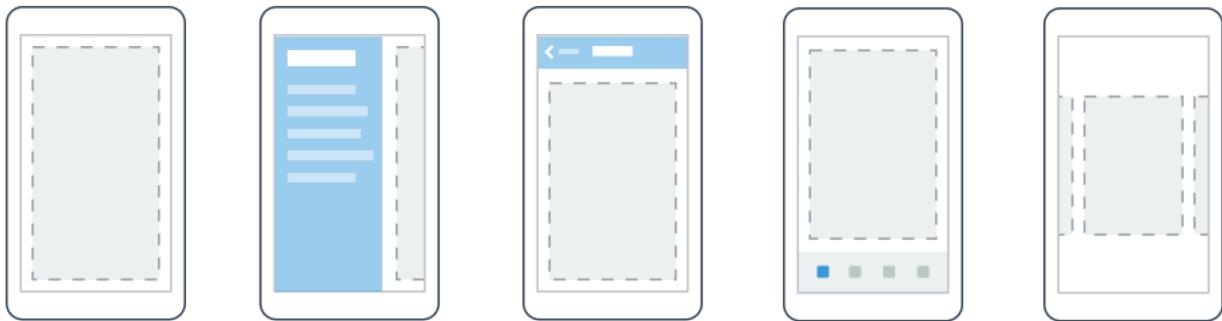
Related links

- [MessagingCenterSample](#)

Xamarin.Forms Navigation

8/4/2022 • 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

Alternatively, Xamarin.Forms Shell applications use a URI-based navigation experience that doesn't enforce a set navigation hierarchy. For more information, see [Xamarin.Forms Shell 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.

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.

FlyoutPage

The Xamarin.Forms `FlyoutPage` is a page that manages two pages of related information – a flyout page that presents items, and a detail page that presents details about items on the flyout 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.

Hierarchical Navigation

8/4/2022 • 10 minutes to read • [Edit Online](#)

 [Download the sample](#)

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:



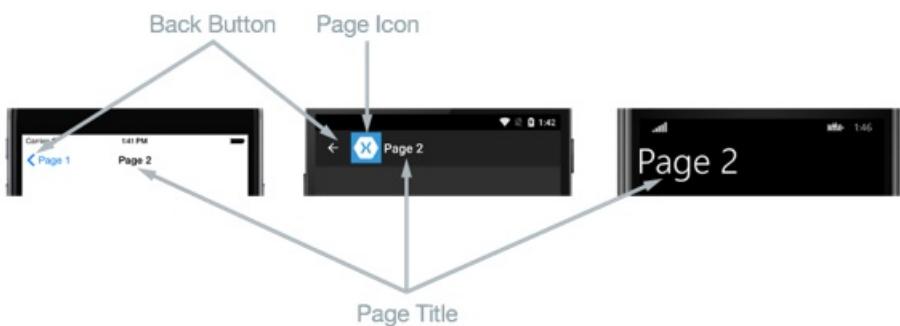
To return back to the previous page, the application will pop the current page from the navigation stack, and the new topmost page becomes the active page, as shown in the following diagram:



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. In addition, the `IconColor` property can be set to a `Color` that's applied to the icon in the navigation bar.

NOTE

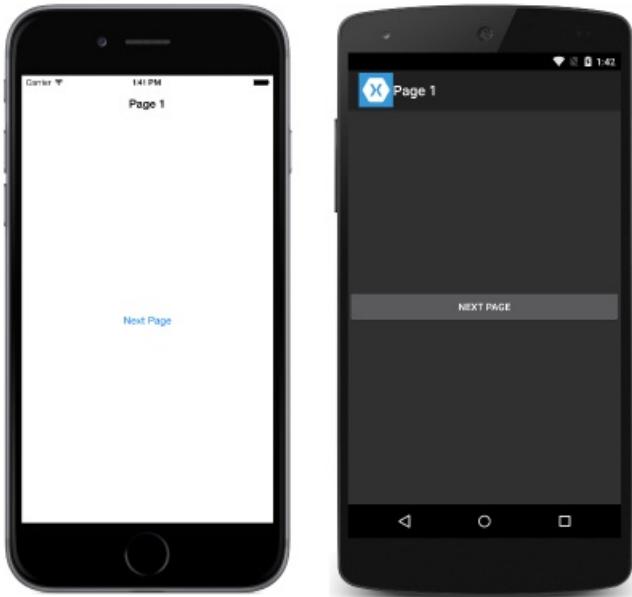
It's recommended that a `NavigationPage` should be populated with `ContentPage` instances only.

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.

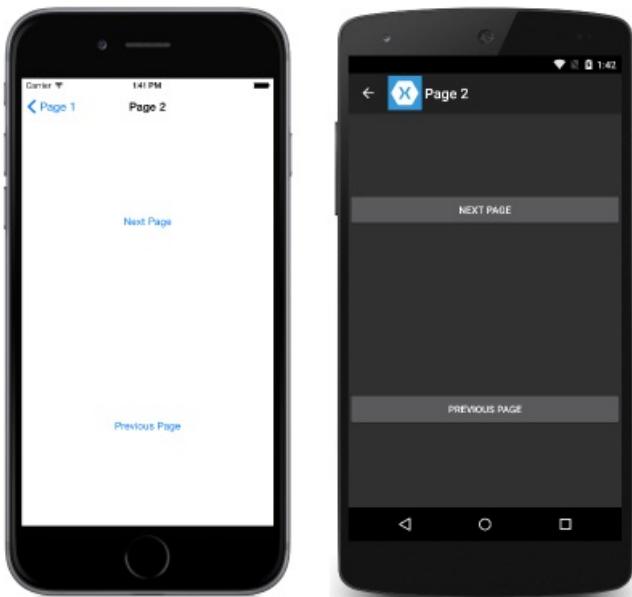
Pushing Pages to the Navigation Stack

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:

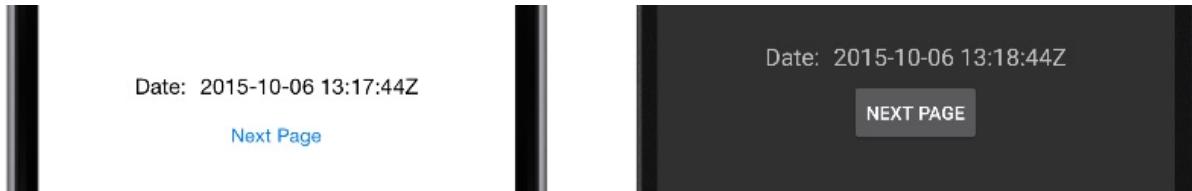
```
public App ()
{
    MainPage = new NavigationPage (new MainPage (DateTime.Now.ToString ("u")));
}
```

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.TitleViewPage">
    <NavigationPage.TitleView>
        <Slider HeightRequest="44" WidthRequest="300" />
    </NavigationPage.TitleView>
    ...
</ContentPage>
```

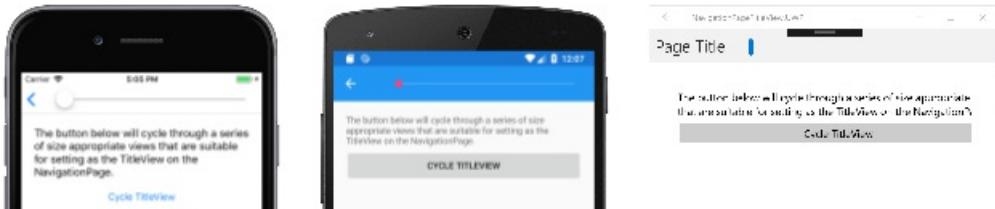
Here is the equivalent C# code:

```

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 `NavigationBar.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 `NavigationBar.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 \(chapter 24\)](#)
- [Hierarchical \(sample\)](#)
- [PassingData \(sample\)](#)
- [LoginFlow \(sample\)](#)
- [TitleView \(sample\)](#)
- [How to Create a Sign In Screen Flow in Xamarin.Forms video](#)
- [NavigationPage](#)

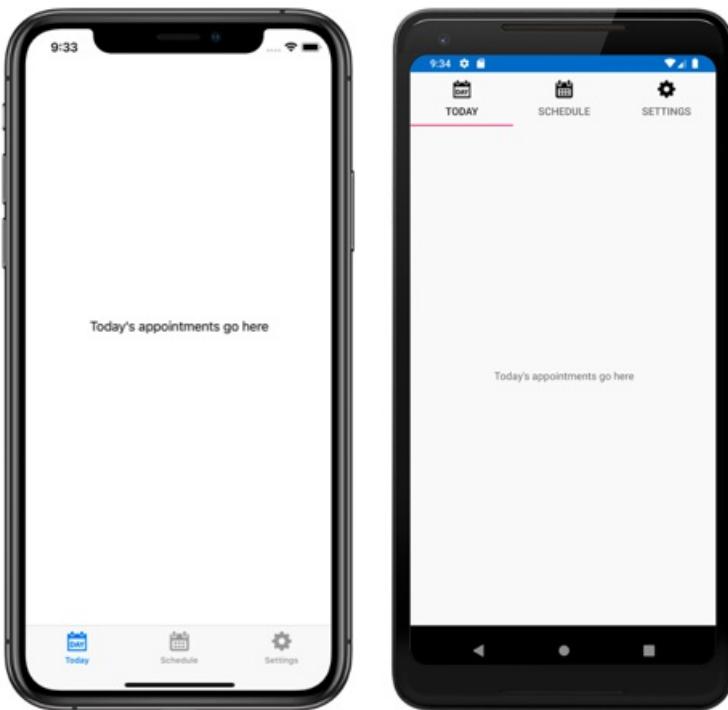
Xamarin.Forms TabbedPage

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

The Xamarin.Forms [TabbedPage](#) consists of a list of tabs and a larger detail area, with each tab loading content into the detail area. The following screenshots show a [TabbedPage](#) on iOS and Android:



On iOS, the list of tabs appears at the bottom of the screen, and the detail area is above. Each tab consists of a title and an icon, which should be a PNG file with an alpha channel. In portrait orientation, tab bar icons appear above tab titles. In landscape orientation, icons and titles appear side by side. In addition, a regular or compact tab bar may be displayed, depending on the device and orientation. If there are more than five tabs, a **More** tab will appear, which can be used to access the additional tabs. For information about icon requirements, see [Tab Bar Icon Size](#) on developer.apple.com.

On Android, the list of tabs appears at the top of the screen, and the detail area is below. Each tab consists of a title and an icon, which should be a PNG file with an alpha channel. However, the tabs can be moved to the bottom of the screen with a platform-specific. If there are more than five tabs, and the tab list is at the bottom of the screen, a *More* tab will appear that can be used to access the additional tabs. For information about icon requirements, see [Tabs](#) on material.io and [Support different pixel densities](#) on developer.android.com. For information about moving the tabs to the bottom of the screen, see [Setting TabbedPage Toolbar Placement and Color](#).

On the Universal Windows Platform (UWP), the list of tabs appears at the top of the screen, and the details area is below. Each tab consists of a title. However, icons can be added to each tab with a platform-specific. For more information, see [TabbedPage Icons on Windows](#).

TIP

Scalable Vector Graphic (SVG) files can be displayed as tab icons on a `TabPage`:

- The iOS `TabbedRenderer` class has an overridable `GetIcon` method that can be used to load tab icons from a specified source. In addition, selected and unselected versions of an icon can be provided if required.
- The Android AppCompat `TabPageRenderer` class has an overridable `SetTabIconImageSource` method that can be used to load tab icons from a custom `Drawable`. Alternatively, SVG files can be converted to vector drawable resources, which can automatically be displayed by Xamarin.Forms. For more information about converting SVG files to vector drawable resources, see [Add multi-density vector graphics](#) on developer.android.com.

For more information, see [Xamarin.Forms TabbedPage with SVG tab icons](#).

Create 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` objects. For more information, see [Populate a TabbedPage with a Page Collection](#).
- Assign a collection to the `ItemsSource` property and assign a `DataTemplate` to the `ItemTemplate` property to return pages for objects in the collection. For more information, see [Populate a TabbedPage with a template](#).

With both approaches, the `TabPage` will display each page as the user selects each tab.

IMPORTANT

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.

In addition, `TabPage` defines the following properties:

- `BarBackgroundColor`, of type `Color`, the background color of the tab bar.
- `BarTextColor`, of type `Color`, the color of text on the tab bar.
- `SelectedTabColor`, of type `Color`, the color of the tab when it's selected.
- `UnselectedTabColor`, of type `Color`, the color of the tab when it's unselected.

All of these properties are backed by `BindableProperty` objects, which means that they can be styled, and the properties can be the targets of data bindings.

WARNING

In a `TabPage`, each `Page` object is created when the `TabPage` is constructed. This can lead to a poor user experience, particularly if the `TabPage` is the root page of the application. However, Xamarin.Forms Shell enables pages accessed through a tab bar to be created on demand, in response to navigation. For more information, see [Xamarin.Forms Shell](#).

Populate a TabbedPage with a Page collection

A `TabPage` can be populated with a collection of child `Page` objects, such as a collection of `ContentPage` objects. This is achieved by adding the `Page` objects to the `TabPage.Children` collection. This is accomplished in XAML as follows:

```

<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 />
    <NavigationPage Title="Schedule" IconImageSource="schedule.png">
        <x:Arguments>
            <local:SchedulePage />
        </x:Arguments>
    </NavigationPage>
</TabbedPage>

```

NOTE

The `Children` property of the `MultiPage<T>` class, from which `TabbedPage` derives, is the `ContentProperty` of `MultiPage<T>`. Therefore, in XAML it's not necessary to explicitly assign the `Page` objects to the `Children` property.

The equivalent C# code is:

```

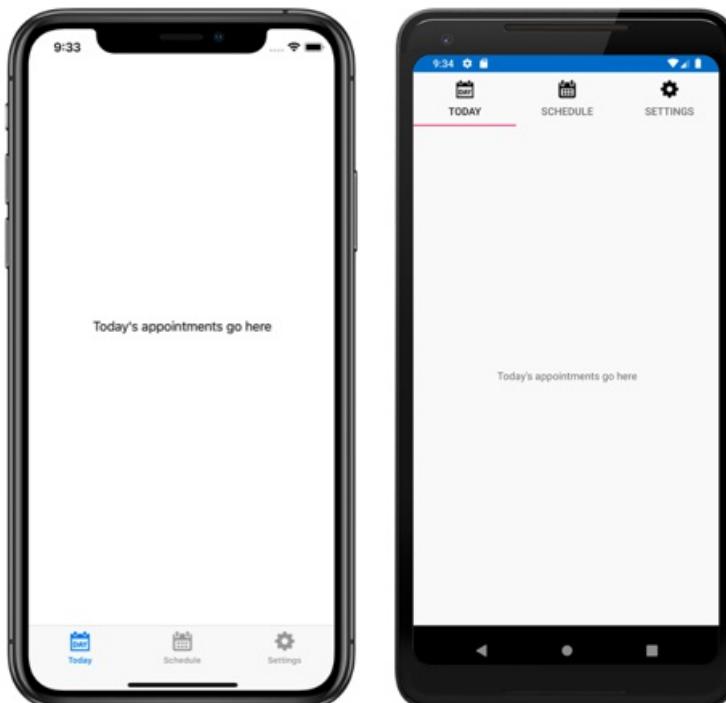
public class MainPageCS : TabbedPage
{
    public MainPageCS ()
    {
        NavigationPage navigationPage = new NavigationPage (new SchedulePageCS ());
        navigationPage.IconImageSource = "schedule.png";
        navigationPage.Title = "Schedule";

        Children.Add (new TodayPageCS ());
        Children.Add (navigationPage);
    }
}

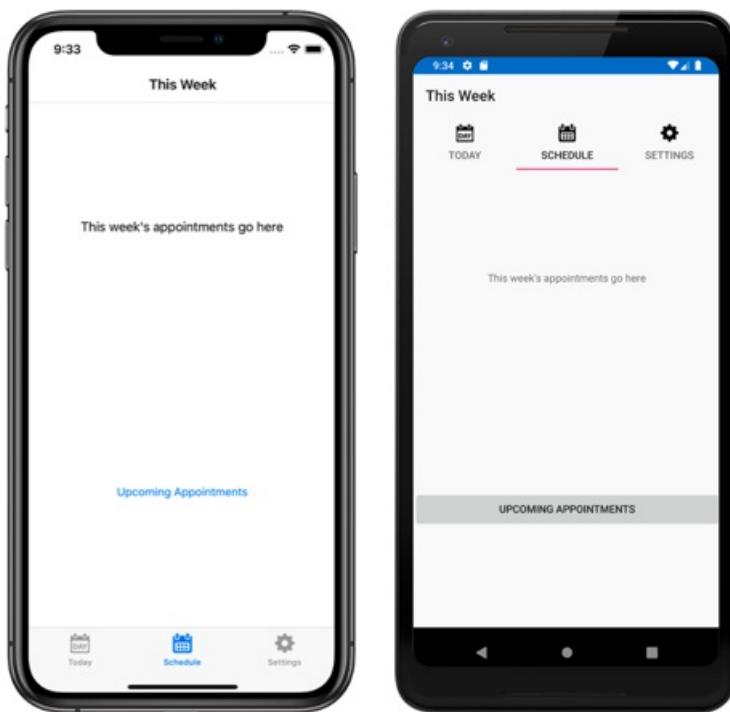
```

In this example, the `TabbedPage` is populated with two `Page` objects. The first child is a `ContentPage` object, and the second child is a `NavigationPage` containing a `ContentPage` object.

The following screenshots show a `ContentPage` object in a `TabbedPage`:



Selecting another tab displays the `ContentPage` object that represents the tab:



On the Schedule tab, the `ContentPage` object is wrapped in a `TabPage` object.

WARNING

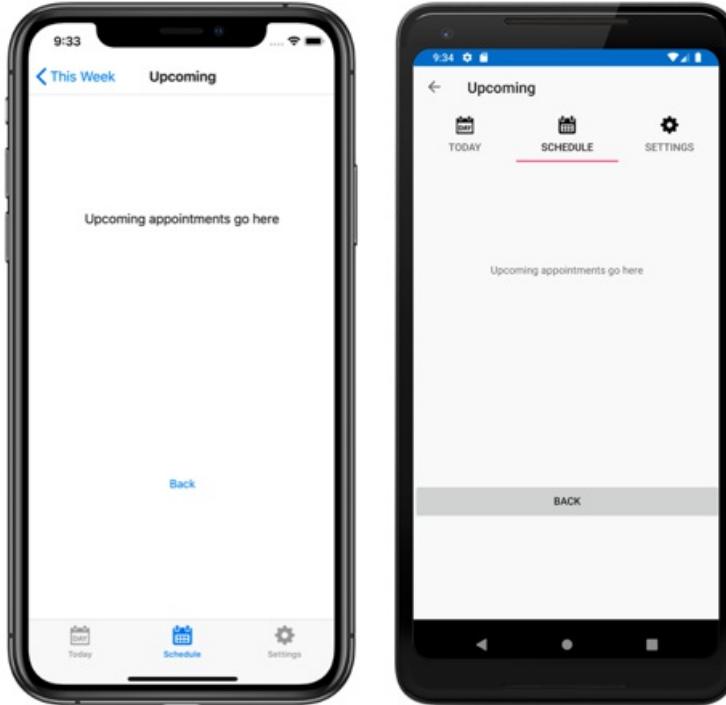
While a `TabPage` can be placed in a `TabbedPage`, it's not recommended to place a `TabbedPage` into a `TabPage`. This is because, on iOS, a `UITabBarController` always acts as a wrapper for the `UINavigationController`. For more information, see [Combined View Controller Interfaces](#) in the iOS Developer Library.

Navigate within a tab

Navigation can be performed within a tab, provided that the `ContentPage` object is wrapped in a `TabPage` object. This is accomplished by invoking the `PushAsync` method on the `Navigation` property of the `ContentPage` object:

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

The page being navigated to is specified as the argument to the `PushAsync` method. In this example, the `UpcomingAppointmentsPage` page is pushed onto the navigation stack, where it becomes the active page:



For more information about performing navigation using the [NavigationPage](#) class, see [Hierarchical Navigation](#).

Populate a TabbedPage with a template

A [TabbedPage](#) can be populated with pages by assigning a collection of data to the [ItemsSource](#) property, and by assigning a [DataTemplate](#) to the [ItemTemplate](#) property that templates the data as [Page](#) objects. This is accomplished in XAML as follows:

```
<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"
    ItemsSource="{x:Static local:MonkeyDataModel.All}">
<TabbedPage.Resources>
    <ResourceDictionary>
        <local:NonNullToBooleanConverter x:Key="booleanConverter" />
    </ResourceDictionary>
</TabbedPage.Resources>
<TabbedPage.ItemTemplate>
    <DataTemplate>
        <ContentPage Title="{Binding Name}" IconImageSource="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 equivalent C# code is:

```

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");

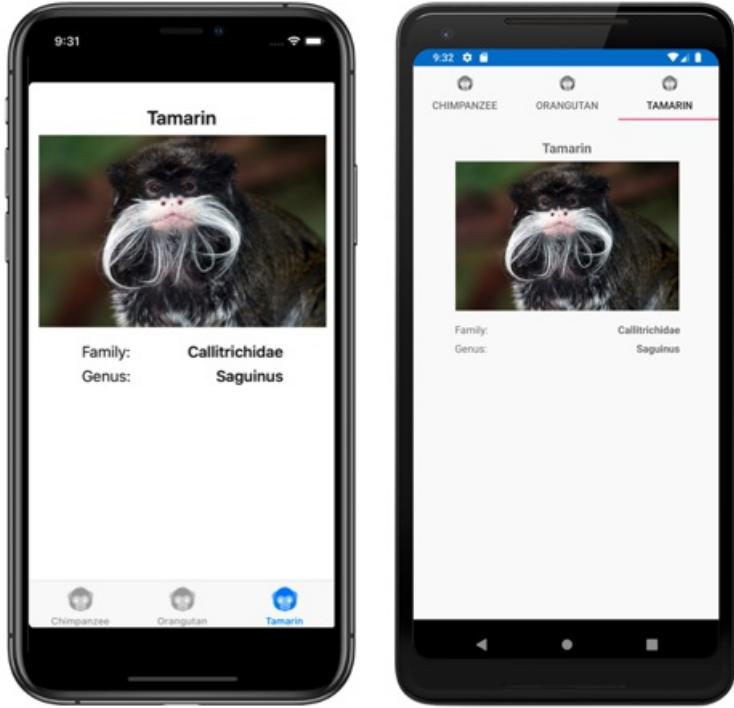
            var image = new Image { WidthRequest = 200, HeightRequest = 200 };
            image.SetBinding (Image.SourceProperty, "PhotoUrl");

            var familyLabel = new Label
            {
                FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
                FontAttributes = FontAttributes.Bold
            };
            familyLabel.SetBinding (Label.TextProperty, "Family");
            ...

            var contentPage = new ContentPage
            {
                IconImageSource = "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;
        }
    }
}

```

In this example, each tab consists of a `ContentPage` object that uses `Image` and `Label` objects to display data for the tab:



Selecting another tab displays the [ContentPage](#) object that represents the tab.

Related links

- [TabbedPageWithNavigationPage \(sample\)](#)
- [TabbedPage \(sample\)](#)
- [TabbedPage with SVG tab icons](#)
- [Hierarchical Navigation](#)
- [Page Varieties \(chapter 25\)](#)
- [TabbedPage API](#)

Xamarin.Forms Carousel Page

8/4/2022 • 3 minutes to read • [Edit Online](#)

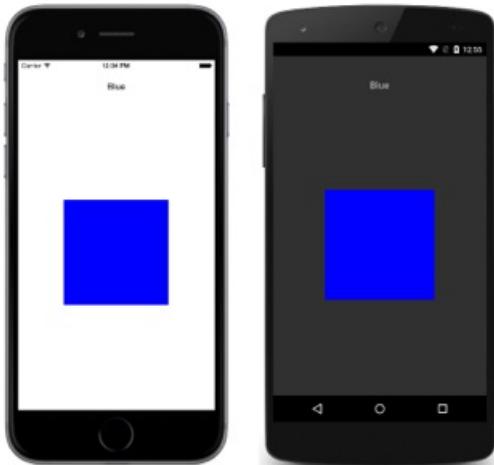
 [Download the sample](#)

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.

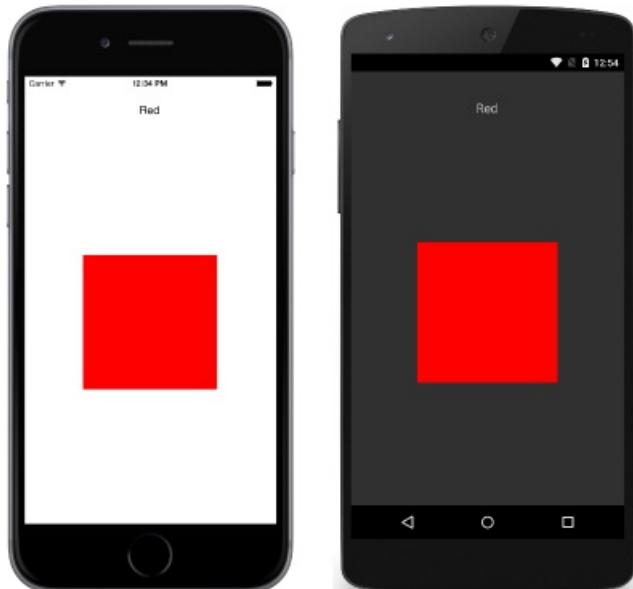
IMPORTANT

The `CarouselPage` has been superseded by the `CarouselView`, which provides a scrollable layout where users can swipe to move through a collection of items. For more information about the `CarouselView`, see [Xamarin.Forms CarouselView](#).

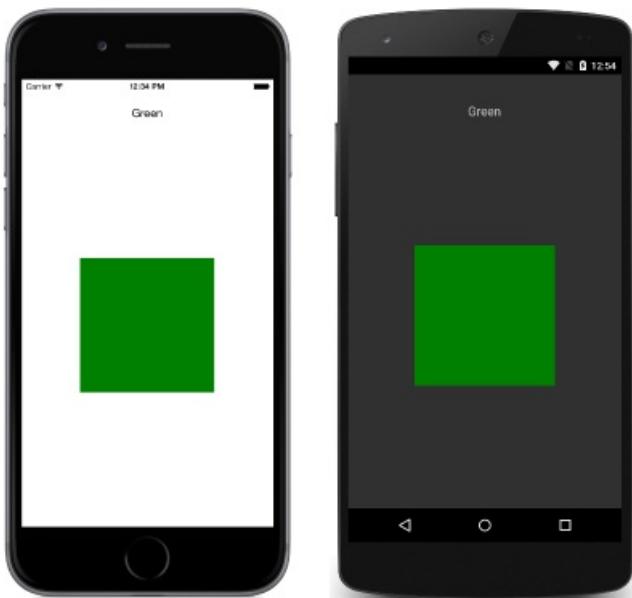
The following screenshots show a `CarouselPage` on each platform:



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.

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 `FlyoutPage`, the `FlyoutPage.IsEnabled` property should be set to `false` to prevent gesture conflicts between the `CarouselPage` and the `FlyoutPage`.

For more information about the `CarouselPage`, see [Chapter 25](#) of Charles Petzold's Xamarin.Forms book.

Create 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.

Populate 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.

Populate 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.

Related links

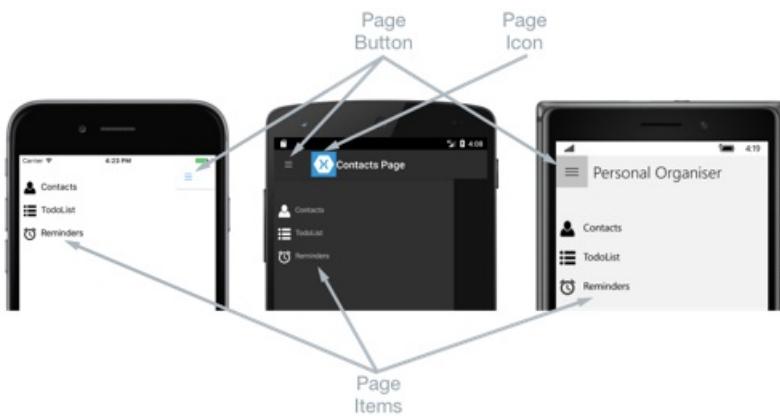
- [Page Varieties](#)
- [CarouselPage \(sample\)](#)
- [CarouselPageTemplate \(sample\)](#)
- [CarouselPage](#)

Xamarin.Forms FlyoutPage

8/4/2022 • 7 minutes to read • [Edit Online](#)

 [Download the sample](#)

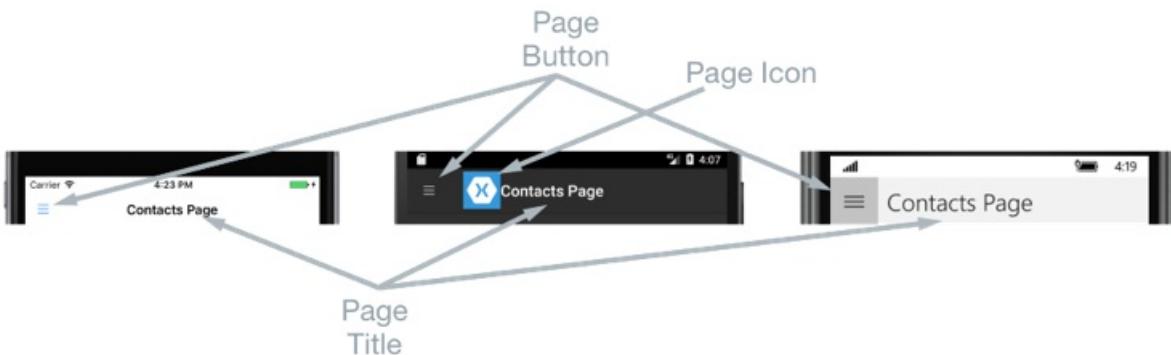
A flyout page typically displays a list of items, as shown in the following screenshots:



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 flyout 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 flyout 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 flyout 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 flyout 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 flyout page, provided that the detail page instance is wrapped in the `NavigationPage` instance. In addition, the flyout 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 flyout 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 flyout page.

Navigation behavior

The behavior of the navigation experience between flyout and detail pages is platform dependent:

- On iOS, the detail page *slides* to the right as the flyout page slides from the left, and the left part of the detail page is still visible.
- On Android, the detail and flyout pages are *overlaid* on each other.
- On UWP, the flyout page slides from the left over part of the detail page, provided that the `FlyoutLayoutBehavior` property is set to `Popover`.

Similar behavior will be observed in landscape mode, except that the flyout page on iOS and Android has a similar width as the flyout page in portrait mode, so more of the detail page will be visible.

For information about controlling the navigation behavior, see [Control the detail page layout behavior](#).

Create a FlyoutPage

A `FlyoutPage` contains `Flyout` and `Detail` properties that are both of type `Page`, which are used to get and set the flyout and detail pages respectively.

IMPORTANT

A `FlyoutPage` 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 flyout page of a `FlyoutPage` should always be a `ContentPage` instance, and that the detail page should only be populated with `TabbedPage`, `TabPage`, and `ContentPage` instances. This will help to ensure a consistent user experience across all platforms.

The following XAML code example shows a `FlyoutPage` that sets the `Flyout` and `Detail` properties:

```
<FlyoutPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            xmlns:local="clr-namespace:FlyoutPageNavigation;assembly=FlyoutPageNavigation"
            x:Class="FlyoutPageNavigation.MainPage">
    <FlyoutPage.Flyout>
        <local:FlyoutMenuPage x:Name="flyoutPage" />
    </FlyoutPage.Flyout>
    <FlyoutPage.Detail>
        <NavigationPage>
            <x:Arguments>
                <local:ContactsPage />
            </x:Arguments>
        </NavigationPage>
    </FlyoutPage.Detail>
</FlyoutPage>
```

The following code example shows the equivalent `FlyoutPage` created in C#:

```

public class MainPageCS : FlyoutPage
{
    FlyoutMenuPageCS flyoutPage;

    public MainPageCS()
    {
        flyoutPage = new FlyoutMenuPageCS();
        Flyout = flyoutPage;
        Detail = new NavigationPage(new ContactsPageCS());
        ...
    }
    ...
}

```

The `Flyout` property is set to a `ContentPage` instance. The `Detail` property is set to a `NavigationPage` containing a `ContentPage` instance.

Create the flyout page

The following XAML code example shows the declaration of the `FlyoutMenuPage` object, which is referenced through the `Flyout` property:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="using:FlyoutPageNavigation"
    x:Class="FlyoutPageNavigation.FlyoutMenuPage"
    Padding="0,40,0,0"
    IconImageSource="hamburger.png"
    Title="Personal Organiser">
    <StackLayout>
        <ListView x:Name="listView" x:FieldModifier="public">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type local:FlyoutPageItem}">
                    <local:FlyoutPageItem Title="Contacts" IconSource="contacts.png" TargetType="{x:Type local:ContactsPage}" />
                    <local:FlyoutPageItem Title="TodoList" IconSource="todo.png" TargetType="{x:Type local:TodoListPage}" />
                    <local:FlyoutPageItem 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 `FlyoutPageItem` objects. Each `FlyoutPageItem` defines `Title`, `IconSource`, and `TargetType` properties.

A `DataTemplate` is assigned to the `ListView.ItemTemplate` property, to display each `FlyoutPageItem`. 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 `FlyoutPageItem`.

The page has its `Title` and `IconImageSource` 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 `Flyout` 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 FlyoutMenuPageCS : ContentPage
{
    ListView listView;
    public ListView ListView { get { return listView; } }

    public FlyoutMenuPageCS()
    {
        var flyoutPageItems = new List<FlyoutPageItem>();
        flyoutPageItems.Add(new FlyoutPageItem
        {
            Title = "Contacts",
            IconSource = "contacts.png",
            TargetType = typeof(ContactsPageCS)
        });
        flyoutPageItems.Add(new FlyoutPageItem
        {
            Title = "TodoList",
            IconSource = "todo.png",
            TargetType = typeof(TodoListPageCS)
        });
        flyoutPageItems.Add(new FlyoutPageItem
        {
            Title = "Reminders",
            IconSource = "reminders.png",
            TargetType = typeof(ReminderPageCS)
        });
    }

    listView = new ListView
    {
        ItemsSource = flyoutPageItems,
        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");

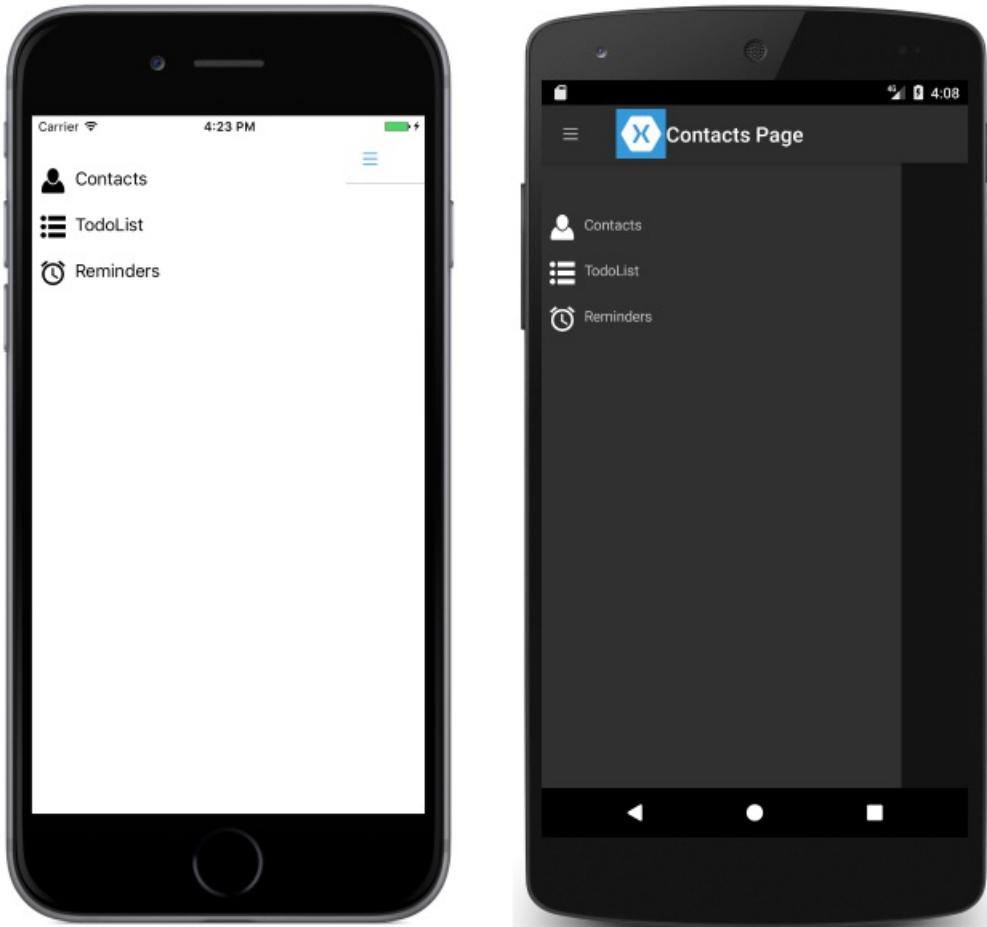
        grid.Children.Add(image);
        grid.Children.Add(label, 1, 0);

        return new ViewCell { View = grid };
    }),
        SeparatorVisibility = SeparatorVisibility.None
    };

    IconImageSource = "hamburger.png";
    Title = "Personal Organiser";
    Padding = new Thickness(0, 40, 0, 0);
    Content = new StackLayout
    {
        Children = { listView }
    };
}
}

```

The following screenshots show the flyout page on each platform:



Create and display the detail page

The `FlyoutMenuPage` instance contains a `ListView` property that exposes its `ListView` instance so that the `MainPage` `FlyoutPage` 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 : FlyoutPage
{
    public MainPage()
    {
        ...
        flyoutPage.listView.ItemSelected += OnItemSelected;
    }

    void OnItemSelected(object sender, SelectedItemChangedEventArgs e)
    {
        var item = e.SelectedItem as FlyoutPageItem;
        if (item != null)
        {
            Detail = new NavigationPage((Page)Activator.CreateInstance(item.TargetType));
            flyoutPage.listView.SelectedItem = null;
            IsPresented = false;
        }
    }
}
```

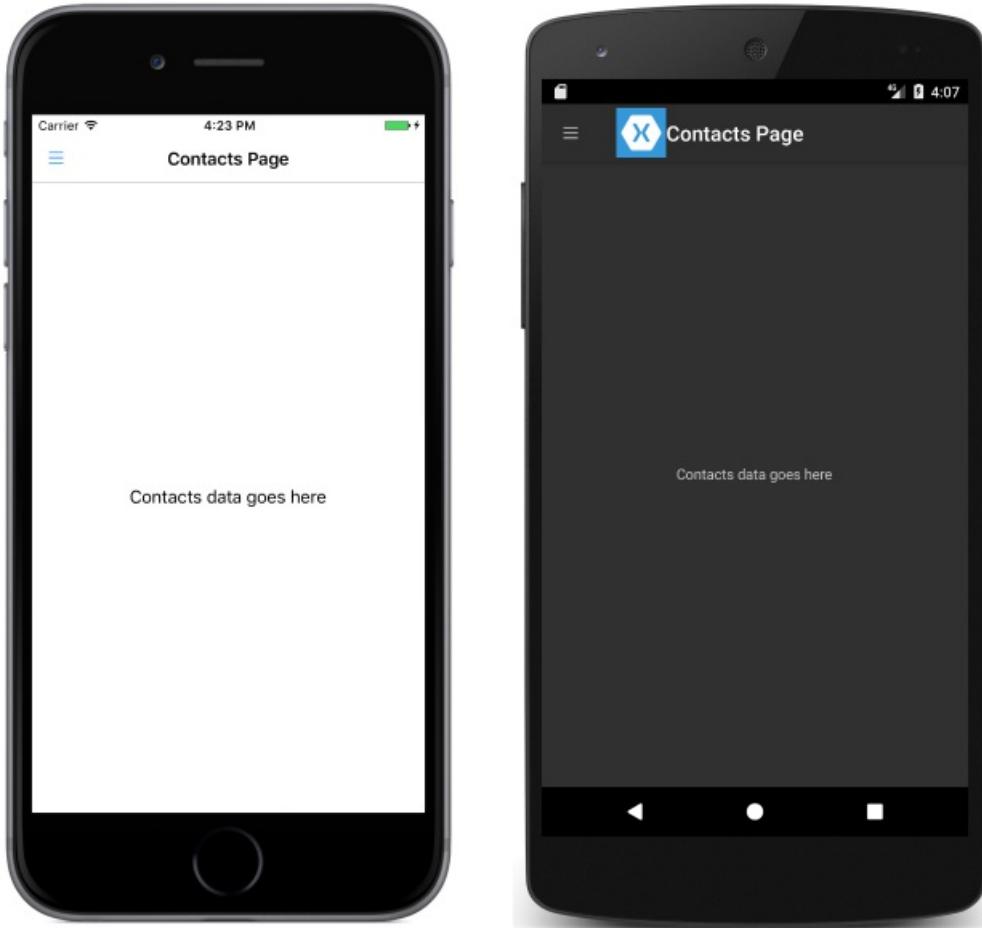
The `OnItemSelected` method performs the following actions:

- 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 `FlyoutPageItem`. The page type is wrapped in a `NavigationPage` instance to ensure that the icon referenced through the

`IconImageSource` property on the `FlyoutMenuPage` 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 `FlyoutMenuPage` is presented.
- The detail page is presented to the user by setting the `FlyoutPage.IsPresented` property to `false`. This property controls whether the flyout or detail page is presented. It should be set to `true` to display the flyout 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 flyout page:



Control the detail page layout behavior

How the `FlyoutPage` manages the flyout 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 `FlyoutLayoutBehavior` 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 flyout page.
- `Split` – The flyout 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 `FlyoutLayoutBehavior` property on a `FlyoutPage`:

```
<FlyoutPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="FlyoutPageNavigation.MainPage"
    FlyoutLayoutBehavior="Popover">
    ...
</FlyoutPage>
```

The following code example shows the equivalent `FlyoutPage` created in C#:

```
public class MainPageCS : FlyoutPage
{
    FlyoutMenuPageCS flyoutPage;

    public MainPageCS()
    {
        ...
        FlyoutLayoutBehavior = FlyoutLayoutBehavior.Popover;
    }
}
```

IMPORTANT

The value of the `FlyoutLayoutBehavior` property only affects applications running on tablets or the desktop. Applications running on phones always have the `Popover` behavior.

Related links

- [Page Varieties \(chapter 25\)](#)
- [FlyoutPage \(sample\)](#)
- [FlyoutPage API](#)

Xamarin.Forms Modal Pages

8/4/2022 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

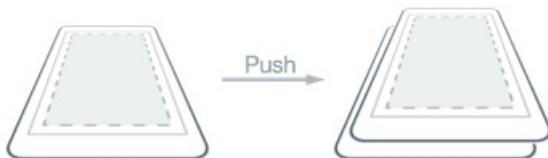
Xamarin.Forms provides support for modal pages. A modal page encourages users to complete a self-contained task that cannot be navigated away from until the task is completed or cancelled. This article demonstrates how to navigate to modal pages.

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

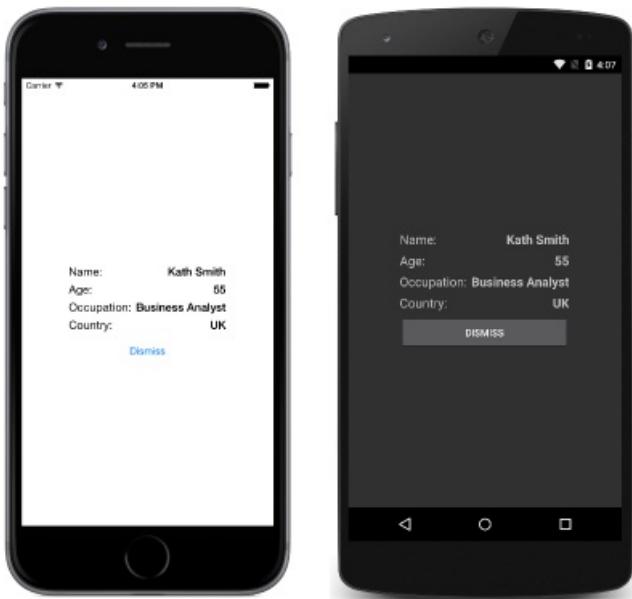
A `NavigationPage` instance is not required for performing modal page navigation.

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.

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 \(sample\)](#)

Xamarin.Forms Shell

8/4/2022 • 2 minutes to read • [Edit Online](#)

Introduction

Xamarin.Forms Shell reduces the complexity of mobile application development by providing the fundamental features that most mobile applications require. This includes a common navigation user experience, a URI-based navigation scheme, and an integrated search handler.

Create a Xamarin.Forms Shell application

The process for creating a Xamarin.Forms Shell application is to create a XAML file that subclasses the `Shell` class, set the `MainPage` property of the application's `App` class to the subclassed `Shell` object, and then describe the visual hierarchy of the application in the subclassed `Shell` class.

Flyout

A flyout is the optional root menu for a Shell application, and is accessible through an icon or by swiping from the side of the screen. The flyout consists of an optional header, flyout items, optional menu items, and an optional footer.

Tabs

After a flyout, the next level of navigation in a Shell application is the bottom tab bar. Alternatively, the navigation pattern for an application can begin with bottom tabs and make no use of a flyout. In both cases, when a bottom tab contains more than one page, the pages will be navigable by top tabs.

Pages

A `ShellContent` object represents the `ContentPage` object for each `FlyoutItem` or `Tab`.

Navigation

Shell applications can utilize a URI-based navigation scheme that uses routes to navigate to any page in the application, without having to follow a set navigation hierarchy.

Search

Shell applications can use integrated search functionality that's provided by a search box that can be added to the top of each page.

Lifecycle

Shell applications respect the Xamarin.Forms lifecycle, and additionally fire an `Appearing` event when a page is about to appear on the screen, and a `Disappearing` event when a page is about to disappear from the screen.

Custom renderers

Shell applications are customizable through the properties and methods that the various Shell classes expose. However, it's also possible to create Shell custom renderers when more sophisticated platform-specific

customizations are required.

Xamarin.Forms Shell introduction

8/4/2022 • 3 minutes to read • [Edit Online](#)

 [Download the sample](#)

Xamarin.Forms Shell reduces the complexity of mobile application development by providing the fundamental features that most mobile applications require, including:

- A single place to describe the visual hierarchy of an application.
- A common navigation user experience.
- A URI-based navigation scheme that permits navigation to any page in the application.
- An integrated search handler.

In addition, Shell applications benefit from an increased rendering speed, and reduced memory consumption.

IMPORTANT

Existing applications can adopt Shell and benefit immediately from navigation, performance, and extensibility improvements.

Application visual hierarchy

In a Xamarin.Forms Shell application, the visual hierarchy of the application is described in a class that subclasses the `Shell` class. This class can consist of three main hierarchical objects:

1. `FlyoutItem` or `TabBar`. A `FlyoutItem` represents one or more items in the flyout, and should be used when the navigation pattern for the application requires a flyout. A `TabBar` represents the bottom tab bar, and should be used when the navigation pattern for the application begins with bottom tabs and doesn't require a flyout.
2. `Tab`, which represents grouped content, navigable by bottom tabs.
3. `ShellContent`, which represents the `ContentPage` objects for each tab.

These objects don't represent any user interface, but rather the organization of the application's visual hierarchy. Shell will take these objects and produce the navigation user interface for the content.

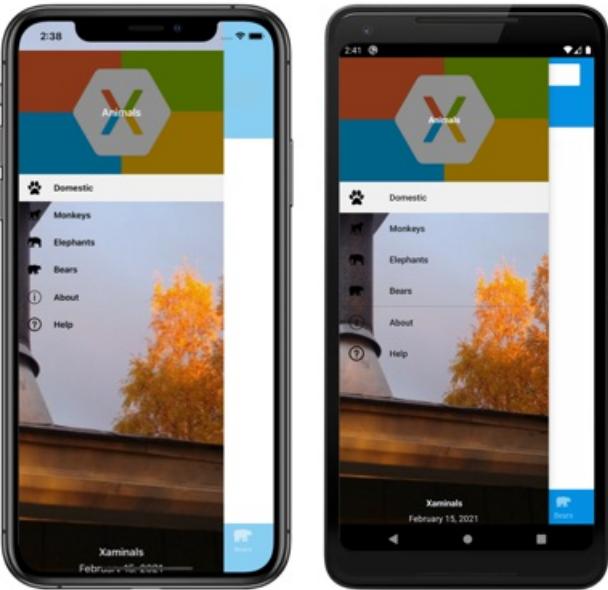
NOTE

Pages are created on demand in Shell applications, in response to navigation.

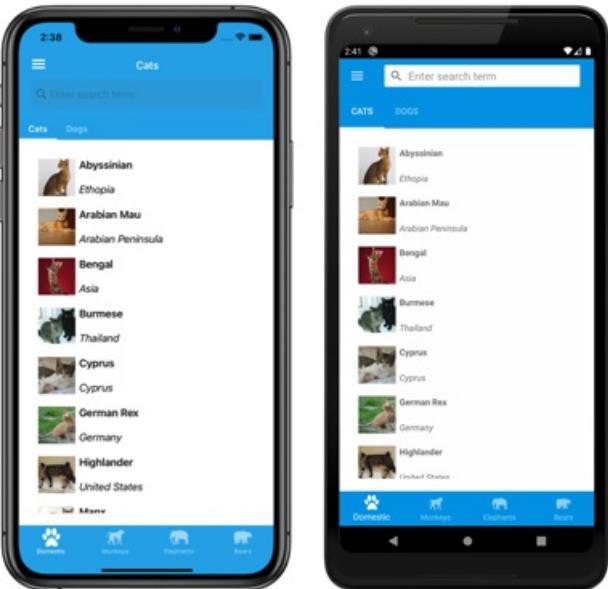
For more information, see [Create a Xamarin.Forms Shell application](#).

Navigation user experience

The navigation experience provided by Xamarin.Forms Shell is based on flyouts and tabs. The top level of navigation in a Shell application is either a flyout or a bottom tab bar, depending on the navigation requirements of the application. The following example shows an application where the top level of navigation is a flyout:



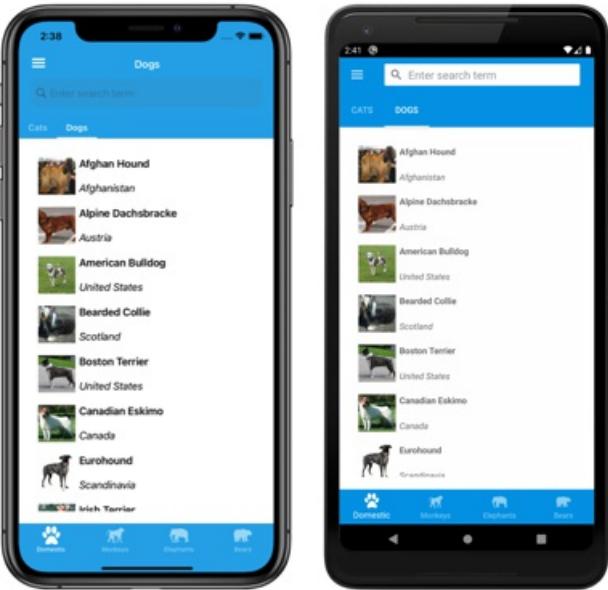
In this example, some flyout items are duplicated as tab bar items. However, there are also items that can only be accessed from the flyout. Selecting a flyout item results in the bottom tab that represents the item being selected and displayed:



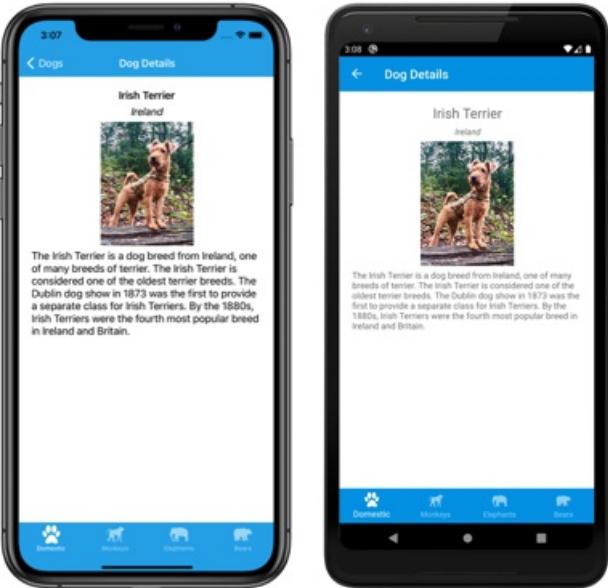
NOTE

When the flyout isn't open the bottom tab bar can be considered to be the top level of navigation in the application.

Each tab on the tab bar displays a [ContentPage](#). However, if a bottom tab contains more than one page, the pages are navigable by the top tab bar:



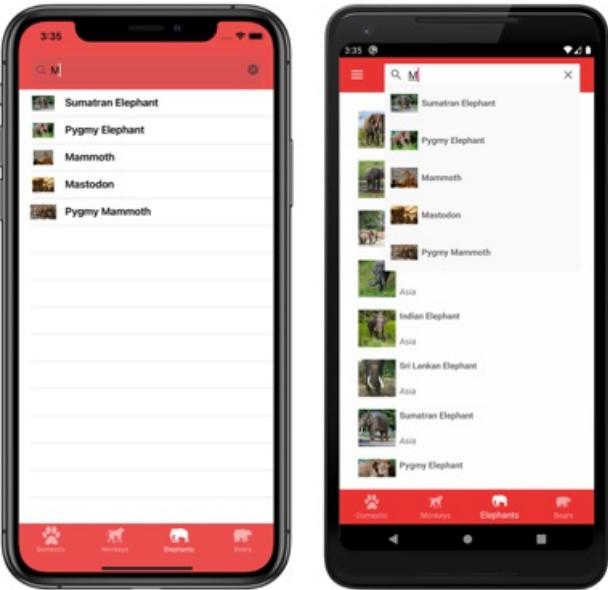
Within each tab, additional [ContentPage](#) objects that are known as detail pages, can be navigated to:



Shell uses a URI-based navigation experience that uses routes to navigate to any page in the application, without having to follow a set navigation hierarchy. In addition, it also provides the ability to navigate backwards without having to visit all of the pages on the navigation stack. For more information, see [Xamarin.Forms Shell navigation](#).

Search

Xamarin.Forms Shell includes integrated search functionality that's provided by the [SearchHandler](#) class. Search capability can be added to a page by adding a subclassed [SearchHandler](#) object to it. This results in a search box being added at the top of the page. When data is entered into the search box, the search suggestions area is populated with data:



Then, when a result is selected from the search suggestions area, custom logic can be executed such as navigating to a detail page.

For more information, see [Xamarin.Forms Shell search](#).

Platform support

Xamarin.Forms Shell is fully available on iOS and Android, but only partially available on the Universal Windows Platform (UWP). In addition, Shell is currently experimental on UWP and can only be used by adding the following line of code to the `App` class in your UWP project, before calling `Forms.Init`:

```
global::Xamarin.Forms.Forms.SetFlags("Shell_UWP_Experimental");
```

For more information about the status of Shell on UWP, see [Xamarin.Forms Shell Project Board](#) on github.com.

Related links

- [Xaminals \(sample\)](#)
- [Create a Xamarin.Forms Shell application](#)
- [Xamarin.Forms Shell navigation](#)
- [Xamarin.Forms Shell search](#)

Create a Xamarin.Forms Shell application

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

The process for creating a Xamarin.Forms Shell application is as follows:

1. Create a new Xamarin.Forms application, or load an existing application that you want to convert to a Shell application.
2. Add a XAML file to the shared code project, that subclasses the `Shell` class. For more information, see [Subclass the Shell class](#).
3. Set the `MainPage` property of the application's `App` class to the subclassed `Shell` object. For more information, see [Bootstrap the Shell application](#).
4. Describe the visual hierarchy of the application in the subclassed `Shell` class. For more information, see [Describe the visual hierarchy of the application](#).

For a step-by-step walkthrough of how to create a Shell application, see [Create a Xamarin.Forms application quickstart](#).

Subclass the Shell class

The first step in creating a Xamarin.Forms Shell application is to add a XAML file to the shared code project that subclasses the `Shell` class. This file can be named anything, but `AppShell` is recommended. The following code example shows a newly created `AppShell.xaml` file:

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

</Shell>
```

The following example shows the code-behind file, `AppShell.xaml.cs`:

```
using Xamarin.Forms;

namespace MyApp
{
    public partial class AppShell : Shell
    {
        public AppShell()
        {
            InitializeComponent();
        }
    }
}
```

Bootstrap the Shell application

After creating the XAML file that subclasses the `Shell` object, the `MainPage` property of the `App` class should be set to the subclassed `shell` object:

```
namespace MyApp
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
            MainPage = new AppShell();
        }
        ...
    }
}
```

In this example, the `AppShell` class is the XAML file that derives from the `Shell` class.

WARNING

While a blank Shell application will build, attempting to run it will result in a `InvalidOperationException` being thrown.

Describe the visual hierarchy of the application

The final step in creating a Xamarin.Forms Shell application is to describe the visual hierarchy of the application, in the subclassed `Shell` class. A subclassed `shell` class consists of three main hierarchical objects:

1. `FlyoutItem` or `TabBar`. A `FlyoutItem` represents one or more items in the flyout, and should be used when the navigation pattern for the application requires a flyout. A `TabBar` represents the bottom tab bar, and should be used when the navigation pattern for the application begins with bottom tabs and doesn't require a flyout. Every `FlyoutItem` object or `TabBar` object is a child of the `Shell` object.
2. `Tab`, which represents grouped content, navigable by bottom tabs. Every `Tab` object is a child of a `FlyoutItem` object or `TabBar` object.
3. `ShellContent`, which represents the `ContentPage` objects for each tab. Every `ShellContent` object is a child of a `Tab` object. When more than one `ShellContent` object is present in a `Tab`, the objects will be navigable by top tabs.

These objects don't represent any user interface, but rather the organization of the application's visual hierarchy. Shell will take these objects and produce the navigation user interface for the content.

The following XAML shows an example of a subclassed `Shell` class:

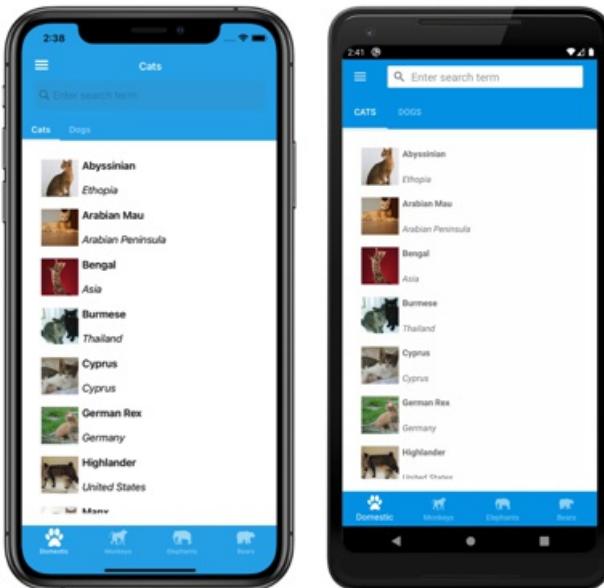
```

<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    ...
    <FlyoutItem FlyoutDisplayOptions="AsMultipleItems">
        <Tab Title="Domestic"
             Icon="paw.png">
            <ShellContent Title="Cats"
                          Icon="cat.png"
                          ContentTemplate="{DataTemplate views:CatsPage}" />
            <ShellContent Title="Dogs"
                          Icon="dog.png"
                          ContentTemplate="{DataTemplate views:DogsPage}" />
        </Tab>
        <!--
            Shell has implicit conversion operators that enable the Shell visual hierarchy to be simplified.
            This is possible because a subclassed Shell object can only ever contain a FlyoutItem object or a
            TabBar object,
            which can only ever contain Tab objects, which can only ever contain ShellContent objects.

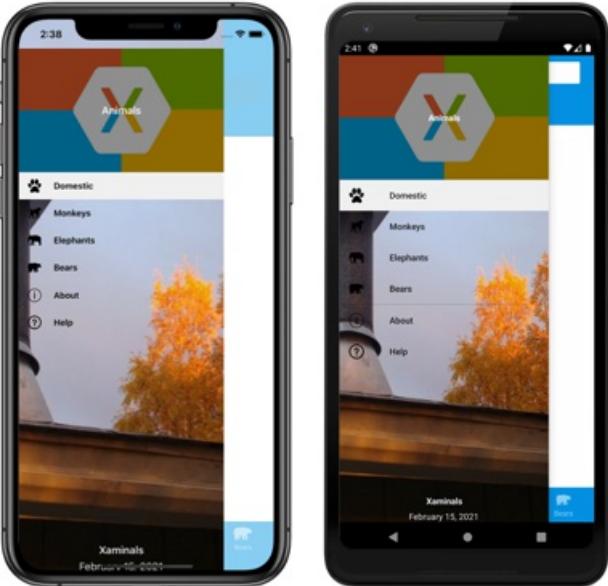
            The implicit conversion automatically wraps the ShellContent objects below in Tab objects.
        -->
        <ShellContent Title="Monkeys"
                      Icon="monkey.png"
                      ContentTemplate="{DataTemplate views:MonkeysPage}" />
        <ShellContent Title="Elephants"
                      Icon="elephant.png"
                      ContentTemplate="{DataTemplate views:ElephantsPage}" />
        <ShellContent Title="Bears"
                      Icon="bear.png"
                      ContentTemplate="{DataTemplate views:BearsPage}" />
    </FlyoutItem>
    ...
</Shell>

```

When run, this XAML displays the `CatsPage`, because it's the first item of content declared in the subclassed `Shell` class:



Pressing the hamburger icon, or swiping from the left, displays the flyout:



Multiple items are displayed on the flyout because the [FlyoutDisplayOptions](#) property is set to `AsMultipleItems`. For more information, see [Flyout display options](#).

IMPORTANT

In a Shell application, pages are created on demand in response to navigation. This is accomplished by using the [DataTemplate](#) markup extension to set the [ContentTemplate](#) property of each [ShellContent](#) object to a [ContentPage](#) object.

Related links

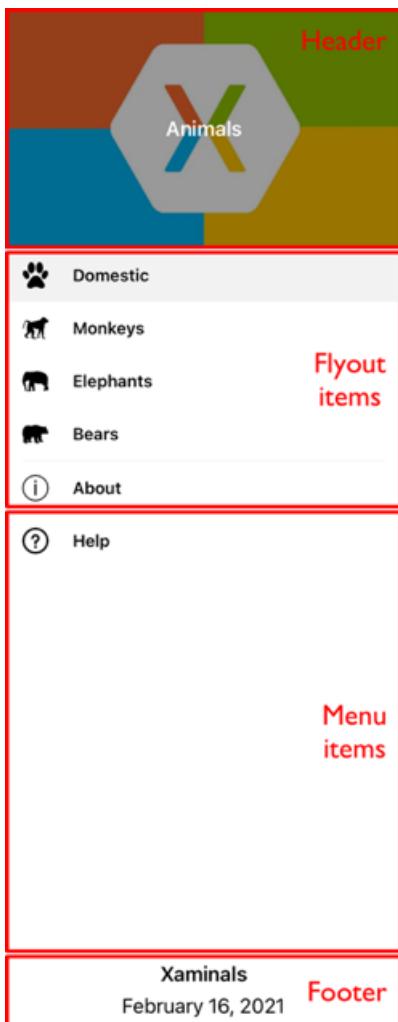
- [Xamimals \(sample\)](#)
- [Create a Xamarin.Forms application quickstart](#)

Xamarin.Forms Shell flyout

8/4/2022 • 16 minutes to read • [Edit Online](#)

 [Download the sample](#)

The navigation experience provided by Xamarin.Forms Shell is based on flyouts and tabs. A flyout is the optional root menu for a Shell application, and is fully customizable. It's accessible through an icon or by swiping from the side of the screen. The flyout consists of an optional header, flyout items, optional menu items, and an optional footer:



Flyout items

One or more flyout items can be added to the flyout, and each flyout item is represented by a `FlyoutItem` object. Each `FlyoutItem` object should be a child of the subclassed `Shell` object. Flyout items appear at the top of the flyout when a flyout header isn't present.

The following example creates a flyout containing two flyout items:

```

<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:controls="clr-namespace:Xaminals.Controls"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    <FlyoutItem Title="Cats"
                Icon="cat.png">
        <Tab>
            <ShellContent ContentTemplate="{DataTemplate views:CatsPage}" />
        </Tab>
    </FlyoutItem>
    <FlyoutItem Title="Dogs"
                Icon="dog.png">
        <Tab>
            <ShellContent ContentTemplate="{DataTemplate views:DogsPage}" />
        </Tab>
    </FlyoutItem>
</Shell>

```

The `FlyoutItem.Title` property, of type `string`, defines the title of the flyout item. The `FlyoutItem.Icon` property, of type `ImageSource`, defines the icon of the flyout item:



In this example, each `ShellContent` object can only be accessed through flyout items, and not through tabs. This is because by default, tabs will only be displayed if the flyout item contains more than one tab.

IMPORTANT

In a Shell application, pages are created on demand in response to navigation. This is accomplished by using the `DataTemplate` markup extension to set the `ContentTemplate` property of each `ShellContent` object to a `ContentPage` object.

Shell has implicit conversion operators that enable the Shell visual hierarchy to be simplified, without introducing additional views into the visual tree. This is possible because a subclassed `Shell` object can only ever contain `FlyoutItem` objects or a `TabBar` object, which can only ever contain `Tab` objects, which can only ever contain `ShellContent` objects. These implicit conversion operators can be used to remove the `FlyoutItem` and `Tab` objects from the previous example:

```

<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:controls="clr-namespace:Xaminals.Controls"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    <ShellContent Title="Cats"
                  Icon="cat.png"
                  ContentTemplate="{DataTemplate views:CatsPage}" />
    <ShellContent Title="Dogs"
                  Icon="dog.png"
                  ContentTemplate="{DataTemplate views:DogsPage}" />
</Shell>

```

This implicit conversion automatically wraps each `ShellContent` object in `Tab` objects, which are wrapped in `FlyoutItem` objects.

NOTE

All `FlyoutItem` objects in a subclassed `Shell` object are automatically added to the `Shell.FlyoutItems` collection, which defines the list of items that will be shown in the flyout.

Flyout display options

The `FlyoutItem.FlyoutDisplayOptions` property configures how a flyout item and its children are displayed in the flyout. This property should be set to a `FlyoutDisplayOptions` enumeration member:

- `AsSingleItem`, indicates that the item will be visible as a single item. This is the default value of the `FlyoutDisplayOptions` property.
- `AsMultipleItems`, indicates that the item and its children will be visible in the flyout as a group of items.

A flyout item for each `Tab` object within a `FlyoutItem` can be displayed by setting the `FlyoutItem.FlyoutDisplayOptions` property to `AsMultipleItems`:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:controls="clr-namespace:Xaminals.Controls"
       xmlns:views="clr-namespace:Xaminals.Views"
       FlyoutHeaderBehavior="CollapseOnScroll"
       x:Class="Xaminals.AppShell">

    <FlyoutItem FlyoutDisplayOptions="AsMultipleItems">
        <Tab Title="Domestic"
             Icon="paw.png">
            <ShellContent Title="Cats"
                         Icon="cat.png"
                         ContentTemplate="{DataTemplate views:CatsPage}" />
            <ShellContent Title="Dogs"
                         Icon="dog.png"
                         ContentTemplate="{DataTemplate views:DogsPage}" />
        </Tab>
        <ShellContent Title="Monkeys"
                     Icon="monkey.png"
                     ContentTemplate="{DataTemplate views:MonkeysPage}" />
        <ShellContent Title="Elephants"
                     Icon="elephant.png"
                     ContentTemplate="{DataTemplate views:ElephantsPage}" />
        <ShellContent Title="Bears"
                     Icon="bear.png"
                     ContentTemplate="{DataTemplate views:BearsPage}" />
    </FlyoutItem>

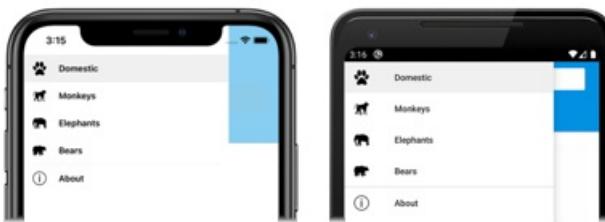
    <ShellContent Title="About"
                  Icon="info.png"
                  ContentTemplate="{DataTemplate views:AboutPage}" />
</Shell>
```

In this example, flyout items are created for the `Tab` object that's a child of the `FlyoutItem` object, and the `ShellContent` objects that are children of the `FlyoutItem` object. This occurs because each `ShellContent` object that's a child of the `FlyoutItem` object is automatically wrapped in a `Tab` object. In addition, a flyout item is created for the final `ShellContent` object, which is automatically wrapped in a `Tab` object, and then in a `FlyoutItem` object.

NOTE

Tabs are displayed when a `FlyoutItem` contains more than one `ShellContent` object.

This results in the following flyout items:



Define FlyoutItem appearance

The appearance of each `FlyoutItem` can be customized by setting the `Shell.ItemTemplate` attached property to a `DataTemplate`:

```
<Shell ...>
    ...
    <Shell.ItemTemplate>
        <DataTemplate>
            <Grid ColumnDefinitions="0.2*,0.8*">
                <Image Source="{Binding FlyoutIcon}"
                    Margin="5"
                    HeightRequest="45" />
                <Label Grid.Column="1"
                    Text="{Binding Title}"
                    FontAttributes="Italic"
                    VerticalTextAlignment="Center" />
            </Grid>
        </DataTemplate>
    </Shell.ItemTemplate>
</Shell>
```

This example displays the title of each `FlyoutItem` object in italics:



Because `Shell.ItemTemplate` is an attached property, different templates can be attached to specific `FlyoutItem` objects.

NOTE

Shell provides the `Title` and `FlyoutIcon` properties to the `BindingContext` of the `ItemTemplate`.

In addition, Shell includes three style classes, which are automatically applied to `FlyoutItem` objects. For more information, see [Style FlyoutItem and MenuItem objects](#).

Default template for FlyoutItems

The default `DataTemplate` used for each `FlyoutItem` is shown below:

```
<DataTemplate x:Key="FlyoutTemplate">
    <Grid x:Name="FlyoutItemLayout"
        HeightRequest="{x:OnPlatform Android=50}"
        ColumnSpacing="{x:OnPlatform UWP=0}"
        RowSpacing="{x:OnPlatform UWP=0}">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroupList>
                <VisualStateGroup x:Name="CommonStates">
```

```

        <VisualState x:Name="Normal" />
        <VisualState x:Name="Selected">
            <VisualState.Setters>
                <Setter Property="BackgroundColor"
                    Value="{x:OnPlatform Android=#F2F2F2, iOS=#F2F2F2}" />
            </VisualState.Setters>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="{x:OnPlatform Android=54, iOS=50, UWP=Auto}" />
    <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>
<Image x:Name="FlyoutItemImage"
    Source="{Binding FlyoutIcon}"
    VerticalOptions="Center"
    HorizontalOptions="{x:OnPlatform Default=Center, UWP=Start}"
    HeightRequest="{x:OnPlatform Android=24, iOS=22, UWP=16}"
    WidthRequest="{x:OnPlatform Android=24, iOS=22, UWP=16}">
    <Image.Margin>
        <OnPlatform x:TypeArguments="Thickness">
            <OnPlatform.Platforms>
                <On Platform="UWP"
                    Value="12,0,12,0" />
            </OnPlatform.Platforms>
        </OnPlatform>
    </Image.Margin>
</Image>
<Label x:Name="FlyoutItemLabel"
    Grid.Column="1"
    Text="{Binding Title}"
    FontSize="{x:OnPlatform Android=14, iOS=Small}"
    HorizontalOptions="{x:OnPlatform UWP=Start}"
    HorizontalTextAlignment="{x:OnPlatform UWP=Start}"
    FontAttributes="{x:OnPlatform iOS=Bold}"
    VerticalTextAlignment="Center">
    <Label.TextColor>
        <OnPlatform x:TypeArguments="Color">
            <OnPlatform.Platforms>
                <On Platform="Android"
                    Value="#D2000000" />
            </OnPlatform.Platforms>
        </OnPlatform>
    </Label.TextColor>
    <Label.Margin>
        <OnPlatform x:TypeArguments="Thickness">
            <OnPlatform.Platforms>
                <On Platform="Android"
                    Value="20, 0, 0, 0" />
            </OnPlatform.Platforms>
        </OnPlatform>
    </Label.Margin>
    <Label.FontFamily>
        <OnPlatform x:TypeArguments="x:String">
            <OnPlatform.Platforms>
                <On Platform="Android"
                    Value="sans-serif-medium" />
            </OnPlatform.Platforms>
        </OnPlatform>
    </Label.FontFamily>
</Label>
</Grid>
</DataTemplate>

```

This template can be used for as a basis for making alterations to the existing flyout layout, and also shows the visual states that are implemented for flyout items.

In addition, the `Grid`, `Image`, and `Label` elements all have `x:Name` values and so can be targeted with the Visual State Manager. For more information, see [Set state on multiple elements](#).

NOTE

The same template can also be used for `MenuItem` objects.

Replace flyout content

Flyout items, which represent the flyout content, can optionally be replaced with your own content by setting the `Shell.FlyoutContent` bindable property to an `object`:

```
<Shell ...>
    ...
    <Shell.FlyoutContent>
        <CollectionView BindingContext="{x:Reference shell}"
            IsGrouped="True"
            ItemsSource="{Binding FlyoutItems}">
            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <Label Text="{Binding Title}"
                        TextColor="White"
                        FontSize="Large" />
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>
    </Shell.FlyoutContent>
</Shell>
```

In this example, the flyout content is replaced with a `CollectionView` that displays the title of each item in the `FlyoutItems` collection.

NOTE

The `FlyoutItems` property, in the `Shell` class, is a read-only collection of flyout items.

Alternatively, flyout content can be defined by setting the `Shell.FlyoutContentTemplate` bindable property to a `DataTemplate`:

```
<Shell ...>
    ...
    <Shell.FlyoutContentTemplate>
        <DataTemplate>
            <CollectionView BindingContext="{x:Reference shell}"
                IsGrouped="True"
                ItemsSource="{Binding FlyoutItems}">
                <CollectionView.ItemTemplate>
                    <DataTemplate>
                        <Label Text="{Binding Title}"
                            TextColor="White"
                            FontSize="Large" />
                    </DataTemplate>
                </CollectionView.ItemTemplate>
            </CollectionView>
        </DataTemplate>
    </Shell.FlyoutContentTemplate>
</Shell>
```

IMPORTANT

A flyout header can optionally be displayed above your flyout content, and a flyout footer can optionally be displayed below your flyout content. If your flyout content is scrollable, Shell will attempt to honor the scroll behavior of your flyout header.

Menu items

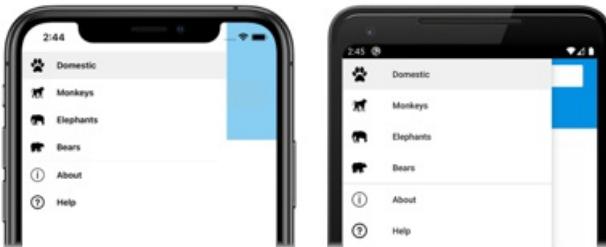
Menu items can be optionally added to the flyout, and each menu item is represented by a `MenuItem` object. The position of `MenuItem` objects on the flyout is dependent upon their declaration order in the Shell visual hierarchy. Therefore, any `MenuItem` objects declared before `FlyoutItem` objects will appear before the `FlyoutItem` objects in the flyout, and any `MenuItem` objects declared after `FlyoutItem` objects will appear after the `FlyoutItem` objects in the flyout.

The `MenuItem` class has a `Clicked` event, and a `Command` property. Therefore, `MenuItem` objects enable scenarios that execute an action in response to the `MenuItem` being tapped.

`MenuItem` objects can be added to the flyout as shown in the following example:

```
<Shell ...>
...
<MenuItem Text="Help"
    IconImageSource="help.png"
    Command="{Binding HelpCommand}"
    CommandParameter="https://docs.microsoft.com/xamarin/xamarin-forms/app-fundamentals/shell" />
</Shell>
```

This example adds a `MenuItem` object to the flyout, beneath all the flyout items:



The `MenuItem` object executes an `ICommand` named `HelpCommand`, which opens the URL specified by the `CommandParameter` property in the system web browser.

NOTE

The `BindingContext` of each `MenuItem` is inherited from the subclassed `Shell` object.

Define MenuItem appearance

The appearance of each `MenuItem` can be customized by setting the `Shell.MenuItemTemplate` attached property to a `DataTemplate`:

```

<Shell ...>
    <Shell.MenuItemTemplate>
        <DataTemplate>
            <Grid ColumnDefinitions="0.2*,0.8*>
                <Image Source="{Binding Icon}" Margin="5" HeightRequest="45" />
                <Label Grid.Column="1" Text="{Binding Text}" FontAttributes="Italic" VerticalTextAlignment="Center" />
            </Grid>
        </DataTemplate>
    </Shell.MenuItemTemplate>
    ...
    <MenuItem Text="Help" IconImageSource="help.png" Command="{Binding HelpCommand}" CommandParameter="https://docs.microsoft.com/xamarin/xamarin-forms/app-fundamentals/shell" />
</Shell>

```

This example attaches the `MenuItemTemplate` to each `MenuItem` object, displaying the title of the `MenuItem` object in italics:



Because `Shell.MenuItemTemplate` is an attached property, different templates can be attached to specific `MenuItem` objects.

NOTE

Shell provides the `Text` and `IconImageSource` properties to the `BindingContext` of the `MenuItemTemplate`. You can also use `Title` in place of `Text` and `Icon` in place of `IconImageSource` which will let you reuse the same template for menu items and flyout items.

The default template for `FlyoutItem` objects can also be used for `MenuItem` objects. For more information, see [Default template for FlyoutItems](#).

Style FlyoutItem and MenuItem objects

Shell includes three style classes, which are automatically applied to `FlyoutItem` and `MenuItem` objects. The style class names are `FlyoutItemLabelStyle`, `FlyoutItemImageStyle`, and `FlyoutItemLayoutStyle`.

The following XAML shows an example of defining styles for these style classes:

```

<Style TargetType="Label"
       Class="FlyoutItemLabelStyle">
    <Setter Property="TextColor"
           Value="Black" />
    <Setter Property="HeightRequest"
           Value="100" />
</Style>

<Style TargetType="Image"
       Class="FlyoutItemImageStyle">
    <Setter Property="Aspect"
           Value="Fill" />
</Style>

<Style TargetType="Layout"
       Class="FlyoutItemLayoutStyle"
       ApplyToDerivedTypes="True">
    <Setter Property="BackgroundColor"
           Value="Teal" />
</Style>

```

These styles will automatically be applied to `FlyoutItem` and `MenuItem` objects, without having to set their `StyleClass` properties to the style class names.

In addition, custom style classes can be defined and applied to `FlyoutItem` and `MenuItem` objects. For more information about style classes, see [Xamarin.Forms Style Classes](#).

Flyout header

The flyout header is the content that optionally appears at the top of the flyout, with its appearance being defined by an `object` that can be set with the `Shell.FlyoutHeader` bindable property:

```

<Shell ...>
    <Shell.FlyoutHeader>
        <controls:FlyoutHeader />
    </Shell.FlyoutHeader>
</Shell>

```

The `FlyoutHeader` type is shown in the following example:

```

<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Xaminals.Controls.FlyoutHeader"
             HeightRequest="200">
    <Grid BackgroundColor="Black">
        <Image Aspect="AspectFill"
              Source="xamarinstore.jpg"
              Opacity="0.6" />
        <Label Text="Animals"
              TextColor="White"
              FontAttributes="Bold"
              HorizontalTextAlignment="Center"
              VerticalTextAlignment="Center" />
    </Grid>
</ContentView>

```

This results in the following flyout header:



Alternatively, the flyout header appearance can be defined by setting the `Shell.FlyoutHeaderTemplate` bindable property to a `DataTemplate`:

```
<Shell ...>
    <Shell.FlyoutHeaderTemplate>
        <DataTemplate>
            <Grid BackgroundColor="Black"
                  HeightRequest="200">
                <Image Aspect="AspectFill"
                      Source="xamarinstore.jpg"
                      Opacity="0.6" />
                <Label Text="Animals"
                      TextColor="White"
                      FontAttributes="Bold"
                      HorizontalTextAlignment="Center"
                      VerticalTextAlignment="Center" />
            </Grid>
        </DataTemplate>
    </Shell.FlyoutHeaderTemplate>
</Shell>
```

By default, the flyout header will be fixed in the flyout while the content below will scroll if there are enough items. However, this behavior can be changed by setting the `Shell.FlyoutHeaderBehavior` bindable property to one of the `FlyoutHeaderBehavior` enumeration members:

- `Default` – indicates that the default behavior for the platform will be used. This is the default value of the `FlyoutHeaderBehavior` property.
- `Fixed` – indicates that the flyout header remains visible and unchanged at all times.
- `Scroll` – indicates that the flyout header scrolls out of view as the user scrolls the items.
- `CollapseOnScroll` – indicates that the flyout header collapses to a title only, as the user scrolls the items.

The following example shows how to collapse the flyout header as the user scrolls:

```
<Shell ...
    FlyoutHeaderBehavior="CollapseOnScroll">
    ...
</Shell>
```

Flyout footer

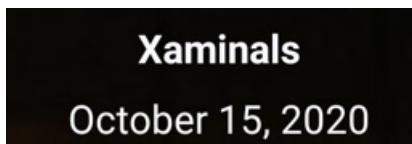
The flyout footer is the content that optionally appears at the bottom of the flyout, with its appearance being defined by an `object` that can be set with the `Shell.FlyoutFooter` bindable property:

```
<Shell ...>
    <Shell.FlyoutFooter>
        <controls:FlyoutFooter />
    </Shell.FlyoutFooter>
</Shell>
```

The `FlyoutFooter` type is shown in the following example:

```
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:sys="clr-namespace:System;assembly=netstandard"
    x:Class="Xaminals.Controls.FlyoutFooter">
<StackLayout>
    <Label Text="Xaminals"
        TextColor="GhostWhite"
        FontAttributes="Bold"
        HorizontalOptions="Center" />
    <Label Text="{Binding Source={x:Static sys:DateTime.Now}, StringFormat='{0:MMMM dd, yyyy}'}"
        TextColor="GhostWhite"
        HorizontalOptions="Center" />
</StackLayout>
</ContentView>
```

This results in the following flyout footer:



Alternatively, the flyout footer appearance can be defined by setting the `Shell.FlyoutFooterTemplate` property to a `DataTemplate`:

```
<Shell ...>
    <Shell.FlyoutFooterTemplate>
        <DataTemplate>
            <StackLayout>
                <Label Text="Xaminals"
                    TextColor="GhostWhite"
                    FontAttributes="Bold"
                    HorizontalOptions="Center" />
                <Label Text="{Binding Source={x:Static sys:DateTime.Now}, StringFormat='{0:MMMM dd, yyyy}'}"
                    TextColor="GhostWhite"
                    HorizontalOptions="Center" />
            </StackLayout>
        </DataTemplate>
    </Shell.FlyoutFooterTemplate>
</Shell>
```

The flyout footer is fixed to the bottom of the flyout, and can be any height. In addition, the footer never obscures any menu items.

Flyout width and height

The width and height of the flyout can be customized by setting the `Shell.FlyoutWidth` and `Shell.FlyoutHeight` attached properties to `double` values:

```
<Shell ...
    FlyoutWidth="400"
    FlyoutHeight="200">
    ...
</Shell>
```

This enables scenarios such as expanding the flyout across the entire screen, or reducing the height of the flyout so that it doesn't obscure the tab bar.

Flyout icon

By default, Shell applications have a hamburger icon which, when pressed, opens the flyout. This icon can be changed by setting the `Shell.FlyoutIcon` bindable property, of type `ImageSource`, to an appropriate icon:

```
<Shell ...>
    FlyoutIcon="flyouticon.png"
    ...
</Shell>
```

Flyout background

The background color of the flyout can be set with the `Shell.FlyoutBackgroundColor` bindable property:

```
<Shell ...>
    FlyoutBackgroundColor="AliceBlue"
    ...
</Shell>
```

NOTE

The `Shell.FlyoutBackgroundColor` can also be set from a Cascading Style Sheet (CSS). For more information, see [Xamarin.Forms Shell specific properties](#).

Alternatively, the background of the flyout can be specified by setting the `Shell.FlyoutBackground` bindable property to a `Brush`:

```
<Shell ...>
    FlyoutBackground="LightGray"
    ...
</Shell>
```

In this example, the flyout background is painted with a light gray `SolidColorBrush`.

The following example shows setting the flyout background to a `LinearGradientBrush`:

```
<Shell ...>
    <Shell.FlyoutBackground>
        <LinearGradientBrush StartPoint="0,0"
            EndPoint="1,1">
            <GradientStop Color="#8A2387"
                Offset="0.1" />
            <GradientStop Color="#E94057"
                Offset="0.6" />
            <GradientStop Color="#F27121"
                Offset="1.0" />
        </LinearGradientBrush>
    </Shell.FlyoutBackground>
    ...
</Shell>
```

For more information about brushes, see [Xamarin.Forms Brushes](#).

Flyout background image

The flyout can have an optional background image, which appears beneath the flyout header and behind any flyout items, menu items, and the flyout footer. The background image can be specified by setting the `FlyoutBackgroundImage` bindable property, of type `ImageSource`, to a file, embedded resource, URL, or stream.

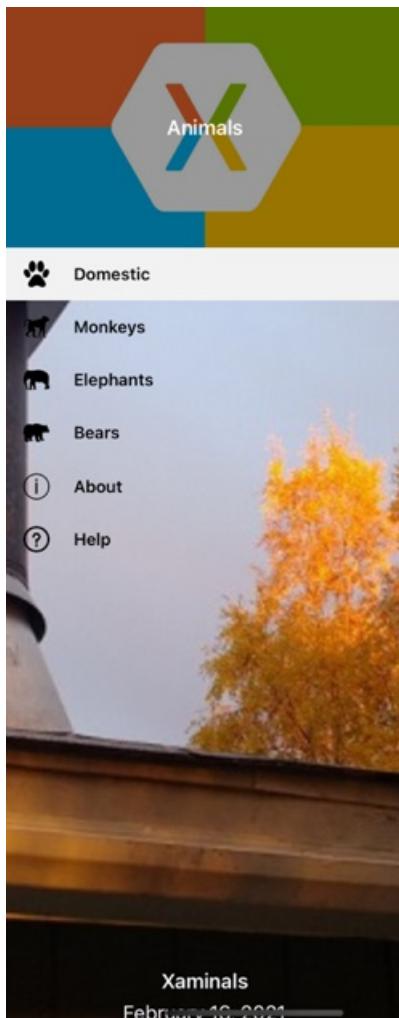
The aspect ratio of the background image can be configured by setting the `FlyoutBackgroundImageAspect` bindable property, of type `Aspect`, to one of the `Aspect` enumeration members:

- `AspectFill` - clips the image so that it fills the display area while preserving the aspect ratio.
- `AspectFit` - letterboxes the image, if required, so that the 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. This is the default value of the `FlyoutBackgroundImageAspect` property.
- `Fill` - stretches the image to completely and exactly fill the display area. This may result in image distortion.

The following example shows setting these properties:

```
<Shell ...>
    FlyoutBackgroundImage="photo.jpg"
    FlyoutBackgroundImageAspect="AspectFill">
    ...
</Shell>
```

This results in a background image appearing in the flyout, below the flyout header:



Flyout backdrop

The backdrop of the flyout, which is the appearance of the flyout overlay, can be specified by setting the `Shell.FlyoutBackdrop` attached property to a `Brush`:

```
<Shell ...>
    FlyoutBackdrop="Silver">
    ...
</Shell>
```

In this example, the flyout backdrop is painted with a silver [SolidColorBrush](#).

IMPORTANT

The [FlyoutBackdrop](#) attached property can be set on any Shell element, but will only be applied when it's set on [Shell](#), [FlyoutItem](#), or [TabBar](#) objects.

The following example shows setting the flyout backdrop to a [LinearGradientBrush](#):

```
<Shell ...>
    <Shell.FlyoutBackdrop>
        <LinearGradientBrush StartPoint="0,0"
                            EndPoint="1,1">
            <GradientStop Color="#8A2387"
                           Offset="0.1" />
            <GradientStop Color="#E94057"
                           Offset="0.6" />
            <GradientStop Color="#F27121"
                           Offset="1.0" />
        </LinearGradientBrush>
    </Shell.FlyoutBackdrop>
    ...
</Shell>
```

For more information about brushes, see [Xamarin.Forms Brushes](#).

Flyout behavior

The flyout can be accessed through the hamburger icon or by swiping from the side of the screen. However, this behavior can be changed by setting the [Shell.FlyoutBehavior](#) attached property to one of the [FlyoutBehavior](#) enumeration members:

- [Disabled](#) – indicates that the flyout can't be opened by the user.
- [Flyout](#) – indicates that the flyout can be opened and closed by the user. This is the default value for the [FlyoutBehavior](#) property.
- [Locked](#) – indicates that the flyout can't be closed by the user, and that it doesn't overlap content.

The following example shows how to disable the flyout:

```
<Shell ...>
    FlyoutBehavior="Disabled">
    ...
</Shell>
```

NOTE

The [FlyoutBehavior](#) attached property can be set on [Shell](#), [FlyoutItem](#), [ShellContent](#), and page objects, to override the default flyout behavior.

Flyout vertical scroll

By default, a flyout can be scrolled vertically when the flyout items don't fit in the flyout. This behavior can be changed by setting the `Shell.FlyoutVerticalScrollMode` bindable property to one of the `ScrollMode` enumeration members:

- `Disabled` – indicates that vertical scrolling will be disabled.
- `Enabled` – indicates that vertical scrolling will be enabled.
- `Auto` – indicates that vertical scrolling will be enabled if the flyout items don't fit in the flyout. This is the default value of the `FlyoutVerticalScrollMode` property.

The following example shows how to disable vertical scrolling:

```
<Shell ...>
    FlyoutVerticalScrollMode="Disabled"
    ...
</Shell>
```

FlyoutItem tab order

By default, the tab order of `FlyoutItem` objects 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 `FlyoutItem` objects will be navigated through with a keyboard, and often this default order is the best order.

The default tab order can be changed by setting the `FlyoutItem.TabIndex` property, which indicates the order in which `FlyoutItem` objects receive focus when the user navigates through items 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:

- `FlyoutItem` objects with a `TabIndex` equal to 0 are added to the tab order based on their declaration order in XAML or child collections.
- `FlyoutItem` objects with a `TabIndex` greater than 0 are added to the tab order based on their `TabIndex` value.
- `FlyoutItem` objects 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 `FlyoutItem` objects in ascending `TabIndex` order, wrapping around to the beginning once the final object is reached.

In addition to setting the tab order of `FlyoutItem` objects, it may be necessary to exclude some objects from the tab order. This can be achieved with the `FlyoutItem.IsTabStop` property, which indicates whether a `FlyoutItem` is included in tab navigation. Its default value is `true`, and when its value is `false` the `FlyoutItem` is ignored by the tab-navigation infrastructure, irrespective if a `TabIndex` is set.

FlyoutItem selection

When a Shell application that uses a flyout is first run, the `Shell.CurrentItem` property will be set to the first `FlyoutItem` object in the subclassed `Shell` object. However, the property can be set to another `FlyoutItem`, as shown in the following example:

```

<Shell ...
    CurrentItem="{x:Reference aboutItem}"
    <FlyoutItem FlyoutDisplayOptions="AsMultipleItems">
        ...
    </FlyoutItem>
    <ShellContent x:Name="aboutItem"
        Title="About"
        Icon="info.png"
        ContentTemplate="{DataTemplate views:AboutPage}" />
</Shell>

```

This example sets the `CurrentItem` property to the `ShellContent` object named `aboutItem`, which results in it being selected and displayed. In this example, an implicit conversion is used to wrap the `ShellContent` object in a `Tab` object, which is wrapped in a `FlyoutItem` object.

The equivalent C# code, given a `ShellContent` object named `aboutItem`, is:

```
 currentItem = aboutItem;
```

In this example, the `CurrentItem` property is set in the subclassed `Shell` class. Alternatively, the `CurrentItem` property can be set in any class through the `shell.Current` static property:

```
 Shell.Current.CurrentItem = aboutItem;
```

NOTE

An application may enter a state where selecting a flyout item is not a valid operation. In such cases, the `FlyoutItem` can be disabled by setting its `IsEnabled` property to `false`. This will prevent users from being able to select the flyout item.

FlyoutItem visibility

Flyout items are visible in the flyout by default. However, an item can be hidden in the flyout with the `FlyoutItem.isVisible` property, and removed from the flyout with the `isVisible` property:

- `FlyoutItem.isVisible`, of type `bool`, indicates if the item is hidden in the flyout, but is still reachable with the `GoToAsync` navigation method. The default value of this property is `true`.
- `isVisible`, of type `bool`, indicates if the item should be removed from the visual tree and therefore not appear in the flyout. Its default value is `true`.

The following example shows hiding an item in the flyout:

```

<Shell ...>
    <FlyoutItem ...
        FlyoutItemisVisible="False">
        ...
    </FlyoutItem>
</Shell>

```

NOTE

There's also a `Shell.FlyoutItemIsVisible` attached property, which can be set on `FlyoutItem`, `MenuItem`, `Tab`, and `ShellContent` objects.

Open and close the flyout programmatically

The flyout can be programmatically opened and closed by setting the `Shell.FlyoutIsPresented` bindable property to a `boolean` value that indicates whether the flyout is currently open:

```
<Shell ...>
    FlyoutIsPresented="{Binding IsFlyoutOpen}"
</Shell>
```

Alternatively, this can be performed in code:

```
Shell.Current.FlyoutIsPresented = false;
```

Related links

- [Xaminals \(sample\)](#)
- [Xamarin.Forms style classes](#)
- [Xamarin.Forms visual state manager](#)
- [Xamarin.Forms brushes](#)
- [Xamarin.Forms Shell specific properties](#)

Xamarin.Forms Shell tabs

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

The navigation experience provided by Xamarin.Forms Shell is based on flyouts and tabs. The top level of navigation in a Shell application is either a flyout or a bottom tab bar, depending on the navigation requirements of the application. When the navigation experience for an application begins with bottom tabs, the child of the subclassed `Shell` object should be a `TabBar` object, which represents the bottom tab bar.

A `TabBar` object can contain one or more `Tab` objects, with each `Tab` object representing a tab on the bottom tab bar. Each `Tab` object can contain one or more `ShellContent` objects, with each `ShellContent` object displaying a single `ContentPage`. When more than one `ShellContent` object is present in a `Tab` object, the `ContentPage` objects will be navigable by top tabs. Within a tab, additional `ContentPage` objects that are known as detail pages, can be navigated to.

IMPORTANT

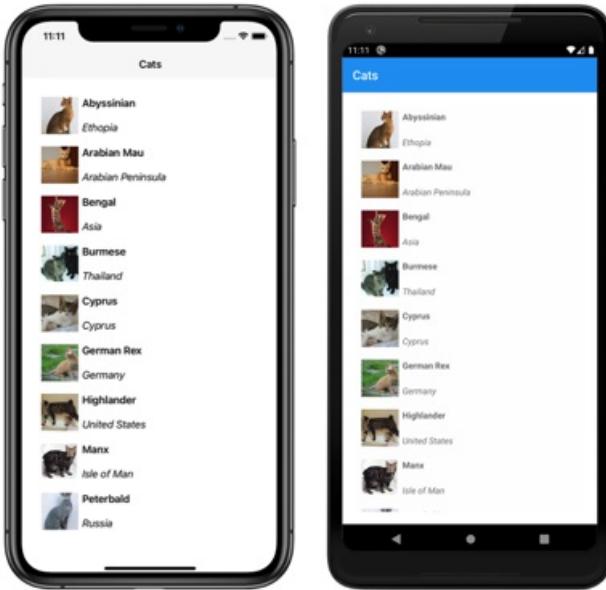
The `TabBar` type disables the flyout.

Single page

A single page Shell application can be created by adding a `Tab` object to a `TabBar` object. Within the `Tab` object, a `ShellContent` object should be set to a `ContentPage` object:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
      xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
      xmlns:views="clr-namespace:Xaminals.Views"
      x:Class="Xaminals.AppShell">
  <TabBar>
    <Tab>
      <ShellContent ContentTemplate="{DataTemplate views:CatsPage}" />
    </Tab>
  </TabBar>
</Shell>
```

This code example results in the following single page application:



Shell has implicit conversion operators that enable the Shell visual hierarchy to be simplified, without introducing additional views into the visual tree. This is possible because a subclassed `Shell` object can only ever contain `FlyoutItem` objects or a `TabBar` object, which can only ever contain `Tab` objects, which can only ever contain `ShellContent` objects. These implicit conversion operators can be used to remove the `Tab` objects from the previous example:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    <Tab>
        <ShellContent ContentTemplate="{DataTemplate views:CatsPage}" />
    </Tab>
</Shell>
```

This implicit conversion automatically wraps the `ShellContent` object in a `Tab` object, which is wrapped in a `TabBar` object.

IMPORTANT

In a Shell application, pages are created on demand in response to navigation. This is accomplished by using the `DataTemplate` markup extension to set the `ContentTemplate` property of each `ShellContent` object to a `ContentPage` object.

Bottom tabs

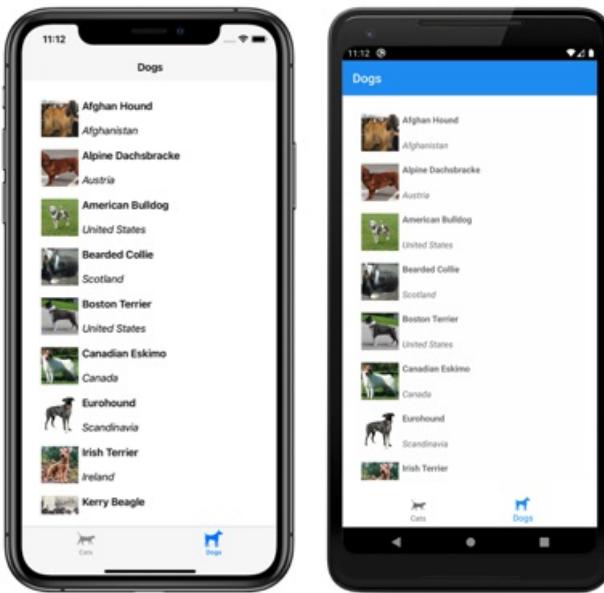
`Tab` objects are rendered as bottom tabs, provided that there are multiple `Tab` objects in a single `TabBar` object:

```

<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    <TabBar>
        <Tab Title="Cats"
              Icon="cat.png">
            <ShellContent ContentTemplate="{DataTemplate views:CatsPage}" />
        </Tab>
        <Tab Title="Dogs"
              Icon="dog.png">
            <ShellContent ContentTemplate="{DataTemplate views:DogsPage}" />
        </Tab>
    </TabBar>
</Shell>

```

The `Title` property, of type `string`, defines the tab title. The `Icon` property, of type `ImageSource`, defines the tab icon:



When there are more than five tabs on a `TabBar`, a `More` tab will appear, which can be used to access the additional tabs:



In addition, Shell's implicit conversion operators can be used to remove the `ShellContent` and `Tab` objects from the previous example:

```

<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    <TabBar>
        <ShellContent Title="Cats"
                      Icon="cat.png"
                      ContentTemplate="{DataTemplate views:CatsPage}" />
        <ShellContent Title="Dogs"
                      Icon="dog.png"
                      ContentTemplate="{DataTemplate views:DogsPage}" />
    </TabBar>
</Shell>

```

This implicit conversion automatically wraps each `ShellContent` object in a `Tab` object.

IMPORTANT

In a Shell application, pages are created on demand in response to navigation. This is accomplished by using the `DataTemplate` markup extension to set the `ContentTemplate` property of each `ShellContent` object to a `ContentPage` object.

Bottom and top tabs

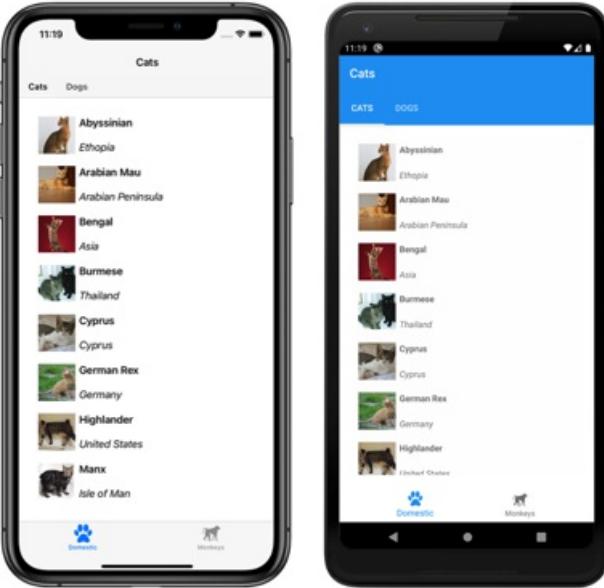
When more than one `ShellContent` object is present in a `Tab` object, a top tab bar is added to the bottom tab, through which the `ContentPage` objects are navigable:

```

<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    <TabBar>
        <Tab Title="Domestic"
             Icon="paw.png">
            <ShellContent Title="Cats"
                          ContentTemplate="{DataTemplate views:CatsPage}" />
            <ShellContent Title="Dogs"
                          ContentTemplate="{DataTemplate views:DogsPage}" />
        </Tab>
        <Tab Title="Monkeys"
             Icon="monkey.png">
            <ShellContent ContentTemplate="{DataTemplate views:MonkeysPage}" />
        </Tab>
    </TabBar>
</Shell>

```

This results in the layout shown in the following screenshots:



In addition, Shell's implicit conversion operators can be used to remove the second `Tab` object from the previous example:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    <TabBar>
        <Tab Title="Domestic"
             Icon="paw.png">
            <ShellContent Title="Cats"
                         Icon="cat.png"
                         ContentTemplate="{DataTemplate views:CatsPage}" />
            <ShellContent Title="Dogs"
                         Icon="dog.png"
                         ContentTemplate="{DataTemplate views:DogsPage}" />
        </Tab>
        <ShellContent Title="Monkeys"
                     Icon="monkey.png"
                     ContentTemplate="{DataTemplate views:MonkeysPage}" />
    </TabBar>
</Shell>
```

This implicit conversion automatically wraps the third `ShellContent` object in a `Tab` object.

Tab appearance

The `Shell` class defines the following attached properties that control the appearance of tabs:

- `TabBarBackgroundColor`, of type `Color`, that defines the background color for the tab bar. If the property is unset, the `BackgroundColor` property value is used.
- `TabBarDisabledColor`, of type `Color`, that defines the disabled color for the tab bar. If the property is unset, the `DisabledColor` property value is used.
- `TabBarForegroundColor`, of type `Color`, that defines the foreground color for the tab bar. If the property is unset, the `ForegroundColor` property value is used.
- `TabBarTitleColor`, of type `Color`, that defines the title color for the tab bar. If the property is unset, the `TitleColor` property value will be used.
- `TabBarUnselectedColor`, of type `Color`, that defines the unselected color for the tab bar. If the property is unset, the `UnselectedColor` property value is used.

All of these properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings, and styled.

The following example shows a XAML style that sets different tab bar color properties:

```
<Style TargetType="TabBar">
    <Setter Property="Shell.TabBarBackgroundColor"
           Value="CornflowerBlue" />
    <Setter Property="Shell.TabBarTitleColor"
           Value="Black" />
    <Setter Property="Shell.TabBarUnselectedColor"
           Value="AntiqueWhite" />
</Style>
```

In addition, tabs can also be styled using Cascading Style Sheets (CSS). For more information, see [Xamarin.Forms Shell specific properties](#).

Tab selection

When a Shell application that uses a tab bar is first run, the `Shell.CurrentItem` property will be set to the first `Tab` object in the subclassed `Shell` object. However, the property can be set to another `Tab`, as shown in the following example:

```
<Shell ...
    CurrentItem="{x:Reference dogsItem}">
    <TabBar>
        <ShellContent Title="Cats"
                      Icon="cat.png"
                      ContentTemplate="{DataTemplate views:CatsPage}" />
        <ShellContent x:Name="dogsItem"
                      Title="Dogs"
                      Icon="dog.png"
                      ContentTemplate="{DataTemplate views:DogsPage}" />
    </TabBar>
</Shell>
```

This example sets the `CurrentItem` property to the `ShellContent` object named `dogsItem`, which results in it being selected and displayed. In this example, an implicit conversion is used to wrap each `ShellContent` object in a `Tab` object.

The equivalent C# code, given a `ShellContent` object named `dogsItem`, is:

```
CurrentItem = dogsItem;
```

In this example, the `CurrentItem` property is set in the subclassed `Shell` class. Alternatively, the `CurrentItem` property can be set in any class through the `shell.Current` static property:

```
Shell.Current.CurrentItem = dogsItem;
```

TabBar and Tab visibility

The tab bar and tabs are visible in Shell applications by default. However, the tab bar can be hidden by setting the `Shell.TabBarIsVisible` attached property to `false`.

While this property can be set on a subclassed `Shell` object, it's typically set on any `ShellContent` or

`ContentPage` objects that want to make the tab bar invisible:

```
<TabBar>
    <Tab Title="Domestic"
        Icon="paw.png">
        <ShellContent Title="Cats"
            ContentTemplate="{DataTemplate views:CatsPage}" />
        <ShellContent Shell.TabBarIsVisible="false"
            Title="Dogs"
            ContentTemplate="{DataTemplate views:DogsPage}" />
    </Tab>
    <Tab Title="Monkeys"
        Icon="monkey.png">
        <ShellContent ContentTemplate="{DataTemplate views:MonkeysPage}" />
    </Tab>
</TabBar>
```

In this example, the tab bar is hidden when the upper **Dogs** tab is selected.

In addition, `Tab` objects can be hidden by setting the `Visible` bindable property to `false`:

```
<TabBar>
    <ShellContent Title="Cats"
        Icon="cat.png"
        ContentTemplate="{DataTemplate views:CatsPage}" />
    <ShellContent Title="Dogs"
        Icon="dog.png"
        ContentTemplate="{DataTemplate views:DogsPage}"
        IsVisible="False" />
    <ShellContent Title="Monkeys"
        Icon="monkey.png"
        ContentTemplate="{DataTemplate views:MonkeysPage}" />
</TabBar>
```

In this example, the second tab is hidden.

Related links

- [Xaminals \(sample\)](#)
- [Xamarin.Forms Shell navigation](#)
- [Xamarin.Forms CSS Shell specific properties](#)

Xamarin.Forms Shell pages

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

A `ShellContent` object represents the `ContentPage` object for each `FlyoutItem` or `Tab`. When more than one `ShellContent` object is present in a `Tab` object, the `ContentPage` objects will be navigable by top tabs. Within a page, additional `ContentPage` objects that are known as detail pages, can be navigated to.

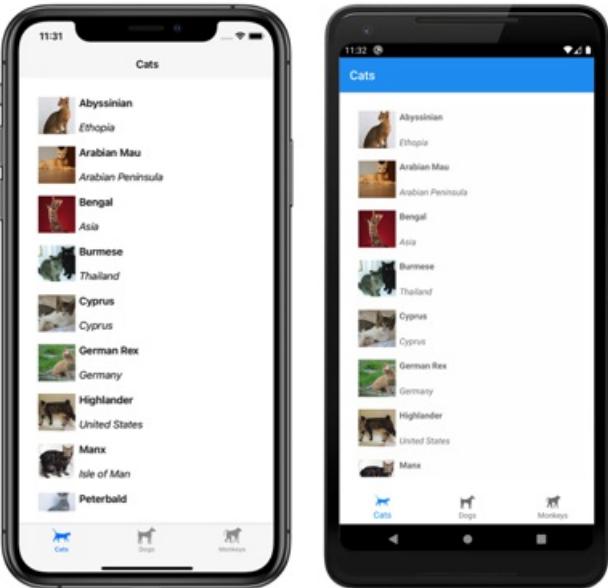
In addition, the `Shell` class defines attached properties that can be used to configure the appearance of pages in Xamarin.Forms Shell applications. This includes setting page colors, setting the page presentation mode, disabling the navigation bar, disabling the tab bar, and displaying views in the navigation bar.

Display pages

In Xamarin.Forms Shell applications, pages are typically created on demand in response to navigation. This is accomplished by using the `DataTemplate` markup extension to set the `ContentTemplate` property of each `ShellContent` object to a `ContentPage` object:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    <TabBar>
        <ShellContent Title="Cats"
                      Icon="cat.png"
                      ContentTemplate="{DataTemplate views:CatsPage}" />
        <ShellContent Title="Dogs"
                      Icon="dog.png"
                      ContentTemplate="{DataTemplate views:DogsPage}" />
        <ShellContent Title="Monkeys"
                      Icon="monkey.png"
                      ContentTemplate="{DataTemplate views:MonkeysPage}" />
    </TabBar>
</Shell>
```

In this example, Shell's implicit conversion operators are used to remove the `Tab` objects from the visual hierarchy. However, each `ShellContent` object is rendered in a tab:



NOTE

The `BindingContext` of each `ShellContent` object is inherited from the parent `Tab` object.

Within each `ContentPage` object, additional `ContentPage` objects can be navigated to. For more information about navigation, see [Xamarin.Forms Shell navigation](#).

Load pages at application startup

In a Shell application, each `ContentPage` object is typically created on demand, in response to navigation. However, it's also possible to create `ContentPage` objects at application startup.

WARNING

`ContentPage` objects that are created at application startup can lead to a poor startup experience.

`ContentPage` objects can be created at application startup by setting the `ShellContent.Content` properties to `ContentPage` objects:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
      xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
      xmlns:views="clr-namespace:Xaminals.Views"
      x:Class="Xaminals.AppShell">
<TabBar>
    <ShellContent Title="Cats"
                  Icon="cat.png">
        <views:CatsPage />
    </ShellContent>
    <ShellContent Title="Dogs"
                  Icon="dog.png">
        <views:DogsPage />
    </ShellContent>
    <ShellContent Title="Monkeys"
                  Icon="monkey.png">
        <views:MonkeysPage />
    </ShellContent>
</TabBar>
</Shell>
```

In this example, `CatsPage`, `DogsPage`, and `MonkeysPage` are all created at application startup, rather than on demand in response to navigation.

NOTE

The `Content` property is the content property of the `ShellContent` class, and therefore does not need to be explicitly set.

Set page colors

The `Shell` class defines the following attached properties that can be used to set page colors in a Shell application:

- `BackgroundColor`, of type `Color`, that defines the background color in the Shell chrome. The color will not fill in behind the Shell content.
- `DisabledColor`, of type `Color`, that defines the color to shade text and icons that are disabled.
- `ForegroundColor`, of type `Color`, that defines the color to shade text and icons.
- `TitleColor`, of type `Color`, that defines the color used for the title of the current page.
- `UnselectedColor`, of type `Color`, that defines the color used for unselected text and icons in the Shell chrome.

All of these properties are backed by `BindableProperty` objects, which mean that the properties can be targets of data bindings, and styled using XAML styles. In addition, the properties can be set using Cascading Style Sheets (CSS). For more information, see [Xamarin.Forms Shell specific properties](#).

NOTE

There are also properties that enable tab colors to be defined. For more information, see [Tab appearance](#).

The following XAML shows setting the color properties in a subclassed `Shell` class:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       x:Class="Xaminals.AppShell"
       BackgroundColor="#455A64"
       ForegroundColor="White"
       TitleColor="White"
       DisabledColor="#B4FFFFFF"
       UnselectedColor="#95FFFFFF">

</Shell>
```

In this example, the color values will be applied to all pages in the Shell application, unless overridden at the page level.

Because the color properties are attached properties, they can also be set on individual pages, to set the colors on that page:

```

<ContentPage ...
    Shell.BackgroundColor="Gray"
    Shell.ForegroundColor="White"
    Shell.TitleColor="Blue"
    Shell.DisabledColor="#95FFFFFF"
    Shell.UnselectedColor="#B4FFFFFF">

</ContentPage>

```

Alternatively, the color properties can be set with a XAML style:

```

<Style x:Key="DomesticShell"
    TargetType="Element" >
    <Setter Property="Shell.BackgroundColor"
        Value="#039BE6" />
    <Setter Property="Shell.ForegroundColor"
        Value="White" />
    <Setter Property="Shell.TitleColor"
        Value="White" />
    <Setter Property="Shell.DisabledColor"
        Value="#B4FFFFFF" />
    <Setter Property="Shell.UnselectedColor"
        Value="#95FFFFFF" />
</Style>

```

For more information about XAML styles, see [Styling Xamarin.Forms Apps using XAML Styles](#).

Set page presentation mode

By default, a small navigation animation occurs when a page is navigated to with the [GoToAsync](#) method.

However, this behavior can be changed by setting the [Shell.PresentationMode](#) attached property on a [ContentPage](#) to one of the [PresentationMode](#) enumeration members:

- [NotAnimated](#) indicates that the page will be displayed without a navigation animation.
- [Animated](#) indicates that the page will be displayed with a navigation animation. This is the default value of the [Shell.PresentationMode](#) attached property.
- [Modal](#) indicates that the page will be displayed as a modal page.
- [ModalAnimated](#) indicates that the page will be displayed as a modal page, with a navigation animation.
- [ModalNotAnimated](#) indicates that the page will be displayed as a modal page, without a navigation animation.

IMPORTANT

The [PresentationMode](#) type is a flags enumeration. This means that a combination of enumeration members can be applied in code. However, for ease of use in XAML, the [ModalAnimated](#) member is a combination of the [Animated](#) and [Modal](#) members, and the [ModalNotAnimated](#) member is a combination of the [NotAnimated](#) and [Modal](#) members.

For more information about flag enumerations, see [Enumeration types as bit flags](#).

The following XAML example sets the [Shell.PresentationMode](#) attached property on a [ContentPage](#):

```

<ContentPage ...
    Shell.PresentationMode="Modal">
    ...
</ContentPage>

```

In this example, the [ContentPage](#) is set to be displayed as a modal page, when the page is navigated to with the

[GoToAsync](#) method.

Enable navigation bar shadow

The `Shell.NavBarHasShadow` attached property, of type `bool`, controls whether the navigation bar has a shadow. By default the value of the property is `false` on iOS, and `true` on Android.

While this property can be set on a subclassed `Shell` object, it can also be set on any pages that want to enable the navigation bar shadow. For example, the following XAML shows enabling the navigation bar shadow from a `ContentPage`:

```
<ContentPage ...>
    Shell.NavBarHasShadow="true"
    ...
</ContentPage>
```

This results in the navigation bar shadow being enabled.

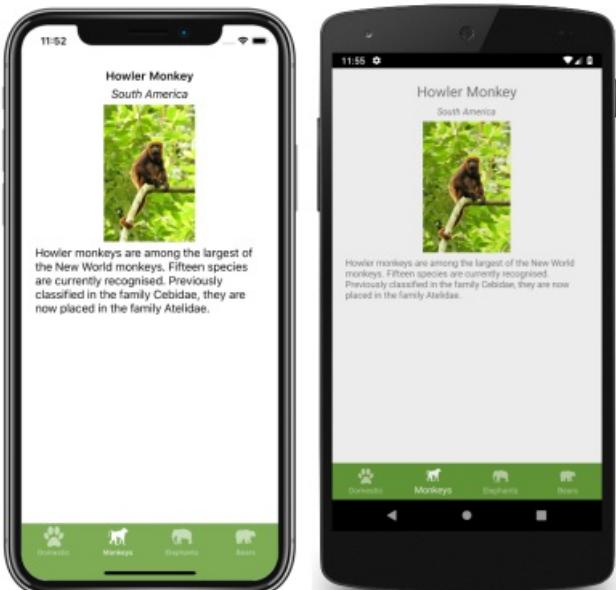
Disable the navigation bar

The `Shell.NavBarIsVisible` attached property, of type `bool`, controls if the navigation bar is visible when a page is presented. By default the value of the property is `true`.

While this property can be set on a subclassed `Shell` object, it's typically set on any pages that want to make the navigation bar invisible. For example, the following XAML shows disabling the navigation bar from a `ContentPage`:

```
<ContentPage ...>
    Shell.NavBarIsVisible="false"
    ...
</ContentPage>
```

This results in the navigation bar becoming invisible when the page is presented:



Display views in the navigation bar

The `Shell.TitleView` attached property, of type `View`, enables any `View` to be displayed in the navigation bar.

While this property can be set on a subclassed `Shell` object, it can also be set on any pages that want to display a view in the navigation bar. For example, the following XAML shows displaying an `Image` in the navigation bar of a `ContentPage`:

```
<ContentPage ...>
    <Shell.TitleView>
        <Image Source="xamarin_logo.png"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
    </Shell.TitleView>
    ...
</ContentPage>
```

This results in an image being displayed in the navigation bar on the page:



IMPORTANT

If the navigation bar has been made invisible, with the `NavBarVisible` attached property, the title view will not be displayed.

Many views won't appear in the navigation bar unless the size of the view is specified with the `WidthRequest` and `HeightRequest` properties, or the location of the view is specified with the `HorizontalOptions` and `VerticalOptions` properties.

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. Similarly, because the `ContentView` class ultimately derives from the `View` class, the `TitleView` attached property can be set to display a `ContentView` that contains a single view.

Page visibility

Shell respects page visibility, set with the `IsVisible` property. Therefore, when a page's `IsVisible` property is set to `false` it won't be visible in the Shell application and it won't be possible to navigate to it.

Related links

- [Xaminals \(sample\)](#)
- [Xamarin.Forms Shell navigation](#)
- [Styling Xamarin.Forms Apps using XAML Styles](#)
- [Xamarin.Forms CSS Shell specific properties](#)

Xamarin.Forms Shell navigation

8/4/2022 • 15 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms Shell includes a URI-based navigation experience that uses routes to navigate to any page in the application, without having to follow a set navigation hierarchy. In addition, it also provides the ability to navigate backwards without having to visit all of the pages on the navigation stack.

The `Shell` class defines the following navigation-related properties:

- `BackButtonBehavior`, of type `BackButtonBehavior`, an attached property that defines the behavior of the back button.
- `CurrentItem`, of type `ShellItem`, the currently selected item.
- `CurrentPage`, of type `Page`, the currently presented page.
- `CurrentState`, of type `ShellNavigationState`, the current navigation state of the `Shell`.
- `Current`, of type `Shell`, a type-casted alias for `Application.Current.MainPage`.

The `BackButtonBehavior`, `CurrentItem`, and `CurrentState` properties are backed by `BindableProperty` objects, which means that these properties can be targets of data bindings.

Navigation is performed by invoking the `GoToAsync` method, from the `Shell` class. When navigation is about to be performed, the `Navigating` event is fired, and the `Navigated` event is fired when navigation completes.

NOTE

Navigation can still be performed between pages in a Shell application by using the `Navigation` property. For more information, see [Hierarchical Navigation](#).

Routes

Navigation is performed in a Shell application by specifying a URI to navigate to. Navigation URIs can have three components:

- A *route*, which defines the path to content that exists as part of the Shell visual hierarchy.
- A *page*. Pages that don't exist in the Shell visual hierarchy can be pushed onto the navigation stack from anywhere within a Shell application. For example, a details page won't be defined in the Shell visual hierarchy, but can be pushed onto the navigation stack as required.
- One or more *query parameters*. Query parameters are parameters that can be passed to the destination page while navigating.

When a navigation URI includes all three components, the structure is: //route/page?queryParameters

Register routes

Routes can be defined on `FlyoutItem`, `TabBar`, `Tab`, and `ShellContent` objects, through their `Route` properties:

```

<Shell ...>
    <FlyoutItem ...>
        <Tab ...>
            <Route="animals">
                <ShellContent ...>
                    <Route="domestic">
                        <ShellContent ...>
                            <Route="cats" />
                        <ShellContent ...>
                            <Route="dogs" />
                    </ShellContent>
                <ShellContent ...>
                    <Route="monkeys" />
                <ShellContent ...>
                    <Route="elephants" />
                <ShellContent ...>
                    <Route="bears" />
                </FlyoutItem>
                <ShellContent ...>
                    <Route="about" />
                ...
            </Shell>

```

NOTE

All items in the Shell hierarchy have a route associated with them. If you don't set a route, one is generated at runtime. However, generated routes are not guaranteed to be consistent across different application sessions.

The above example creates the following route hierarchy, which can be used in programmatic navigation:

```

animals
  domestic
    cats
    dogs
  monkeys
  elephants
  bears
about

```

To navigate to the `ShellContent` object for the `dogs` route, the absolute route URI is `//animals/domestic/dogs`. Similarly, to navigate to the `ShellContent` object for the `about` route, the absolute route URI is `//about`.

WARNING

An `ArgumentException` will be thrown on application startup if a duplicate route is detected. This exception will also be thrown if two or more routes at the same level in the hierarchy share a route name.

Register detail page routes

In the `Shell` subclass constructor, or any other location that runs before a route is invoked, additional routes can be explicitly registered for any detail pages that aren't represented in the Shell visual hierarchy. This is accomplished with the `Routing.RegisterRoute` method:

```

Routing.RegisterRoute("monkeydetails", typeof(MonkeyDetailPage));
Routing.RegisterRoute("beardetails", typeof(BearDetailPage));
Routing.RegisterRoute("catdetails", typeof(CatDetailPage));
Routing.RegisterRoute("dogdetails", typeof(DogDetailPage));
Routing.RegisterRoute("elephantdetails", typeof(ElephantDetailPage));

```

This example registers detail pages, that aren't defined in the `Shell` subclass, as routes. These detail pages can then be navigated to using URI-based navigation, from anywhere within the application. The routes for such pages are known as *global routes*.

WARNING

An `ArgumentException` will be thrown if the `Routing.RegisterRoute` method attempts to register the same route to two or more different types.

Alternatively, pages can be registered at different route hierarchies if required:

```
Routing.RegisterRoute("monkeys/details", typeof(MonkeyDetailPage));
Routing.RegisterRoute("bears/details", typeof(BearDetailPage));
Routing.RegisterRoute("cats/details", typeof(CatDetailPage));
Routing.RegisterRoute("dogs/details", typeof(DogDetailPage));
Routing.RegisterRoute("elephants/details", typeof(ElephantDetailPage));
```

This example enables contextual page navigation, where navigating to the `details` route from the page for the `monkeys` route displays the `MonkeyDetailPage`. Similarly, navigating to the `details` route from the page for the `elephants` route displays the `ElephantDetailPage`. For more information, see [Contextual navigation](#).

NOTE

Pages whose routes have been registered with the `Routing.RegisterRoute` method can be deregistered with the `Routing.UnRegisterRoute` method, if required.

Perform navigation

To perform navigation, a reference to the `Shell` subclass must first be obtained. This reference can be obtained by casting the `App.Current.MainPage` property to a `Shell` object, or through the `Shell.Current` property. Navigation can then be performed by calling the `GoToAsync` method on the `Shell` object. This method navigates to a `ShellNavigationState` and returns a `Task` that will complete once the navigation animation has completed. The `ShellNavigationState` object is constructed by the `GoToAsync` method, from a `string`, or a `Uri`, and it has its `Location` property set to the `string` or `Uri` argument.

IMPORTANT

When a route from the Shell visual hierarchy is navigated to, a navigation stack isn't created. However, when a page that's not in the Shell visual hierarchy is navigated to, a navigation stack is created.

The current navigation state of the `Shell` object can be retrieved through the `Shell.Current.CurrentState` property, which includes the URI of the displayed route in the `Location` property.

Absolute routes

Navigation can be performed by specifying a valid absolute URI as an argument to the `GoToAsync` method:

```
await Shell.Current.GoToAsync("//animals/monkeys");
```

This example navigates to the page for the `monkeys` route, with the route being defined on a `ShellContent` object. The `ShellContent` object that represents the `monkeys` route is a child of a `FlyoutItem` object, whose route is `animals`.

Relative routes

Navigation can also be performed by specifying a valid relative URI as an argument to the `GoToAsync` method. The routing system will attempt to match the URI to a `ShellContent` object. Therefore, if all the routes in an application are unique, navigation can be performed by only specifying the unique route name as a relative URI.

The following relative route formats are supported:

FORMAT	DESCRIPTION
<code>route</code>	The route hierarchy will be searched for the specified route, upwards from the current position. The matching page will be pushed to the navigation stack.
<code>/route</code>	The route hierarchy will be searched from the specified route, downwards from the current position. The matching page will be pushed to the navigation stack.
<code>//route</code>	The route hierarchy will be searched for the specified route, upwards from the current position. The matching page will replace the navigation stack.
<code>///route</code>	The route hierarchy will be searched for the specified route, downwards from the current position. The matching page will replace the navigation stack.

The following example navigates to the page for the `monkeydetails` route:

```
await Shell.Current.GoToAsync("monkeydetails");
```

In this example, the `monkeyDetails` route is searched for up the hierarchy until the matching page is found. When the page is found, it's pushed to the navigation stack.

Contextual navigation

Relative routes enable contextual navigation. For example, consider the following route hierarchy:

```
monkeys
  details
bears
  details
```

When the registered page for the `monkeys` route is displayed, navigating to the `details` route will display the registered page for the `monkeys/details` route. Similarly, when the registered page for the `bears` route is displayed, navigating to the `details` route will display the registered page for the `bears/details` route. For information on how to register the routes in this example, see [Register page routes](#).

Backwards navigation

Backwards navigation can be performed by specifying `".."` as the argument to the `GoToAsync` method:

```
await Shell.Current.GoToAsync(..);
```

Backwards navigation with `".."` can also be combined with a route:

```
await Shell.Current.GoToAsync("../route");
```

In this example, backwards navigation is performed, and then navigation to the specified route.

IMPORTANT

Navigating backwards and into a specified route is only possible if the backwards navigation places you at the current location in the route hierarchy to navigate to the specified route.

Similarly, it's possible to navigate backwards multiple times, and then navigate to a specified route:

```
await Shell.Current.GoToAsync("../route");
```

In this example, backwards navigation is performed twice, and then navigation to the specified route.

In addition, data can be passed through query properties when navigating backwards:

```
await Shell.Current.GoToAsync($"..?parameterToPassBack={parameterValueToPassBack}");
```

In this example, backwards navigation is performed, and the query parameter value is passed to the query parameter on the previous page.

NOTE

Query parameters can be appended to any backwards navigation request.

For more information about passing data when navigating, see [Pass data](#).

Invalid routes

The following route formats are invalid:

FORMAT	EXPLANATION
//page or ///page	Global routes currently can't be the only page on the navigation stack. Therefore, absolute routing to global routes is unsupported.

Use of these route formats results in an `Exception` being thrown.

WARNING

Attempting to navigate to a non-existent route results in an `ArgumentException` exception being thrown.

Debugging navigation

Some of the Shell classes are decorated with the `DebuggerDisplayAttribute`, which specifies how a class or field is displayed by the debugger. This can help to debug navigation requests by displaying data related to the navigation request. For example, the following screenshot shows the `CurrentItem` and `CurrentState` properties of the `Shell.Current` object:

- ▶ `P CurrentItem` Title = Animals, Route = animals
- ▶ `P CurrentState` Location = {app://xamarin.com/xaminals/animals/domestic/cats}

In this example, the `CurrentItem` property, of type `FlyoutItem`, displays the title and route of the `FlyoutItem` object. Similarly, the `CurrentState` property, of type `ShellNavigationState`, displays the URI of the displayed route within the Shell application.

Navigation stack

The `Tab` class defines a `Stack` property, of type `IReadOnlyList<Page>`, which represents the current navigation stack within the `Tab`. The class also provides the following overridable navigation methods:

- `GetNavigationStack`, returns `IReadOnlyList<Page>`, the current navigation stack.
- `OnInsertPageBefore`, that's called when `INavigation.InsertPageBefore` is called.
- `OnPopAsync`, returns `Task<Page>`, and is called when `INavigation.PopAsync` is called.
- `OnPopToRootAsync`, returns `Task`, and is called when `INavigation.OnPopToRootAsync` is called.
- `OnPushAsync`, returns `Task`, and is called when `INavigation.PushAsync` is called.
- `OnRemovePage`, that's called when `INavigation.RemovePage` is called.

The following example shows how to override the `OnRemovePage` method:

```
public class MyTab : Tab
{
    protected override void OnRemovePage(Page page)
    {
        base.OnRemovePage(page);

        // Custom logic
    }
}
```

In this example, `MyTab` objects should be consumed in your Shell visual hierarchy instead of `Tab` objects.

Navigation events

The `Shell` class defines the `Navigating` event, which is fired when navigation is about to be performed, either due to programmatic navigation or user interaction. The `ShellNavigatingEventArgs` object that accompanies the `Navigating` event provides the following properties:

PROPERTY	TYPE	DESCRIPTION
<code>Current</code>	<code>ShellNavigationState</code>	The URI of the current page.
<code>Source</code>	<code>ShellNavigationSource</code>	The type of navigation that occurred.
<code>Target</code>	<code>ShellNavigationState</code>	The URI representing where the navigation is destined.
<code>CanCancel</code>	<code>bool</code>	A value indicating if it's possible to cancel the navigation.
<code>Cancelled</code>	<code>bool</code>	A value indicating if the navigation was canceled.

In addition, the `ShellNavigatingEventArgs` class provides a `Cancel` method that can be used to cancel navigation, and a `GetDeferral` method that returns a `ShellNavigatingDeferral` token that can be used to complete navigation. For more information about navigation deferral, see [Navigation deferral](#).

The `Shell` class also defines the `Navigated` event, which is fired when navigation has completed. The `ShellNavigatedEventArgs` object that accompanies the `Navigated` event provides the following properties:

PROPERTY	TYPE	DESCRIPTION
<code>Current</code>	<code>ShellNavigationState</code>	The URI of the current page.
<code>Previous</code>	<code>ShellNavigationState</code>	The URI of the previous page.
<code>Source</code>	<code>ShellNavigationState</code>	The type of navigation that occurred.

IMPORTANT

The `OnNavigating` method is called when the `Navigating` event fires. Similarly, the `OnNavigated` method is called when the `Navigated` event fires. Both methods can be overridden in your `Shell` subclass to intercept navigation requests.

The `ShellNavigatedEventArgs` and `ShellNavigatingEventArgs` classes both have `Source` properties, of type `ShellNavigationSource`. This enumeration provides the following values:

- `Unknown`
- `Push`
- `Pop`
- `PopToRoot`
- `Insert`
- `Remove`
- `ShellItemChanged`
- `ShellSectionChanged`
- `ShellContentChanged`

Therefore, navigation can be intercepted in an `OnNavigating` override and actions can be performed based on the navigation source. For example, the following code shows how to cancel backwards navigation if the data on the page is unsaved:

```
protected override void OnNavigating(ShellNavigatingEventArgs args)
{
    base.OnNavigating(args);

    // Cancel any back navigation.
    if (args.Source == ShellNavigationSource.Pop)
    {
        args.Cancel();
    }
}
```

Navigation deferral

Shell navigation can be intercepted and completed or canceled based on user choice. This can be achieved by overriding the `OnNavigating` method in your `Shell` subclass, and by calling the `GetDeferral` method on the `ShellNavigatingEventArgs` object. This method returns a `ShellNavigatingDeferral` token that has a `Complete` method, which can be used to complete the navigation request:

```

public MyShell : Shell
{
    // ...
    protected override async void OnNavigating(ShellNavigatingEventArgs args)
    {
        base.OnNavigating(args);

        ShellNavigatingDeferral token = args.GetDeferral();

        var result = await DisplayActionSheet("Navigate?", "Cancel", "Yes", "No");
        if (result != "Yes")
        {
            args.Cancel();
        }
        token.Complete();
    }
}

```

In this example, an action sheet is displayed that invites the user to complete the navigation request, or cancel it. Navigation is canceled by invoking the `Cancel` method on the `ShellNavigatingEventArgs` object. Navigation is completed by invoking the `Complete` method on the `ShellNavigatingDeferral` token that was retrieved by the `GetDeferral` method on the `ShellNavigatingEventArgs` object.

WARNING

The `GoToAsync` method will throw a `InvalidOperationException` if a user tries to navigate while there is a pending navigation deferral.

Pass data

Data can be passed as query parameters when performing URI-based programmatic navigation. This is achieved by appending `?` after a route, followed by a query parameter id, `=`, and a value. For example, the following code is executed in the sample application when a user selects an elephant on the `ElephantsPage`:

```

async void OnCollectionViewSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    string elephantName = (e.CurrentSelection.FirstOrDefault() as Animal).Name;
    await Shell.Current.GoToAsync($"elephantdetails?name={elephantName}");
}

```

This code example retrieves the currently selected elephant in the `CollectionView`, and navigates to the `elephantdetails` route, passing `elephantName` as a query parameter.

There are two approaches to receiving navigation data:

1. The class that represents the page being navigated to, or the class for the page's `BindingContext`, can be decorated with a `QueryPropertyAttribute` for each query parameter. For more information, see [Process navigation data using query property attributes](#).
2. The class that represents the page being navigated to, or the class for the page's `BindingContext`, can implement the `IQueryAttributable` interface. For more information, see [Process navigation data using a single method](#).

Process navigation data using query property attributes

Navigation data can be received by decorating the receiving class with a `QueryPropertyAttribute` for each query parameter:

```
[QueryProperty(nameof(Name), "name")]
public partial class ElephantDetailPage : ContentPage
{
    public string Name
    {
        set
        {
            LoadAnimal(value);
        }
    }
    ...
    void LoadAnimal(string name)
    {
        try
        {
            Animal animal = ElephantData.Elephants.FirstOrDefault(a => a.Name == name);
            BindingContext = animal;
        }
        catch (Exception)
        {
            Console.WriteLine("Failed to load animal.");
        }
    }
}
```

The first argument for the `QueryPropertyAttribute` specifies the name of the property that will receive the data, with the second argument specifying the query parameter id. Therefore, the `QueryPropertyAttribute` in the above example specifies that the `Name` property will receive the data passed in the `name` query parameter from the URI in the `GoToAsync` method call. The `Name` property setter calls the `LoadAnimal` method to retrieve the `Animal` object for the `name`, and sets it as the `BindingContext` of the page.

NOTE

Query parameter values that are received via the `QueryPropertyAttribute` are automatically URL decoded.

Process navigation data using a single method

Navigation data can be received by implementing the `IQueryAttributable` interface on the receiving class. The `IQueryAttributable` interface specifies that the implementing class must implement the `ApplyQueryAttributes` method. This method has a `query` argument, of type `IDictionary<string, string>`, that contains any data passed during navigation. Each key in the dictionary is a query parameter id, with its value being the query parameter value. The advantage of using this approach is that navigation data can be processed using a single method, which can be useful when you have multiple items of navigation data that require processing as a whole.

The following example shows a view model class that implements the `IQueryAttributable` interface:

```

public class MonkeyDetailViewModel : IQueryAttributable, INotifyPropertyChanged
{
    public Animal Monkey { get; private set; }

    public void ApplyQueryAttributes(IDictionary<string, string> query)
    {
        // The query parameter requires URL decoding.
        string name = HttpUtility.UrlDecode(query["name"]);
        LoadAnimal(name);
    }

    void LoadAnimal(string name)
    {
        try
        {
            Monkey = MonkeyData.Monkeys.FirstOrDefault(a => a.Name == name);
            OnPropertyChanged("Monkey");
        }
        catch (Exception)
        {
            Console.WriteLine("Failed to load animal.");
        }
    }
    ...
}

```

In this example, the `ApplyQueryAttributes` method retrieves the value of the `name` query parameter from the URI in the `GoToAsync` method call. Then, the `LoadAnimal` method is called to retrieve the `Animal` object, where its set as the value of the `Monkey` property that is data bound to.

IMPORTANT

Query parameter values that are received via the `IQueryAttributable` interface aren't automatically URL decoded.

Pass and process multiple query parameters

Multiple query parameters can be passed by connecting them with `&`. For example, the following code passes two data items:

```

async void OnCollectionViewSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    string elephantName = (e.CurrentSelection.FirstOrDefault() as Animal).Name;
    string elephantLocation = (e.CurrentSelection.FirstOrDefault() as Animal).Location;
    await Shell.Current.GoToAsync($"elephantdetails?name={elephantName}&location={elephantLocation}");
}

```

This code example retrieves the currently selected elephant in the `CollectionView`, and navigates to the `elephantdetails` route, passing `elephantName` and `elephantLocation` as query parameters.

To receive multiple items of data, the class that represents the page being navigated to, or the class for the page's `BindingContext`, can be decorated with a `QueryPropertyAttribute` for each query parameter:

```

[QueryProperty(nameof(Name), "name")]
[QueryProperty(nameof(Location), "location")]
public partial class ElephantDetailPage : ContentPage
{
    public string Name
    {
        set
        {
            // Custom logic
        }
    }

    public string Location
    {
        set
        {
            // Custom logic
        }
    }
    ...
}

```

In this example, the class is decorated with a `QueryPropertyAttribute` for each query parameter. The first `QueryPropertyAttribute` specifies that the `Name` property will receive the data passed in the `name` query parameter, while the second `QueryPropertyAttribute` specifies that the `Location` property will receive the data passed in the `location` query parameter. In both cases, the query parameter values are specified in the URI in the `GoToAsync` method call.

Alternatively, navigation data can be processed by a single method by implementing the `IQueryAttributable` interface on the class that represents the page being navigated to, or the class for the page's `BindingContext`:

```

public class ElephantDetailViewModel : IQueryAttributable, INotifyPropertyChanged
{
    public Animal Elephant { get; private set; }

    public void ApplyQueryAttributes(IDictionary<string, string> query)
    {
        string name = HttpUtility.UrlDecode(query["name"]);
        string location = HttpUtility.UrlDecode(query["location"]);
        ...
    }
    ...
}

```

In this example, the `ApplyQueryAttributes` method retrieves the value of the `name` and `location` query parameters from the URI in the `GoToAsync` method call.

Back button behavior

Back button appearance and behavior can be redefined by setting the `BackButtonBehavior` attached property to a `BackButtonBehavior` object. The `BackButtonBehavior` class defines the following properties:

- `Command`, of type `ICommand`, which is executed when the back button is pressed.
- `CommandParameter`, of type `object`, which is the parameter that's passed to the `Command`.
- `IconOverride`, of type `ImageSource`, the icon used for the back button.
- `IsEnabled`, of type `boolean`, indicates whether the back button is enabled. The default value is `true`.
- `TextOverride`, of type `string`, the text used for the back button.

All of these properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

The following code shows an example of redefining back button appearance and behavior:

```
<ContentPage ...>
    <Shell.BackButtonBehavior>
        <BackButtonBehavior Command="{Binding BackCommand}"
                            IconOverride="back.png" />
    </Shell.BackButtonBehavior>
    ...
</ContentPage>
```

The equivalent C# code is:

```
Shell.SetBackButtonBehavior(this, new BackButtonBehavior
{
    Command = new Command(() =>
    {
        ...
    }),
    IconOverride = "back.png"
});
```

The `Command` property is set to an `ICommand` to be executed when the back button is pressed, and the `IconOverride` property is set to the icon that's used for the back button:



Related links

- [Xaminals \(sample\)](#)

Xamarin.Forms Shell search

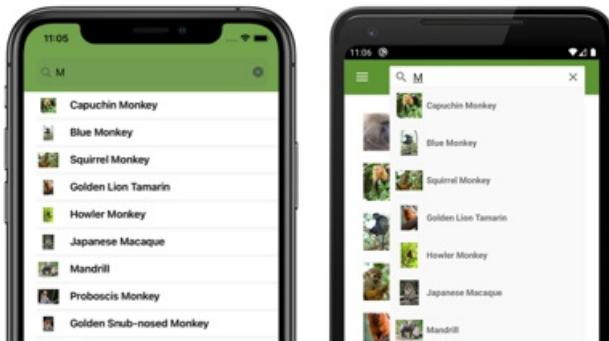
8/4/2022 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

Xamarin.Forms Shell includes integrated search functionality that's provided by the `SearchHandler` class. Search capability can be added to a page by setting the `Shell.SearchHandler` attached property to a subclassed `SearchHandler` object. This results in a search box being added at the top of the page:



When a query is entered into the search box, the `Query` property is updated, and on each update the `OnQueryChanged` method is executed. This method can be overridden to populate the search suggestions area with data:



Then, when a result is selected from the search suggestions area, the `OnItemSelected` method is executed. This method can be overridden to respond appropriately, such as by navigating to a detail page.

Create a SearchHandler

Search functionality can be added to a Shell application by subclassing the `SearchHandler` class, and overriding the `OnQueryChanged` and `OnItemSelected` methods:

```

public class AnimalSearchHandler : SearchHandler
{
    public IList<Animal> Animals { get; set; }
    public Type SelectedItemNavigationTarget { get; set; }

    protected override void OnQueryChanged(string oldValue, string newValue)
    {
        base.OnQueryChanged(oldValue, newValue);

        if (string.IsNullOrWhiteSpace(newValue))
        {
            ItemsSource = null;
        }
        else
        {
            ItemsSource = Animals
                .Where(animal => animal.Name.ToLower().Contains(newValue.ToLower()))
                .ToList<Animal>();
        }
    }

    protected override async void OnItemSelected(object item)
    {
        base.OnItemSelected(item);

        // Let the animation complete
        await Task.Delay(1000);

        ShellNavigationState state = (App.Current.MainPage as Shell).CurrentState;
        // The following route works because route names are unique in this application.
        await Shell.Current.GoToAsync($"{GetNavigationTarget()}?name={((Animal)item).Name}");
    }

    string GetNavigationTarget()
    {
        return (Shell.Current as AppShell).Routes.FirstOrDefault(route =>
route.Value.Equals(SelectedItemNavigationTarget)).Key;
    }
}

```

The `OnQueryChanged` override has two arguments: `oldValue`, which contains the previous search query, and `newValue`, which contains the current search query. The search suggestions area can be updated by setting the `SearchHandler.ItemsSource` property to an `IEnumerable` collection that contains items that match the current search query.

When a search result is selected by the user, the `OnItemSelected` override is executed and the `SelectedItem` property is set. In this example, the method navigates to another page that displays data about the selected `Animal`. For more information about navigation, see [Xamarin.Forms Shell navigation](#).

NOTE

Additional `SearchHandler` properties can be set to control the search box appearance.

Consume a SearchHandler

The subclassed `SearchHandler` can be consumed by setting the `Shell.SearchHandler` attached property to an object of the subclassed type, on the consuming page:

```

<ContentPage ...
    xmlns:controls="clr-namespace:Xaminals.Controls">
    <Shell.SearchHandler>
        <controls:AnimalSearchHandler Placeholder="Enter search term"
            ShowsResults="true"
            DisplayMemberName="Name" />
    </Shell.SearchHandler>
    ...
</ContentPage>

```

The equivalent C# code is:

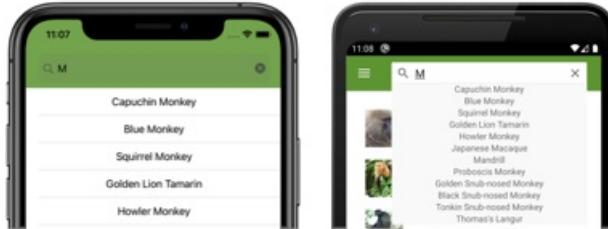
```

Shell.SetSearchHandler(this, new AnimalSearchHandler
{
    Placeholder = "Enter search term",
    ShowsResults = true,
    DisplayMemberName = "Name"
});

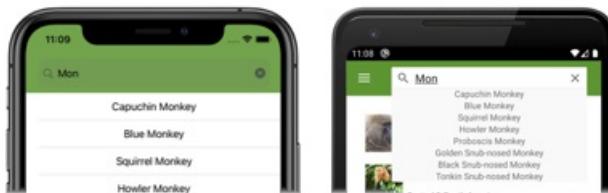
```

The `AnimalSearchHandler.OnQueryChanged` method returns a `List` of `Animal` objects. The `DisplayMemberName` property is set to the `Name` property of each `Animal` object, and so the data displayed in the suggestions area will be each animal name.

The `ShowsResults` property is set to `true`, so that search suggestions are displayed as the user enters a search query:



As the search query changes, the search suggestions area is updated:



When a search result is selected, the `MonkeyDetailPage` is navigated to, and a detail page about the selected monkey is displayed:



Define search results item appearance

In addition to displaying `string` data in the search results, the appearance of each search result item can be defined by setting the `SearchHandler.ItemTemplate` property to a `DataTemplate`:

```
<ContentPage ...>
    <xamlns:controls="clr-namespace:Xaminals.Controls">
        <Shell.SearchHandler>
            <controls:AnimalSearchHandler Placeholder="Enter search term"
                ShowsResults="true">
                <controls:AnimalSearchHandler.ItemTemplate>
                    <DataTemplate>
                        <Grid Padding="10"
                            ColumnDefinitions="0.15*,0.85*>
                            <Image Source="{Binding ImageUrl}"
                                HeightRequest="40"
                                WidthRequest="40" />
                            <Label Grid.Column="1"
                                Text="{Binding Name}"
                                FontAttributes="Bold"
                                VerticalOptions="Center" />
                    </Grid>
                </DataTemplate>
            </controls:AnimalSearchHandler.ItemTemplate>
        </controls:AnimalSearchHandler>
    </Shell.SearchHandler>
    ...
</ContentPage>
```

The equivalent C# code is:

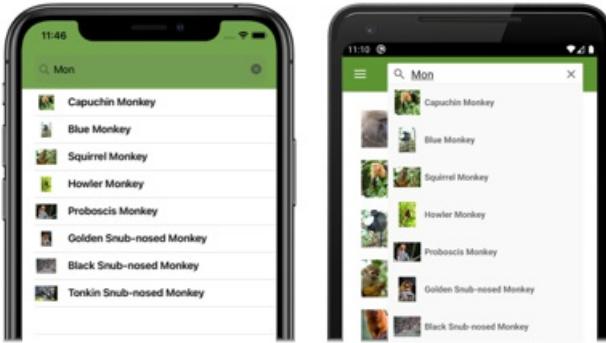
```
Shell.SetSearchHandler(this, new AnimalSearchHandler
{
    Placeholder = "Enter search term",
    ShowsResults = true,
    ItemTemplate = new DataTemplate(() =>
    {
        Grid grid = new Grid { Padding = 10 };
        grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength(0.15, GridUnitType.Star) });
        grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength(0.85, GridUnitType.Star) });

        Image image = new Image { HeightRequest = 40, WidthRequest = 40 };
        image.SetBinding(Image.SourceProperty, "ImageUrl");
        Label nameLabel = new Label { FontAttributes = FontAttributes.Bold, VerticalOptions =
LayoutOptions.Center };
        nameLabel.SetBinding(Label.TextProperty, "Name");

        grid.Children.Add(image);
        grid.Children.Add(nameLabel, 1, 0);
        return grid;
    })
});
```

The elements specified in the `DataTemplate` define the appearance of each item in the suggestions area. In this example, layout within the `DataTemplate` is managed by a `Grid`. The `Grid` contains an `Image` object, and a `Label` object, that both bind to properties of each `Monkey` object.

The following screenshots show the result of templating each item in the suggestions area:



For more information about data templates, see [Xamarin.Forms data templates](#).

Search box visibility

By default, when a `SearchHandler` is added at the top of a page, the search box is visible and fully expanded. However, this behavior can be changed by setting the `SearchHandler.SearchBoxVisibility` property to one of the `SearchBoxVisibility` enumeration members:

- `Hidden` – the search box is not visible or accessible.
- `Collapsible` – the search box is hidden until the user performs an action to reveal it. On iOS the search box is revealed by vertically bouncing the page content, and on Android the search box is revealed by tapping the question mark icon.
- `Expanded` – the search box is visible and fully expanded. This is the default value of the `SearchBoxVisibility` property.

IMPORTANT

On iOS, a collapsible search box requires iOS 11 or greater.

The following example shows how to hide the search box:

```
<ContentPage ...>
    xmlns:controls="clr-namespace:Xaminals.Controls"
    <Shell.SearchHandler>
        <controls:AnimalSearchHandler SearchBoxVisibility="Hidden">
            ...
        </controls:AnimalSearchHandler>
    ...
</ContentPage>
```

Search box focus

Tapping in a search box invokes the onscreen keyboard, with the search box gaining input focus. This can also be achieved programmatically by calling the `Focus` method, which attempts to set input focus on the search box, and returns `true` if successful. When a search box gains focus, the `Focused` event is fired and the overridable `OnFocused` method is called.

When a search box has input focus, tapping elsewhere on the screen dismisses the onscreen keyboard, and the search box loses input focus. This can also be achieved programmatically by calling the `Unfocus` method. When a search box loses focus, the `Unfocused` event is fired and the overridable `OnUnfocus` method is called.

The focus state of a search box can be retrieved through the `IsFocused` property, which returns `true` if a `SearchHandler` currently has input focus.

SearchHandler keyboard

The keyboard that's presented when users interact with a `SearchHandler` 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:

```
<SearchHandler Keyboard="Email" />
```

The equivalent C# code is:

```
SearchHandler searchHandler = new SearchHandler { Keyboard = Keyboard.Email };
```

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:

```
<SearchHandler Placeholder="Enter search terms">
    <SearchHandler.Keyboard>
        <Keyboard x:FactoryMethod="Create">
            <x:Arguments>
                <KeyboardFlags>Suggestions,CapitalizeCharacter</KeyboardFlags>
            </x:Arguments>
        </Keyboard>
    </SearchHandler.Keyboard>
</SearchHandler>
```

The equivalent C# code is:

```
SearchHandler searchHandler = new SearchHandler { Placeholder = "Enter search terms" };
searchHandler.Keyboard = Keyboard.Create(KeyboardFlags.Suggestions | KeyboardFlags.CapitalizeCharacter);
```

Related links

- [Xaminals \(sample\)](#)
- [Xamarin.Forms Shell navigation](#)

Xamarin.Forms Shell lifecycle

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

Shell applications respect the Xamarin.Forms lifecycle, and additionally fire an `Appearing` event when a page is about to appear on the screen, and a `Disappearing` event when a page is about to disappear from the screen. These events are propagated to pages, and can be handled by overriding the `OnAppearing` or `OnDisappearing` methods on the page.

NOTE

In a Shell application, the `Appearing` and `Disappearing` events are raised from cross-platform code, prior to platform code making a page visible, or removing a page from the screen.

For more information about the Xamarin.Forms app lifecycle, see [Xamarin.Forms app lifecycle](#).

Hierarchical navigation

In a Shell application, pushing a page onto the navigation stack will result in the currently visible `ShellContent` object, and its page content, raising the `Disappearing` event. Similarly, popping the last page from the navigation stack will result in the newly visible `ShellContent` object, and its page content, raising the `Appearing` event.

For more information about hierarchical navigation, see [Xamarin.Forms hierarchical navigation](#).

Modal navigation

In a Shell application, pushing a modal page onto the modal navigation stack will result in all visible Shell objects raising the `Disappearing` event. Similarly, popping the last modal page from the modal navigation stack will result in all visible Shell objects raising the `Appearing` event.

For more information about modal navigation, see [Xamarin.Forms modal pages](#).

Related links

- [Xaminals \(sample\)](#)
- [Xamarin.Forms app lifecycle](#)
- [Xamarin.Forms modal pages](#)

Xamarin.Forms Shell custom renderers

8/4/2022 • 3 minutes to read • [Edit Online](#)

One of the advantages of Xamarin.Forms Shell applications is that their appearance and behavior is highly customizable through the properties and methods that the various Shell classes expose. However, it's also possible to create a Shell custom renderer when more extensive platform-specific customizations are required. As with other custom renderers, a Shell custom renderer can be added to just one platform project to customize appearance and behavior, while allowing the default behavior on the other platform; or a different Shell custom renderer can be added to each platform project to customize appearance and behavior on both iOS and Android.

Shell applications are rendered using the `ShellRenderer` class on iOS and Android. On iOS, the `ShellRenderer` class can be found in the `Xamarin.Forms.Platform.iOS` namespace. On Android, the `ShellRenderer` class can be found in the `Xamarin.Forms.Platform.Android` namespace.

The process for creating a Shell custom renderer is as follows:

1. Subclass the `Shell` class. This will already be accomplished in your Shell application.
2. Consume the subclassed `Shell` class. This will already be accomplished in your Shell application.
3. Create a custom renderer class that derives from the `ShellRenderer` class, on the required platforms.

Create a custom renderer class

The process for creating a Shell custom renderer class is as follows:

1. Create a subclass of the `ShellRenderer` class.
2. Override the required methods to perform the required customization.
3. Add an `ExportRendererAttribute` to the `ShellRenderer` subclass, to specify that it will be used to render the Shell application. This attribute is used to register the custom renderer with Xamarin.Forms.

NOTE

It's optional to provide a Shell custom renderer in each platform project. If a custom renderer isn't registered, then the default `ShellRenderer` class will be used.

The `ShellRenderer` class exposes the following overridable methods:

IOS	ANDROID	UWP
SetElementSize CreateFlyoutRenderer CreateNavBarAppearanceTracker CreatePageRendererTracker CreateShellFlyoutContentRenderer CreateShellItemRenderer CreateShellItemTransition CreateShellSearchResultsRenderer CreateShellSectionRenderer CreateTabBarAppearanceTracker Dispose OnCurrentItemChanged OnElementPropertyChanged OnElementSet UpdateBackgroundColor	CreateFragmentForPage CreateShellFlyoutContentRenderer CreateShellFlyoutRenderer CreateShellItemRenderer CreateShellSectionRenderer CreateTrackerForToolbar CreateToolbarAppearanceTracker CreateTabLayoutAppearanceTracker CreateBottomNavViewAppearanceTracker OnElementPropertyChanged OnElementSet SwitchFragment Dispose	CreateShellFlyoutTemplateSelector CreateShellHeaderRenderer CreateShellItemRenderer CreateShellSectionRenderer OnElementPropertyChanged OnElementSet UpdateFlyoutBackdropColor UpdateFlyoutBackgroundColor

The `FlyoutItem` and `TabBar` classes are aliases for the `ShellItem` class, and the `Tab` class is an alias for the `ShellSection` class. Therefore, the `CreateShellItemRenderer` method should be overridden when creating a custom renderer for `FlyoutItem` objects, and the `CreateShellSectionRenderer` method should be overridden when creating a custom renderer for `Tab` objects.

IMPORTANT

There are additional Shell renderer classes, such as `ShellSectionRenderer` and `ShellItemRenderer`, on iOS, Android, and UWP. However, these additional renderer classes are created by overrides in the `ShellRenderer` class. Therefore, customizing the behavior of these additional renderer classes can be achieved by subclassing them, and creating an instance of the subclass in the appropriate override in the subclassed `ShellRenderer` class.

iOS example

The following code example shows a subclassed `ShellRenderer`, for iOS, that sets a background image on the navigation bar of the Shell application:

```

using UIKit;
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer(typeof(Xaminals.AppShell), typeof(Xaminals.iOS.MyShellRenderer))]
namespace Xaminals.iOS
{
    public class MyShellRenderer : ShellRenderer
    {
        protected override IShellSectionRenderer CreateShellSectionRenderer(ShellSection shellSection)
        {
            var renderer = base.CreateShellSectionRenderer(shellSection);
            if (renderer != null)
            {
                (renderer as
ShellSectionRenderer).NavigationBar.SetBackgroundImage(UIImage.FromFile("monkey.png"),
UIBarMetrics.Default);
            }
            return renderer;
        }
    }
}

```

The `MyShellRenderer` class overrides the `CreateShellSectionRenderer` method, and retrieves the renderer created by the base class. It then modifies the renderer by setting a background image on the navigation bar, before returning the renderer.

Android example

The following code example shows a subclassed `ShellRenderer`, for Android, that sets a background image on the navigation bar of the Shell application:

```
using Android.Content;
using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

[assembly: ExportRenderer(typeof(Xaminals.AppShell), typeof(Xaminals.Droid.MyShellRenderer))]
namespace Xaminals.Droid
{
    public class MyShellRenderer : ShellRenderer
    {
        public MyShellRenderer(Context context) : base(context)
        {

        }

        protected override IShellToolbarAppearanceTracker CreateToolbarAppearanceTracker()
        {
            return new MyShellToolbarAppearanceTracker(this);
        }
    }
}
```

The `MyShellRenderer` class overrides the `CreateToolbarAppearanceTracker` method, and returns an instance of the `MyShellToolbarAppearanceTracker` class. The `MyShellToolbarAppearanceTracker` class, which derives from the `ShellToolbarAppearanceTracker` class, is shown in the following example:

```
using AndroidX.AppCompat.Widget;
using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

namespace Xaminals.Droid
{
    public class MyShellToolbarAppearanceTracker : ShellToolbarAppearanceTracker
    {
        public MyShellToolbarAppearanceTracker(IShellContext context) : base(context)
        {

        }

        public override void SetAppearance(Toolbar toolbar, IShellToolbarTracker toolbarTracker,
ShellAppearance appearance)
        {
            base.SetAppearance(toolbar, toolbarTracker, appearance);
            toolbar.SetBackgroundResource(Resource.Drawable.monkey);
        }
    }
}
```

The `MyShellToolbarAppearanceTracker` class overrides the `SetAppearance` method, and modifies the toolbar by setting a background image on it.

IMPORTANT

It's only necessary to add the `ExportRendererAttribute` to a custom renderer that derives from the `ShellRenderer` class. Additional subclassed Shell renderer classes are created by the subclassed `ShellRenderer` class.

Related links

- [Xamarin.Forms Custom Renderers](#)

Xamarin.Forms templates

8/4/2022 • 2 minutes to read • [Edit Online](#)

Control templates

Xamarin.Forms control templates define the visual structure of [ContentView](#) derived custom controls, and [ContentPage](#) derived pages.

Data templates

Xamarin.Forms data templates define the presentation of data on supported controls.

Xamarin.Forms control templates

8/4/2022 • 13 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms control templates enable you to define the visual structure of `ContentView` derived custom controls, and `ContentPage` derived pages. Control templates separate the user interface (UI) for a custom control, or page, from the logic that implements the control or page. Additional content can also be inserted into the templated custom control, or templated page, at a pre-defined location.

For example, a control template can be created that redefines the UI provided by a custom control. The control template can then be consumed by the required custom control instance. Alternatively, a control template can be created that defines any common UI that will be used by multiple pages in an application. The control template can then be consumed by multiple pages, with each page still displaying its unique content.

Create a ControlTemplate

The following example shows the code for a `CardView` custom control:

```
public class CardView : ContentView
{
    public static readonly BindableProperty CardTitleProperty = BindableProperty.Create(nameof(CardTitle),
        typeof(string), typeof(CardView), string.Empty);
    public static readonly BindableProperty CardDescriptionProperty =
        BindableProperty.Create(nameof(CardDescription), typeof(string), typeof(CardView), string.Empty);
    // ...

    public string CardTitle
    {
        get => (string)GetValue(CardTitleProperty);
        set => SetValue(CardTitleProperty, value);
    }

    public string CardDescription
    {
        get => (string)GetValue(CardDescriptionProperty);
        set => SetValue(CardDescriptionProperty, value);
    }
    // ...
}
```

The `CardView` class, which derives from the `ContentView` class, represents a custom control that displays data in a card-like layout. The class contains properties, which are backed by bindable properties, for the data it displays. However, the `CardView` class does not define any UI. Instead, the UI will be defined with a control template. For more information about creating `ContentView` derived custom controls, see [Xamarin.Forms ContentView](#).

A control template is created with the `ControlTemplate` type. When you create a `ControlTemplate`, you combine `View` objects to build the UI for a custom control, or page. A `ControlTemplate` must have only one `View` as its root element. However, the root element usually contains other `View` objects. The combination of objects makes up the control's visual structure.

While a `ControlTemplate` can be defined inline, the typical approach to declaring a `ControlTemplate` is as a resource in a resource dictionary. Because control templates are resources, they obey the same scoping rules

that apply to all resources. For example, if you declare a control template in the root element of your application definition XAML file, the template can be used anywhere in your application. If you define the template in a page, only that page can use the control template. For more information about resources, see [Xamarin.Forms Resource Dictionaries](#).

The following XAML example shows a `ControlTemplate` for `CardView` objects:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    ...
<ContentPage.Resources>
    <ControlTemplate x:Key="CardViewControlTemplate">
        <Frame BindingContext="{Binding Source={RelativeSource TemplatedParent}}"
            BackgroundColor="{Binding CardColor}"
            BorderColor="{Binding BorderColor}"
            CornerRadius="5"
            HasShadow="True"
            Padding="8"
            HorizontalOptions="Center"
            VerticalOptions="Center">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="75" />
                    <RowDefinition Height="4" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="75" />
                    <ColumnDefinition Width="200" />
                </Grid.ColumnDefinitions>
                <Frame IsClippedToBounds="True"
                    BorderColor="{Binding BorderColor}"
                    BackgroundColor="{Binding IconBackgroundColor}"
                    CornerRadius="38"
                    HeightRequest="60"
                    WidthRequest="60"
                    HorizontalOptions="Center"
                    VerticalOptions="Center">
                    <Image Source="{Binding IconImageSource}"
                        Margin="-20"
                        WidthRequest="100"
                        HeightRequest="100"
                        Aspect="AspectFill" />
                </Frame>
                <Label Grid.Column="1"
                    Text="{Binding CardTitle}"
                    FontAttributes="Bold"
                    FontSize="Large"
                    VerticalTextAlignment="Center"
                    HorizontalTextAlignment="Start" />
                <BoxView Grid.Row="1"
                    Grid.ColumnSpan="2"
                    BackgroundColor="{Binding BorderColor}"
                    HeightRequest="2"
                    HorizontalOptions="Fill" />
                <Label Grid.Row="2"
                    Grid.ColumnSpan="2"
                    Text="{Binding CardDescription}"
                    VerticalTextAlignment="Start"
                    VerticalOptions="Fill"
                    HorizontalOptions="Fill" />
            </Grid>
        </Frame>
    </ControlTemplate>
</ContentPage.Resources>
...
</ContentPage>

```

When a `ControlTemplate` is declared as a resource, it must have a key specified with the `x:Key` attribute so that it can be identified in the resource dictionary. In this example, the root element of the `cardViewControlTemplate` is a `Frame` object. The `Frame` object uses the `RelativeSource` markup extension to set its `BindingContext` to the runtime object instance to which the template will be applied, which is known as the *templated parent*. The

`Frame` object uses a combination of `Grid`, `Frame`, `Image`, `Label`, and `BoxView` objects to define the visual structure of a `CardView` object. The binding expressions of these objects resolve against `CardView` properties, due to inheriting the `BindingContext` from the root `Frame` element. For more information about the `RelativeSource` markup extension, see [Xamarin.Forms Relative Bindings](#).

Consume a ControlTemplate

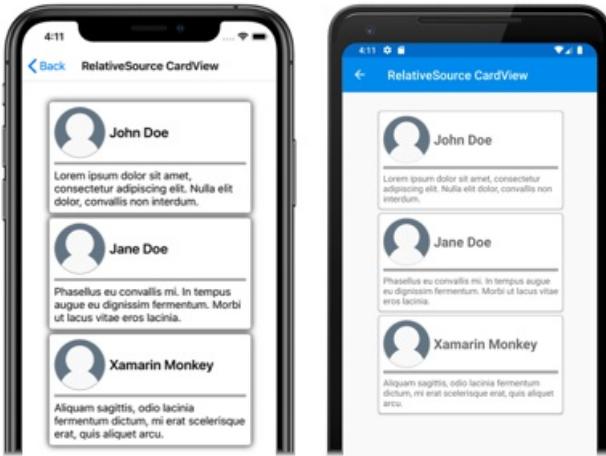
A `ControlTemplate` can be applied to a `ContentView` derived custom control by setting its `ControlTemplate` property to the control template object. Similarly, a `ControlTemplate` can be applied to a `ContentPage` derived page by setting its `ControlTemplate` property to the control template object. At runtime, when a `ControlTemplate` is applied, all of the controls that are defined in the `ControlTemplate` are added to the visual tree of the templated custom control, or templated page.

The following example shows the `CardViewControlTemplate` being assigned to the `ControlTemplate` property of each `CardView` object:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="clr-namespace:ControlTemplateDemos.Controls"
    ...>
<StackLayout Margin="30">
    <controls:CardView BorderColor="DarkGray"
        CardTitle="John Doe"
        CardDescription="Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla
        elit dolor, convallis non interdum."
        IconBackgroundColor="SlateGray"
        IconImageSource="user.png"
        ControlTemplate="{StaticResource CardViewControlTemplate}" />
    <controls:CardView BorderColor="DarkGray"
        CardTitle="Jane Doe"
        CardDescription="Phasellus eu convallis mi. In tempus augue eu dignissim
        fermentum. Morbi ut lacus vitae eros lacinia."
        IconBackgroundColor="SlateGray"
        IconImageSource="user.png"
        ControlTemplate="{StaticResource CardViewControlTemplate}" />
    <controls:CardView BorderColor="DarkGray"
        CardTitle="Xamarin Monkey"
        CardDescription="Aliquam sagittis, odio lacinia fermentum dictum, mi erat
        scelerisque erat, quis aliquet arcu."
        IconBackgroundColor="SlateGray"
        IconImageSource="user.png"
        ControlTemplate="{StaticResource CardViewControlTemplate}" />
</StackLayout>
</ContentPage>
```

In this example, the controls in the `CardViewControlTemplate` become part of the visual tree for each `CardView` object. Because the root `Frame` object for the control template sets its `BindingContext` to the templated parent, the `Frame` and its children resolve their binding expressions against the properties of each `CardView` object.

The following screenshots show the `CardViewControlTemplate` applied to the three `CardView` objects:



IMPORTANT

The point in time that a `ControlTemplate` is applied to a control instance can be detected by overriding the `OnApplyTemplate` method in the templated custom control, or templated page. For more information, see [Get a named element from a template](#).

Pass parameters with TemplateBinding

The `TemplateBinding` markup extension binds a property of an element that is in a `ControlTemplate` to a public property that is defined by the templated custom control or templated page. When you use a `TemplateBinding`, you enable properties on the control to act as parameters to the template. Therefore, when a property on a templated custom control or templated page is set, that value is passed onto the element that has the `TemplateBinding` on it.

IMPORTANT

The `TemplateBinding` markup expression enables the `RelativeSource` binding from the previous control template to be removed, and replaces the `Binding` expressions.

The `TemplateBinding` markup extension defines the following properties:

- `Path`, of type `string`, the path to the property.
- `Mode`, of type `BindingMode`, the direction in which changes propagate between the *source* and *target*.
- `Converter`, of type `IValueConverter`, the binding value converter.
- `ConverterParameter`, of type `object`, the parameter to the binding value converter.
- `StringFormat`, of type `string`, the string format for the binding.

The `ContentProperty` for the `TemplateBinding` markup extension is `Path`. Therefore, the "Path=" part of the markup extension can be omitted if the path is the first item in the `TemplateBinding` expression. For more information about using these properties in a binding expression, see [Xamarin.Forms Data Binding](#).

WARNING

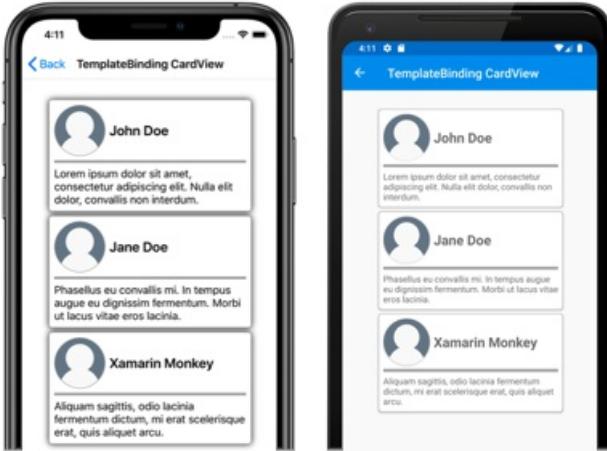
The `TemplateBinding` markup extension should only be used within a `ControlTemplate`. However, attempting to use a `TemplateBinding` expression outside of a `ControlTemplate` will not result in a build error or an exception being thrown.

The following XAML example shows a `ControlTemplate` for `CardView` objects, that uses the `TemplateBinding`

markup extension:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    ...
    <ContentPage.Resources>
        <ControlTemplate x:Key="CardViewControlTemplate">
            <Frame BackgroundColor="{TemplateBinding CardColor}"
                BorderColor="{TemplateBinding BorderColor}"
                CornerRadius="5"
                HasShadow="True"
                Padding="8"
                HorizontalOptions="Center"
                VerticalOptions="Center">
                <Grid>
                    <Grid.RowDefinitions>
                        <RowDefinition Height="75" />
                        <RowDefinition Height="4" />
                        <RowDefinition Height="Auto" />
                    </Grid.RowDefinitions>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="75" />
                        <ColumnDefinition Width="200" />
                    </Grid.ColumnDefinitions>
                    <Frame IsClippedToBounds="True"
                        BorderColor="{TemplateBinding BorderColor}"
                        BackgroundColor="{TemplateBinding IconBackgroundColor}"
                        CornerRadius="38"
                        HeightRequest="60"
                        WidthRequest="60"
                        HorizontalOptions="Center"
                        VerticalOptions="Center">
                        <Image Source="{TemplateBinding IconImageSource}"
                            Margin="-20"
                            WidthRequest="100"
                            HeightRequest="100"
                            Aspect="AspectFill" />
                    </Frame>
                    <Label Grid.Column="1"
                        Text="{TemplateBinding CardTitle}"
                        FontAttributes="Bold"
                        FontSize="Large"
                        VerticalTextAlignment="Center"
                        HorizontalTextAlignment="Start" />
                    <BoxView Grid.Row="1"
                        Grid.ColumnSpan="2"
                        BackgroundColor="{TemplateBinding BorderColor}"
                        HeightRequest="2"
                        HorizontalOptions="Fill" />
                    <Label Grid.Row="2"
                        Grid.ColumnSpan="2"
                        Text="{TemplateBinding CardDescription}"
                        VerticalTextAlignment="Start"
                        VerticalOptions="Fill"
                        HorizontalOptions="Fill" />
                </Grid>
            </Frame>
        </ControlTemplate>
    </ContentPage.Resources>
    ...
</ContentPage>
```

In this example, the `TemplateBinding` markup extension resolves binding expressions against the properties of each `CardView` object. The following screenshots show the `CardViewControlTemplate` applied to the three `CardView` objects:



IMPORTANT

Using the `TemplateBinding` markup extension is equivalent to setting the `BindingContext` of the root element in the template to its templated parent with the `RelativeSource` markup extension, and then resolving bindings of child objects with the `Binding` markup extension. In fact, the `TemplateBinding` markup extension creates a `Binding` whose `Source` is `RelativeBindingSource.TemplatedParent`.

Apply a ControlTemplate with a style

Control templates can also be applied with styles. This is achieved by creating an *implicit* or *explicit* style that consumes the `ControlTemplate`.

The following XAML example shows an *implicit* style that consumes the `CardViewControlTemplate`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="clr-namespace:ControlTemplateDemos.Controls"
    ...>
<ContentPage.Resources>
    <ControlTemplate x:Key="CardViewControlTemplate">
        ...
    </ControlTemplate>

    <Style TargetType="controls:CardView">
        <Setter Property="ControlTemplate"
            Value="{StaticResource CardViewControlTemplate}" />
    </Style>
</ContentPage.Resources>
<StackLayout Margin="30">
    <controls:CardView BorderColor="DarkGray"
        CardTitle="John Doe"
        CardDescription="Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla
        elit dolor, convallis non interdum."
        IconBackgroundColor="SlateGray"
        IconImageSource="user.png" />
    <controls:CardView BorderColor="DarkGray"
        CardTitle="Jane Doe"
        CardDescription="Phasellus eu convallis mi. In tempus augue eu dignissim
        fermentum. Morbi ut lacus vitae eros lacinia."
        IconBackgroundColor="SlateGray"
        IconImageSource="user.png"/>
    <controls:CardView BorderColor="DarkGray"
        CardTitle="Xamarin Monkey"
        CardDescription="Aliquam sagittis, odio lacinia fermentum dictum, mi erat
        scelerisque erat, quis aliquet arcu."
        IconBackgroundColor="SlateGray"
        IconImageSource="user.png" />
</StackLayout>
</ContentPage>

```

In this example, the *implicit* `Style` is automatically applied to each `CardView` object, and sets the `ControlTemplate` property of each `cardView` to `CardViewControlTemplate`.

For more information about styles, see [Xamarin.Forms Styles](#).

Redefine a control's UI

When a `ControlTemplate` is instantiated and assigned to the `ControlTemplate` property of a `ContentView` derived custom control, or a `ContentPage` derived page, the visual structure defined for the custom control or page is replaced with the visual structure defined in the `ControlTemplate`.

For example, the `CardViewUI` custom control defines its user interface using the following XAML:

```

<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ControlTemplateDemos.Controls.CardViewUI"
    x:Name="this">
    <Frame BindingContext="{x:Reference this}"
        BackgroundColor="{Binding CardColor}"
        BorderColor="{Binding BorderColor}"
        CornerRadius="5"
        HasShadow="True"
        Padding="8"
        HorizontalOptions="Center"
        VerticalOptions="Center">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="75" />
                <RowDefinition Height="4" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="75" />
                <ColumnDefinition Width="200" />
            </Grid.ColumnDefinitions>
            <Frame IsClippedToBounds="True"
                BorderColor="{Binding BorderColor, FallbackValue='Black'}"
                BackgroundColor="{Binding IconBackgroundColor, FallbackValue='Gray'}"
                CornerRadius="38"
                HeightRequest="60"
                WidthRequest="60"
                HorizontalOptions="Center"
                VerticalOptions="Center">
                <Image Source="{Binding IconImageSource}"
                    Margin="-20"
                    WidthRequest="100"
                    HeightRequest="100"
                    Aspect="AspectFill" />
            </Frame>
            <Label Grid.Column="1"
                Text="{Binding CardTitle, FallbackValue='Card title'}"
                FontAttributes="Bold"
                FontSize="Large"
                VerticalTextAlignment="Center"
                HorizontalTextAlignment="Start" />
            <BoxView Grid.Row="1"
                Grid.ColumnSpan="2"
                BackgroundColor="{Binding BorderColor, FallbackValue='Black'}"
                HeightRequest="2"
                HorizontalOptions="Fill" />
            <Label Grid.Row="2"
                Grid.ColumnSpan="2"
                Text="{Binding CardDescription, FallbackValue='Card description'}"
                VerticalTextAlignment="Start"
                VerticalOptions="Fill"
                HorizontalOptions="Fill" />
        </Grid>
    </Frame>
</ContentView>

```

However, the controls that comprise this UI can be replaced by defining a new visual structure in a `ControlTemplate`, and assigning it to the `ControlTemplate` property of a `CardViewUI` object:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    ...
    </ContentPage.Resources>
        <ControlTemplate x:Key="CardViewCompressed">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="100" />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="100" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>
                <Image Source="{TemplateBinding IconImageSource}"
                    BackgroundColor="{TemplateBinding IconBackgroundColor}"
                    WidthRequest="100"
                    HeightRequest="100"
                    Aspect="AspectFill"
                    HorizontalOptions="Center"
                    VerticalOptions="Center" />
                <StackLayout Grid.Column="1">
                    <Label Text="{TemplateBinding CardTitle}"
                        FontAttributes="Bold" />
                    <Label Text="{TemplateBinding CardDescription}" />
                </StackLayout>
            </Grid>
        </ControlTemplate>
    </ContentPage.Resources>
    <StackLayout Margin="30">
        <controls:CardViewUI BorderColor="DarkGray"
            CardTitle="John Doe"
            CardDescription="Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla
            elit dolor, convallis non interdum."
            IconBackgroundColor="SlateGray"
            IconImageSource="user.png"
            ControlTemplate="{StaticResource CardViewCompressed}" />
        <controls:CardViewUI BorderColor="DarkGray"
            CardTitle="Jane Doe"
            CardDescription="Phasellus eu convallis mi. In tempus augue eu dignissim
            fermentum. Morbi ut lacus vitae eros lacinia."
            IconBackgroundColor="SlateGray"
            IconImageSource="user.png"
            ControlTemplate="{StaticResource CardViewCompressed}" />
        <controls:CardViewUI BorderColor="DarkGray"
            CardTitle="Xamarin Monkey"
            CardDescription="Aliquam sagittis, odio lacinia fermentum dictum, mi erat
            scelerisque erat, quis aliquet arcu."
            IconBackgroundColor="SlateGray"
            IconImageSource="user.png"
            ControlTemplate="{StaticResource CardViewCompressed}" />
    </StackLayout>
</ContentPage>

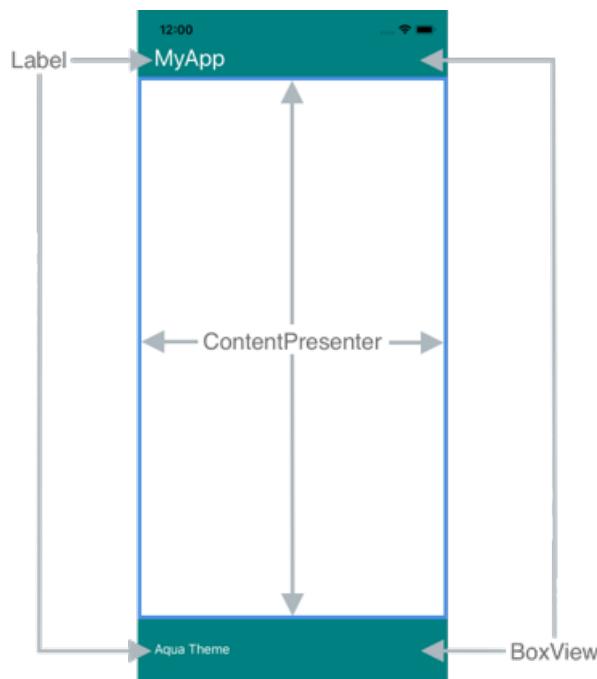
```

In this example, the visual structure of the `CardViewUI` object is redefined in a `ControlTemplate` that provides a more compact visual structure that's suitable for a condensed list:



Substitute content into a ContentPresenter

A `ContentPresenter` can be placed in a control template to mark where content to be displayed by the templated custom control or templated page will appear. The custom control or page that consumes the control template 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:



The following XAML shows a control template named `TealTemplate` that contains a `ContentPresenter` in its visual structure:

```

<ControlTemplate x:Key="TealTemplate">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="0.1*" />
            <RowDefinition Height="0.8*" />
            <RowDefinition Height="0.1*" />
        </Grid.RowDefinitions>
        <BoxView Color="Teal" />
        <Label Margin="20,0,0,0"
            Text="{TemplateBinding HeaderText}"
            TextColor="White"
            FontSize="Title"
            VerticalOptions="Center" />
        <ContentPresenter Grid.Row="1" />
        <BoxView Grid.Row="2"
            Color="Teal" />
        <Label x:Name="changeThemeLabel"
            Grid.Row="2"
            Margin="20,0,0,0"
            Text="Change Theme"
            TextColor="White"
            HorizontalOptions="Start"
            VerticalOptions="Center">
            <Label.GestureRecognizers>
                <TapGestureRecognizer Tapped="OnChangeThemeLabelTapped" />
            </Label.GestureRecognizers>
        </Label>
        <controls:HyperlinkLabel Grid.Row="2"
            Margin="0,0,20,0"
            Text="Help"
            TextColor="White"
            Url="https://docs.microsoft.com/xamarin/xamarin-forms/"
            HorizontalOptions="End"
            VerticalOptions="Center" />
    </Grid>
</ControlTemplate>

```

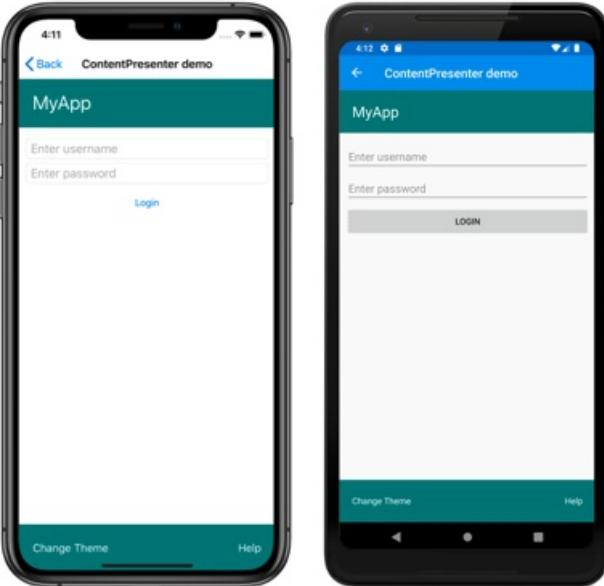
The following example shows `TealTemplate` assigned to the `ControlTemplate` property of a `ContentPage` derived page:

```

<controls:HeaderFooterPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="clr-namespace:ControlTemplateDemos.Controls"
    ControlTemplate="{StaticResource TealTemplate}"
    HeaderText="MyApp"
    ...>
    <StackLayout Margin="10">
        <Entry Placeholder="Enter username" />
        <Entry Placeholder="Enter password"
            IsPassword="True" />
        <Button Text="Login" />
    </StackLayout>
</controls:HeaderFooterPage>

```

At runtime, when `TealTemplate` is applied to the page, the page content is substituted into the `ContentPresenter` defined in the control template:



Get a named element from a template

Named elements within a control template can be retrieved from the templated custom control or templated page. This can be achieved with the `GetTemplateChild` method, which returns the named element in the instantiated `ControlTemplate` visual tree, if found. Otherwise, it returns `null`.

After a control template has been instantiated, the template's `OnApplyTemplate` method is called. The `GetTemplateChild` method should therefore be called from a `OnApplyTemplate` override in the templated control or templated page.

IMPORTANT

The `GetTemplateChild` method should only be called after the `OnApplyTemplate` method has been called.

The following XAML shows a control template named `TealTemplate` that can be applied to `ContentPage` derived pages:

```
<ControlTemplate x:Key="TealTemplate">
    <Grid>
        ...
        <Label x:Name="changeThemeLabel"
            Grid.Row="2"
            Margin="20,0,0,0"
            Text="Change Theme"
            TextColor="White"
            HorizontalOptions="Start"
            VerticalOptions="Center">
            <Label.GestureRecognizers>
                <TapGestureRecognizer Tapped="OnChangeThemeLabelTapped" />
            </Label.GestureRecognizers>
        </Label>
        ...
    </Grid>
</ControlTemplate>
```

In this example, the `Label` element is named, and can be retrieved in the code for the templated page. This is achieved by calling the `GetTemplateChild` method from the `OnApplyTemplate` override for the templated page:

```

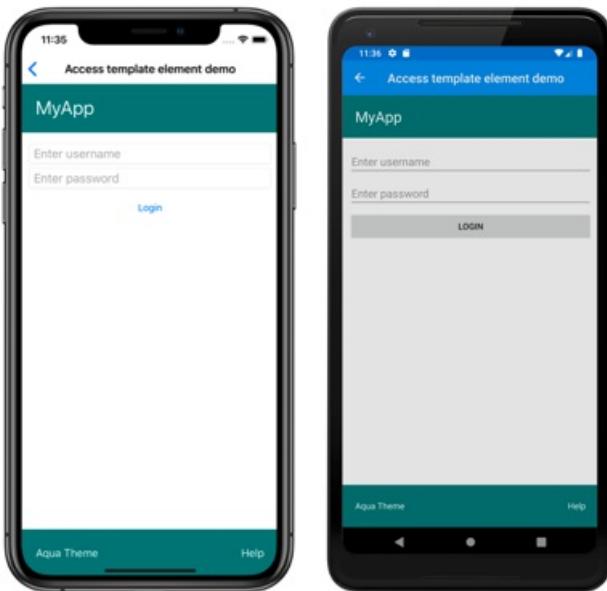
public partial class AccessTemplateElementPage : HeaderFooterPage
{
    Label themeLabel;

    public AccessTemplateElementPage()
    {
        InitializeComponent();
    }

    protected override void OnApplyTemplate()
    {
        base.OnApplyTemplate();
        themeLabel = (Label)GetTemplateChild("changeThemeLabel");
        themeLabel.Text = OriginalTemplate ? "Aqua Theme" : "Teal Theme";
    }
}

```

In this example, the `Label` object named `changeThemeLabel` is retrieved once the `ControlTemplate` has been instantiated. `changeThemeLabel` can then be accessed and manipulated by the `AccessTemplateElementPage` class. The following screenshots show that the text displayed by the `Label` has been changed:



Bind to a viewmodel

A `ControlTemplate` can data bind to a.viewmodel, even when the `ControlTemplate` binds to the templated parent (the runtime object instance to which the template is applied).

The following XAML example shows a page that consumes a.viewmodel named `PeopleViewModel`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:ControlTemplateDemos"
    xmlns:controls="clr-namespace:ControlTemplateDemos.Controls"
    ...>
<ContentPage.BindingContext>
    <local:PeopleViewModel />
</ContentPage.BindingContext>

<ContentPage.Resources>
    <DataTemplate x:Key="PersonTemplate">
        <controls:CardView BorderColor="DarkGray"
            CardTitle="{Binding Name}"
            CardDescription="{Binding Description}"
            ControlTemplate="{StaticResource CardViewControlTemplate}" />
    </DataTemplate>
</ContentPage.Resources>

<StackLayout Margin="10"
    BindableLayout.ItemsSource="{Binding People}"
    BindableLayout.ItemTemplate="{StaticResource PersonTemplate}" />
</ContentPage>

```

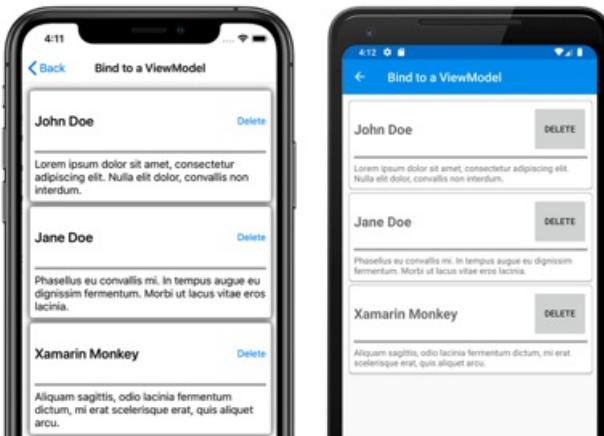
In this example, the `BindingContext` of the page is set to a `PeopleViewModel` instance. This viewmodel exposes a `People` collection and an `ICommand` named `DeletePersonCommand`. The `StackLayout` on the page uses a bindable layout to data bind to the `People` collection, and the `ItemTemplate` of the bindable layout is set to the `PersonTemplate` resource. This `DataTemplate` specifies that each item in the `People` collection will be displayed using a `CardView` object. The visual structure of the `CardView` object is defined using a `ControlTemplate` named `CardViewControlTemplate`:

```

<ControlTemplate x:Key="CardViewControlTemplate">
    <Frame BindingContext="{Binding Source={RelativeSource TemplatedParent}}"
        BackgroundColor="{Binding CardColor}"
        BorderColor="{Binding BorderColor}"
        CornerRadius="5"
        HasShadow="True"
        Padding="8"
        HorizontalOptions="Center"
        VerticalOptions="Center">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="75" />
                <RowDefinition Height="4" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Label Text="{Binding CardTitle}"
                FontAttributes="Bold"
                FontSize="Large"
                VerticalTextAlignment="Center"
                HorizontalTextAlignment="Start" />
            <BoxView Grid.Row="1"
                BackgroundColor="{Binding BorderColor}"
                HeightRequest="2"
                HorizontalOptions="Fill" />
            <Label Grid.Row="2"
                Text="{Binding CardDescription}"
                VerticalTextAlignment="Start"
                VerticalOptions="Fill"
                HorizontalOptions="Fill" />
            <Button Text="Delete"
                Command="{Binding Source={RelativeSource AncestorType={x:Type local:PeopleViewModel}}, Path=DeletePersonCommand}"
                CommandParameter="{Binding CardTitle}"
                HorizontalOptions="End" />
        </Grid>
    </Frame>
</ControlTemplate>

```

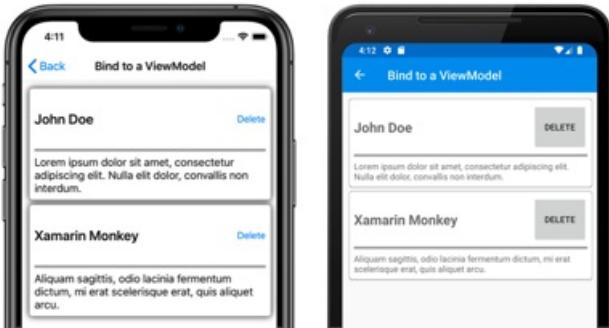
In this example, the root element of the `ControlTemplate` is a `Frame` object. The `Frame` object uses the `RelativeSource` markup extension to set its `BindingContext` to the templated parent. The binding expressions of the `Frame` object and its children resolve against `CardView` properties, due to inheriting the `BindingContext` from the root `Frame` element. The following screenshots show the page displaying the `People` collection, which consists of three items:



While the objects in the `ControlTemplate` bind to properties on its templated parent, the `Button` within the control template binds to both its templated parent, and to the `DeletePersonCommand` in the viewmodel. This is because the `Button.Command` property redefines its binding source to be the binding context of the ancestor

whose binding context type is `PeopleViewModel`, which is the `StackLayout`. The `Path` part of the binding expressions can then resolve the `DeletePersonCommand` property. However, the `Button.CommandParameter` property doesn't alter its binding source, instead inheriting it from its parent in the `ControlTemplate`. Therefore, the `CommandParameter` property binds to the `CardTitle` property of the `CardView`.

The overall effect of the `Button` bindings is that when the `Button` is tapped, the `DeletePersonCommand` in the `PeopleViewModel` class is executed, with the value of the `CardName` property being passed to the `DeletePersonCommand`. This results in the specified `CardView` being removed from the bindable layout:



For more information about relative bindings, see [Xamarin.Forms Relative Bindings](#).

Related links

- [ControlTemplateDemos \(sample\)](#)
- [Xamarin.Forms ContentView](#)
- [Xamarin.Forms Relative Bindings](#)
- [Xamarin.Forms Resource Dictionaries](#)
- [Xamarin.Forms Data Binding](#)
- [Xamarin.Forms Styles](#)

Xamarin.Forms Data Templates

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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:

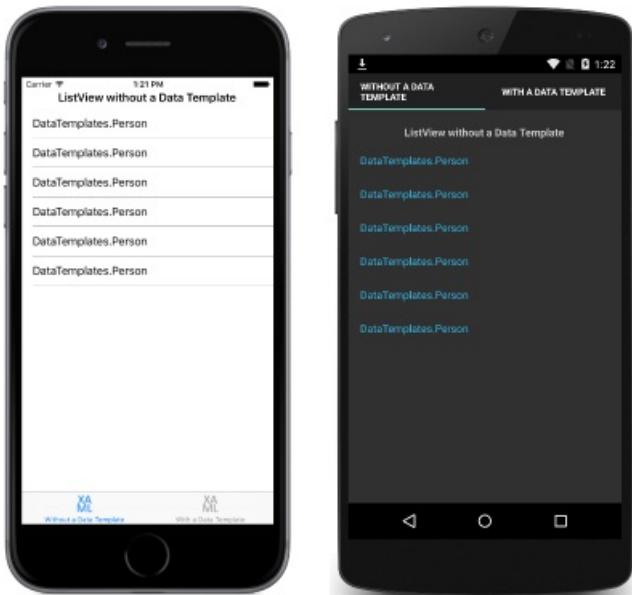
```

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:



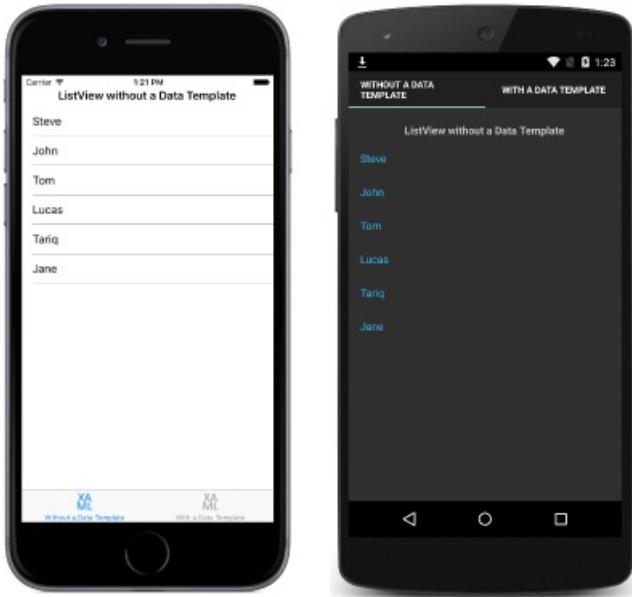
The `Person` object can override the `ToString` method to display meaningful data, as shown in the following code example:

```

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` 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

8/4/2022 • 5 minutes to read • [Edit Online](#)

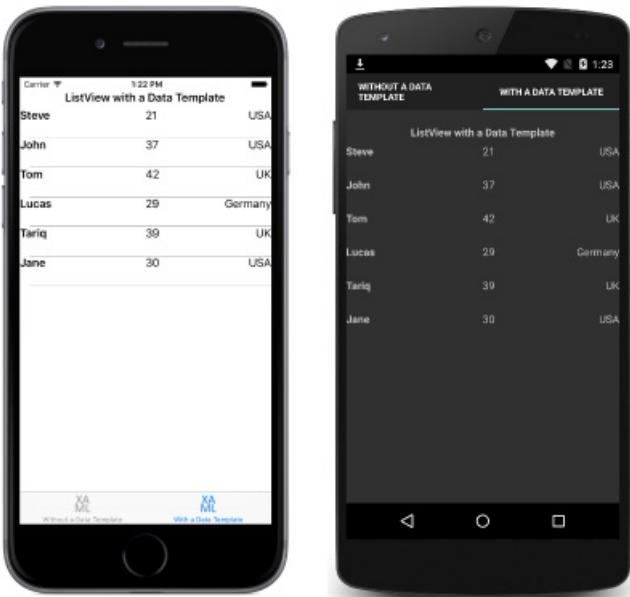
 [Download the sample](#)

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 `Cell`. This example uses a `viewCell`, which derives from `cell`. 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

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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>

```

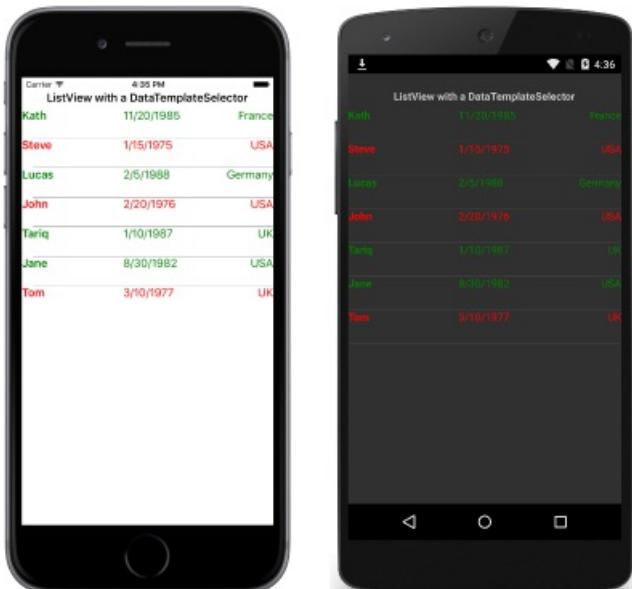
This page level `ResourceDictionary` defines two `DataTemplate` instances and a `PersonDataTemplateSelector` instance. The `PersonDataTemplateSelector` instance sets its `ValidTemplate` and `InvalidTemplate` properties to the appropriate `DataTemplate` instances by using the `StaticResource` markup extension. 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

8/4/2022 • 13 minutes to read • [Edit Online](#)

 [Download the sample](#)

Triggers allow you to express actions declaratively in XAML that change the appearance of controls based on events or property changes. In addition, state triggers, which are a specialized group of triggers, define when a `VisualState` should be applied.

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.

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" />
            <!-- multiple Setters elements are allowed -->
        </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 (i.e. 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" />
                    <!-- multiple Setters elements are allowed -->
                </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" />
            <!-- multiple Setters elements are allowed -->
        </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. For an example of this, see the `VisualElementPopTriggerAction` class in the accompanying sample application.

```
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 event trigger can then be consumed from XAML:

```
<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`.

```

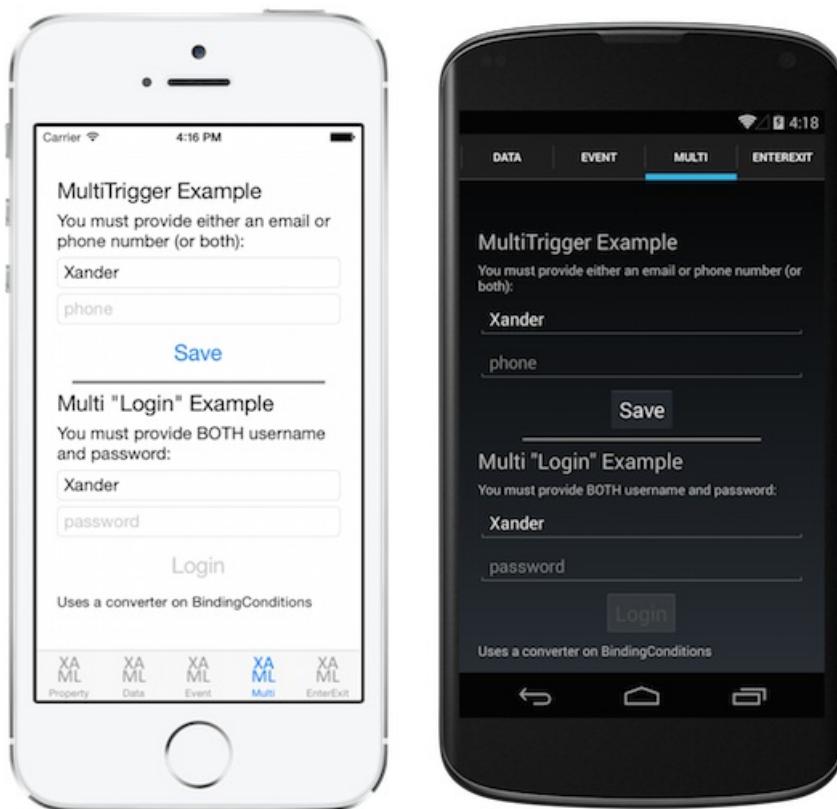
<Entry x:Name="user" Text="" Placeholder="user name" />

<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.

The `EnterActions` collection is used to define an `IList` of `TriggerAction` objects that will be invoked when the trigger condition is met. The `ExitActions` collection is used to define an `IList` of `TriggerAction` objects that

will be invoked after the trigger condition is no longer met.

NOTE

The `TriggerAction` objects defined in the `EnterActions` and `ExitActions` collections are ignored by the `EventTrigger` class.

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 int StartsFrom { set; get; }

    protected override void Invoke(VisualElement sender)
    {
        sender.Animate("FadeTriggerAction", new Animation((d) =>
        {
            var val = StartsFrom == 1 ? d : 1 - d;
            // so i was aiming for a different color, but then i liked the pink :(
            sender.BackgroundColor = Color.FromRgb(1, val, 1);
        }),
        length: 1000, // milliseconds
        easing: Easing.Linear);
    }
}
```

State triggers

State triggers are a specialized group of triggers that define the conditions under which a `VisualState` should be applied.

State triggers are added to the `StateTriggers` collection of a `VisualState`. This collection can contain a single state trigger, or multiple state triggers. A `VisualState` will be applied when any state triggers in the collection are active.

When using state triggers to control visual states, Xamarin.Forms uses the following precedence rules to determine which trigger (and corresponding `VisualState`) will be active:

1. Any trigger that derives from `StateTriggerBase`.
2. An `AdaptiveTrigger` activated due to the `MinWindowWidth` condition being met.
3. An `AdaptiveTrigger` activated due to the `MinWindowHeight` condition being met.

If multiple triggers are simultaneously active (for example, two custom triggers) then the first trigger declared in the markup takes precedence.

NOTE

State triggers can be set in a `Style`, or directly on elements.

For more information about visual states, see [Xamarin.Forms Visual State Manager](#).

State trigger

The `StateTrigger` class, which derives from the `StateTriggerBase` class, has an `IsActive` bindable property. A `StateTrigger` triggers a `VisualState` change when the `IsActive` property changes value.

The `StateTriggerBase` class, which is the base class for all state triggers, has an `IsActive` property and an `IsActiveChanged` event. This event fires whenever a `VisualState` change occurs. In addition, the `StateTriggerBase` class has overridable `OnAttached` and `OnDetached` methods.

IMPORTANT

The `StateTrigger.IsActive` bindable property hides the inherited `StateTriggerBase.IsActive` property.

The following XAML example shows a `Style` that includes `StateTrigger` objects:

```

<Style TargetType="Grid">
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup>
                <VisualState x:Name="Checked">
                    <VisualState.StateTriggers>
                        <StateTrigger IsActive="{Binding IsToggled}"
                            IsActiveChanged="OnCheckedStateIsActiveChanged" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor"
                            Value="Black" />
                    </VisualState.Setters>
                </VisualState>
                <VisualState x:Name="Unchecked">
                    <VisualState.StateTriggers>
                        <StateTrigger IsActive="{Binding IsToggled, Converter={StaticResource
inverseBooleanConverter}}"
                            IsActiveChanged="OnUncheckedStateIsActiveChanged" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor"
                            Value="White" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>

```

In this example, the implicit `Style` targets `Grid` objects. When the `IsToggled` property of the bound object is `true`, the background color of the `Grid` is set to black. When the `IsToggled` property of the bound object becomes `false`, a `VisualStyle` change is triggered, and the background color of the `Grid` becomes white.

In addition, every time a `VisualStyle` change occurs, the `IsActiveChanged` event for the `VisualStyle` is fired. Each `VisualStyle` registers an event handler for this event:

```

void OnCheckedStateIsActiveChanged(object sender, EventArgs e)
{
    StateTriggerBase stateTrigger = sender as StateTriggerBase;
    Console.WriteLine($"Checked state active: {stateTrigger.IsActive}");
}

void OnUncheckedStateIsActiveChanged(object sender, EventArgs e)
{
    StateTriggerBase stateTrigger = sender as StateTriggerBase;
    Console.WriteLine($"Unchecked state active: {stateTrigger.IsActive}");
}

```

In this example, when a handler for the `IsActiveChanged` event is fired, the handler outputs whether the `VisualStyle` is active or not. For example, the following messages are output to the console window when changing from the `Checked` visual state to the `Unchecked` visual state:

```

Checked state active: False
Unchecked state active: True

```

NOTE

Custom state triggers can be created by deriving from the `StateTriggerBase` class, and overriding the `OnAttached` and `OnDetached` methods to perform any required registrations and cleanup.

Adaptive trigger

An `AdaptiveTrigger` triggers a `VisualState` change when the window is a specified height or width. This trigger has two bindable properties:

- `MinWindowHeight`, of type `double`, which indicates the minimum window height at which the `VisualState` should be applied.
- `MinWindowWidth`, of type `double`, which indicates the minimum window width at which the `VisualState` should be applied.

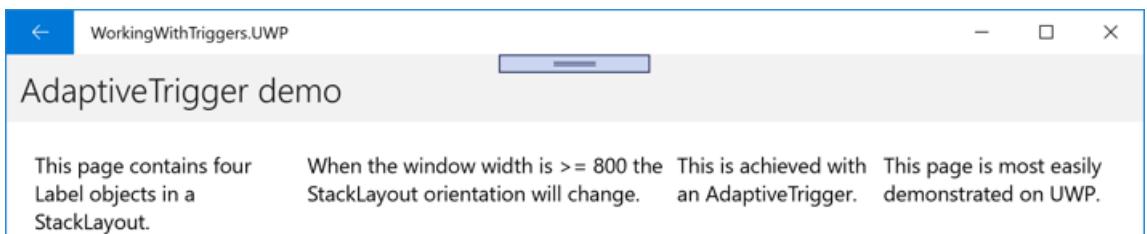
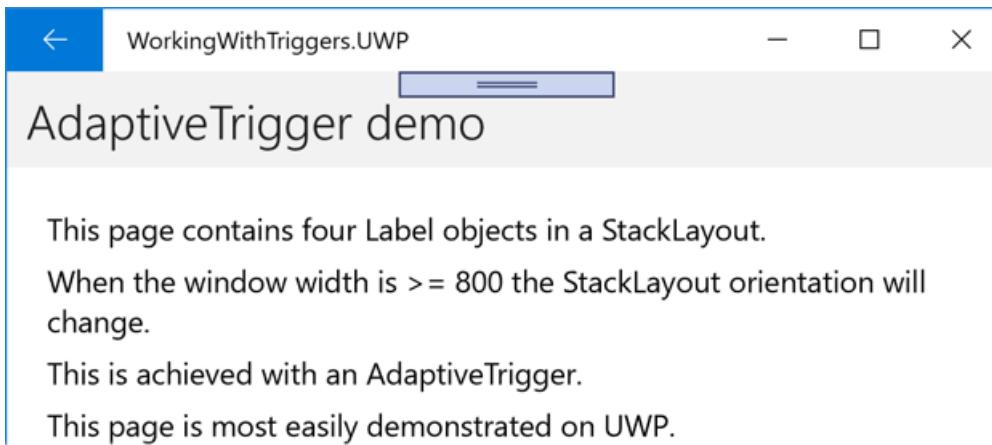
NOTE

The `AdaptiveTrigger` derives from the `StateTriggerBase` class and can therefore attach an event handler to the `IsActiveChanged` event.

The following XAML example shows a `Style` that includes `AdaptiveTrigger` objects:

```
<Style TargetType="StackLayout">
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup>
                <VisualState x:Name="Vertical">
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="0" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="Orientation"
                               Value="Vertical" />
                    </VisualState.Setters>
                </VisualState>
                <VisualState x:Name="Horizontal">
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="800" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="Orientation"
                               Value="Horizontal" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>
```

In this example, the implicit `Style` targets `StackLayout` objects. When the window width is between 0 and 800 device-independent units, `StackLayout` objects to which the `Style` is applied will have a vertical orientation. When the window width is ≥ 800 device-independent units, the `VisualState` change is triggered, and the `StackLayout` orientation changes to horizontal:



The `MinWindowHeight` and `MinWindowSize` properties can be used independently or in conjunction with each other. The following XAML shows an example of setting both properties:

```
<AdaptiveTrigger MinWindowWidth="800"  
                 MinWindowHeight="1200"/>
```

In this example, the `AdaptiveTrigger` indicates that the corresponding `VisualState` will be applied when the current window width is >= 800 device-independent units and the current window height is >= 1200 device-independent units.

Compare state trigger

The `CompareStateTrigger` triggers a `VisualState` change when a property is equal to a specific value. This trigger has two bindable properties:

- `Property`, of type `object`, which indicates the property being compared by the trigger.
- `Value`, of type `object`, which indicates the value at which the `VisualState` should be applied.

NOTE

The `CompareStateTrigger` derives from the `StateTriggerBase` class and can therefore attach an event handler to the `IsActiveChanged` event.

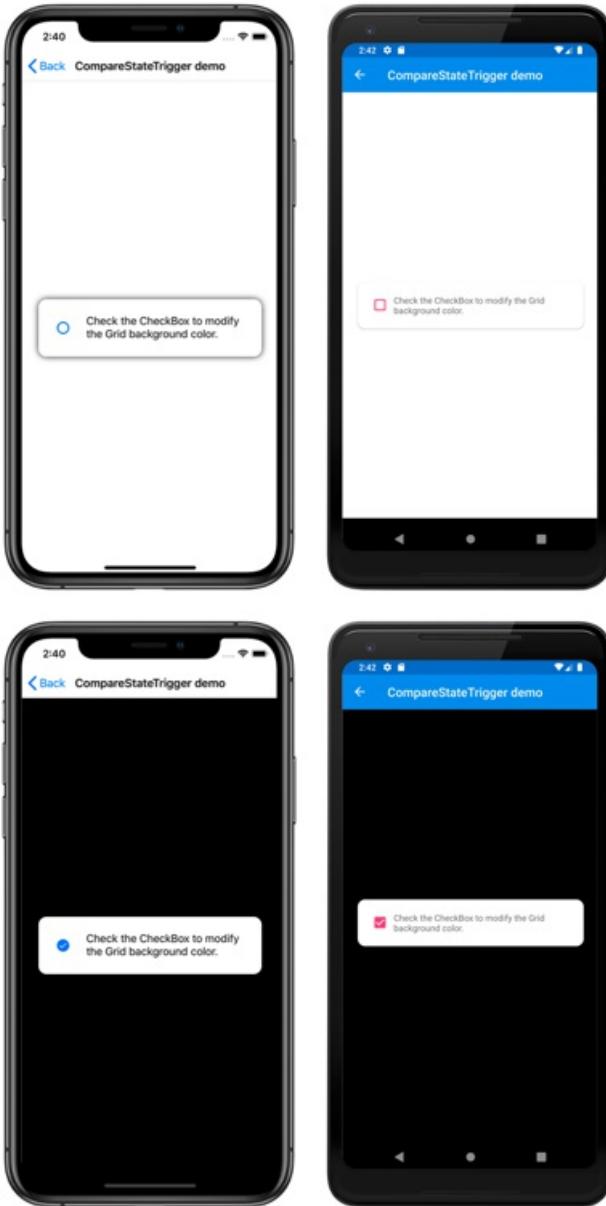
The following XAML example shows a `Style` that includes `CompareStateTrigger` objects:

```

<Style TargetType="Grid">
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup>
                <VisualState x:Name="Checked">
                    <VisualState.StateTriggers>
                        <CompareStateTrigger Property="{Binding Source={x:Reference checkBox}, Path=IsChecked}" Value="True" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor" Value="Black" />
                    </VisualState.Setters>
                </VisualState>
                <VisualState x:Name="Unchecked">
                    <VisualState.StateTriggers>
                        <CompareStateTrigger Property="{Binding Source={x:Reference checkBox}, Path=IsChecked}" Value="False" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor" Value="White" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>
...
<Grid>
    <Frame BackgroundColor="White"
        CornerRadius="12"
        Margin="24"
        HorizontalOptions="Center"
        VerticalOptions="Center">
        <StackLayout Orientation="Horizontal">
            <CheckBox x:Name="checkBox"
                VerticalOptions="Center" />
            <Label Text="Check the CheckBox to modify the Grid background color."
                VerticalOptions="Center" />
        </StackLayout>
    </Frame>
</Grid>

```

In this example, the implicit `Style` targets `Grid` objects. When the `.IsChecked` property of the `CheckBox` is `false`, the background color of the `Grid` is set to white. When the `CheckBox.IsChecked` property becomes `true`, a `VisualState` change is triggered, and the background color of the `Grid` becomes black:



Device state trigger

The `DeviceStateTrigger` triggers a `VisualState` change based on the device platform the app is running on. This trigger has a single bindable property:

- `Device`, of type `string`, which indicates the device platform on which the `VisualState` should be applied.

NOTE

The `DeviceStateTrigger` derives from the `StateTriggerBase` class and can therefore attach an event handler to the `IsActiveChanged` event.

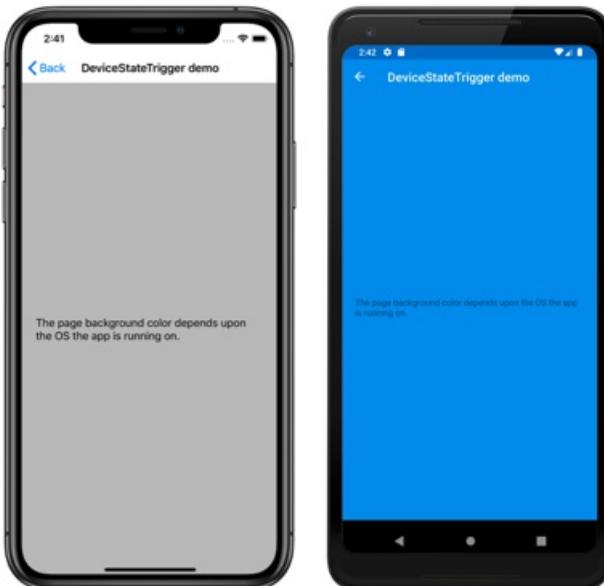
The following XAML example shows a `Style` that includes `DeviceStateTrigger` objects:

```

<Style x:Key="DeviceStateTriggerPageStyle"
    TargetType="ContentPage">
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup>
                <VisualState x:Name="iOS">
                    <VisualState.StateTriggers>
                        <DeviceStateTrigger Device="iOS" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor"
                            Value="Silver" />
                    </VisualState.Setters>
                </VisualState>
                <VisualState x:Name="Android">
                    <VisualState.StateTriggers>
                        <DeviceStateTrigger Device="Android" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor"
                            Value="#2196F3" />
                    </VisualState.Setters>
                </VisualState>
                <VisualState x:Name="UWP">
                    <VisualState.StateTriggers>
                        <DeviceStateTrigger Device="UWP" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor"
                            Value="Aquamarine" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>

```

In this example, the explicit `Style` targets `ContentPage` objects. `ContentPage` objects that consume the style set their background color to silver on iOS, to pale blue on Android, and to aquamarine on UWP. The following screenshots show the resulting pages on iOS and Android:



Orientation state trigger

The `OrientationStateTrigger` triggers a `VisualState` change when the orientation of the device changes. This trigger has a single bindable property:

- **Orientation**, of type **DeviceOrientation**, which indicates the orientation to which the **VisualState** should be applied.

NOTE

The **OrientationStateTrigger** derives from the **StateTriggerBase** class and can therefore attach an event handler to the **IsActiveChanged** event.

The following XAML example shows a **Style** that includes **OrientationStateTrigger** objects:

```
<Style x:Key="OrientationStateTriggerPageStyle"
    TargetType="ContentPage">
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup>
                <VisualState x:Name="Portrait">
                    <VisualState.StateTriggers>
                        <OrientationStateTrigger Orientation="Portrait" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor"
                            Value="Silver" />
                    </VisualState.Setters>
                </VisualState>
                <VisualState x:Name="Landscape">
                    <VisualState.StateTriggers>
                        <OrientationStateTrigger Orientation="Landscape" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Property="BackgroundColor"
                            Value="White" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>
```

In this example, the explicit **Style** targets **ContentPage** objects. **ContentPage** objects that consume the style set their background color to silver when the orientation is portrait, and set their background color to white when the orientation is landscape.

Related links

- [Triggers Sample](#)
- [Xamarin.Forms Visual State Manager](#)
- [Xamarin.Forms Trigger API](#)

Controls Reference

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The user 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](#).

The four main control groups used to create the user interface of a Xamarin.Forms application are as follows:

- [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](#). A class diagram that shows the hierarchy of types that are typically used to build a user interface in Xamarin.Forms can be found at [Xamarin.Forms Controls Class Hierarchy](#).

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 and Android 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.

NOTE

Pages, Layouts, and Views derive from the `VisualElement` class. The `VisualElement` class provides a variety of properties, methods, and events that are useful in deriving classes. For more information, see [VisualElement properties, methods, and events](#).

In addition to the controls supplied with Xamarin.Forms, third-party controls are available. For more information, see [Third Party Controls](#).

Related Links

- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Controls Class Hierarchy](#)
- [API Documentation](#)

Xamarin.Forms Pages

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

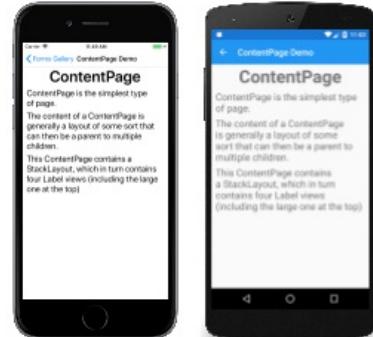
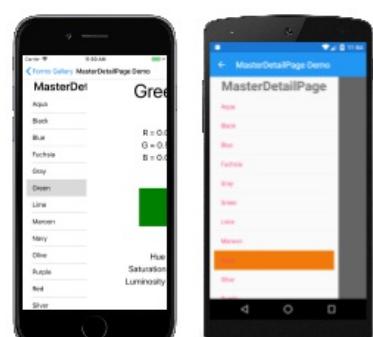
Xamarin.Forms Pages represent cross-platform mobile application screens.

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.



Pages

Xamarin.Forms supports the following page types:

TYPE	DESCRIPTION	APPEARANCE
<code>ContentPage</code>	<p><code>ContentPage</code> is the simplest and most common type of page. Set the <code>Content</code> property to a single <code>View</code> object, which is most often a <code>Layout</code> such as <code>StackLayout</code>, <code>Grid</code>, or <code>ScrollView</code>.</p> <p>API Documentation</p>	 <p>C# code for this page / XAML page</p>
<code>FlyoutPage</code>	<p>A <code>FlyoutPage</code> manages two panes of information. Set the <code>Flyout</code> property to a page generally showing a list or menu. Set the <code>Detail</code> property to a page showing a selected item from the flyout page. The <code>IsPresented</code> property governs whether the flyout or detail page is visible.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page with <code>code-behind</code></p>

TYPE	DESCRIPTION	APPEARANCE
<code>NavigationPage</code>	The <code>NavigationPage</code> 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 <code>NavigationPage</code> object.	 C# code for this page / XAML Page with <code>code=behind</code>
<code>TabbedPage</code>	<code>TabbedPage</code> derives from the abstract <code>MultiPage</code> class and allows navigation among child pages using tabs. Set the <code>Children</code> property to a collection of pages, or set the <code>ItemsSource</code> property to a collection of data objects and the <code>ItemTemplate</code> property to a <code>DataTemplate</code> describing how each object is to be visually represented.	 C# code for this page / XAML page
<code>CarouselPage</code>	<code>CarouselPage</code> derives from the abstract <code>MultiPage</code> class and allows navigation among child pages through finger swiping. Set the <code>Children</code> property to a collection of <code>ContentPage</code> objects, or set the <code>ItemsSource</code> property to a collection of data objects and the <code>ItemTemplate</code> property to a <code>DataTemplate</code> describing how each object is to be visually represented.	 C# code for this page / XAML page
<code>TemplatedPage</code>	<code>TemplatedPage</code> displays full-screen content with a control template, and is the base class for <code>ContentPage</code> .	

Related links

- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API Documentation](#)

Xamarin.Forms Layouts

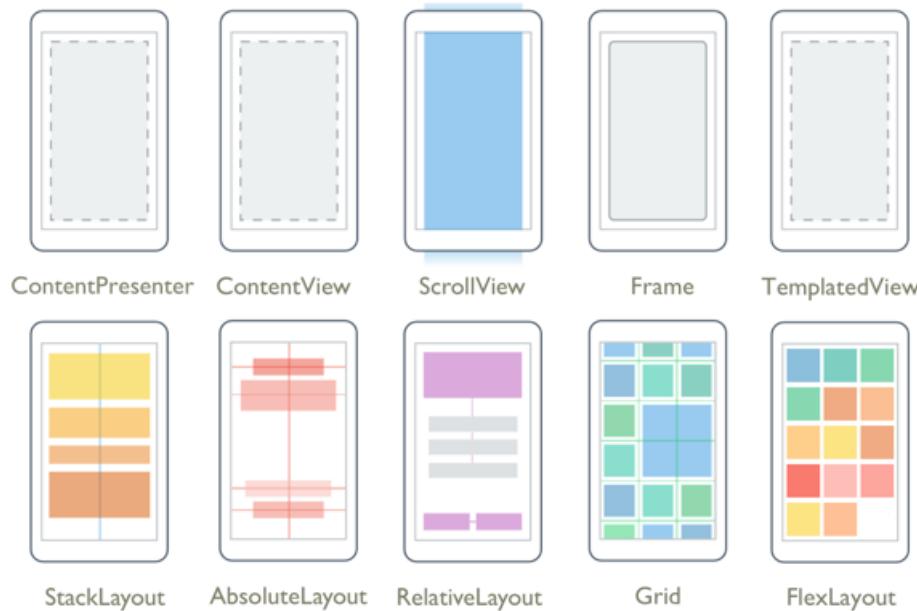
8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms Layouts are used to compose user-interface controls into visual structures.

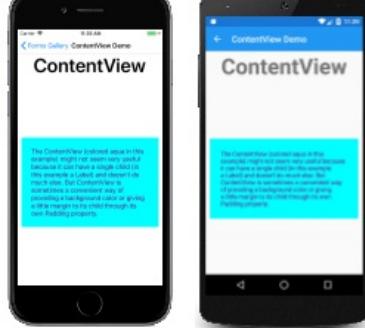
The `Layout` and `Layout<T>` classes in Xamarin.Forms are specialized subtypes of views that act as containers for views and other layouts. The `Layout` class itself derives from `View`. A `Layout` derivative typically contains logic to set the position and size of child elements in Xamarin.Forms applications.

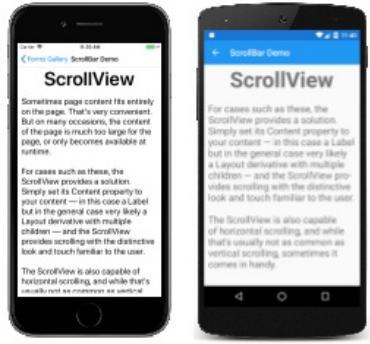


The classes that derive from `Layout` can be divided into two categories:

Layouts with Single Content

These classes derive from `Layout`, which defines `Padding` and `IsClippedToBounds` properties:

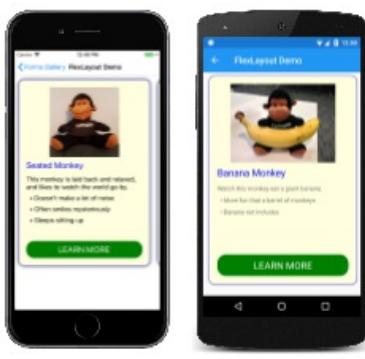
Type	Description	Appearance
<code>ContentView</code>	<p><code>ContentView</code> contains a single child that is set with the <code>Content</code> property. The <code>Content</code> property can be set to any <code>View</code> derivative, including other <code>Layout</code> derivatives. <code>ContentView</code> is mostly used as a structural element and serves as a base class to <code>Frame</code>.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page</p>

TYPE	DESCRIPTION	APPEARANCE
Frame	<p>The Frame class derives from ContentView and displays a border, or frame, around its child. The Frame class has a default Padding value of 20, and also defines BorderColor, CornerRadius, and HasShadow properties.</p> <p>API Documentation / Guide / Sample</p>	
		C# code for this page / XAML page
ScrollView	<p>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.</p> <p>API Documentation / Guide / Sample</p>	
		C# code for this page / XAML page
TemplatedView	<p>TemplatedView displays content with a control template, and is the base class for ContentView.</p> <p>API Documentation / Guide</p>	
ContentPresenter	<p>ContentPresenter is a layout manager for templated views, used within a ControlTemplate to mark where the content that is to be presented appears.</p> <p>API Documentation / Guide</p>	

Layouts with Multiple Children

These classes derive from [Layout<View>](#):

TYPE	DESCRIPTION	APPEARANCE
<code>StackLayout</code>	<p><code>StackLayout</code> positions child elements in a stack either horizontally or vertically based on the <code>Orientation</code> property. The <code>Spacing</code> property governs the spacing between the children, and has a default value of 6.</p> <p>API Documentation / Guide / Sample</p>	
		C# code for this page / XAML page
<code>Grid</code>	<p><code>Grid</code> positions its child elements in a grid of rows and columns. A child's position is indicated using the attached properties <code>Row</code>, <code>Column</code>, <code>RowSpan</code>, and <code>ColumnSpan</code>.</p> <p>API Documentation / Guide / Sample</p>	
		C# code for this page / XAML page
<code>AbsoluteLayout</code>	<p><code>AbsoluteLayout</code> positions child elements at specific locations relative to its parent. A child's position is indicated using the attached properties <code>LayoutBounds</code> and <code>LayoutFlags</code>. An <code>AbsoluteLayout</code> is useful for animating the positions of views.</p> <p>API Documentation / Guide / Sample</p>	
		C# code for this page / XAML page with code-behind
<code>RelativeLayout</code>	<p><code>RelativeLayout</code> positions child elements relative to the <code>RelativeLayout</code> itself or to their siblings. A child's position is indicated using the attached properties that are set to objects of type <code>Constraint</code> and <code>BoundsConstraint</code>.</p> <p>API Documentation / Guide / Sample</p>	
		C# code for this page / XAML page

TYPE	DESCRIPTION	APPEARANCE
FlexLayout	<p>FlexLayout is based on the CSS Flexible Box Layout Module, commonly known as <i>flex layout</i> or <i>flex-box</i>. FlexLayout defines six bindable properties and five attached bindable properties that allow children to be stacked or wrapped with many alignment and orientation options.</p> <p>API Documentation / Guide / Sample</p>	
		C# code for this page / XAML page

Related links

- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API Documentation](#)

Xamarin.Forms Views

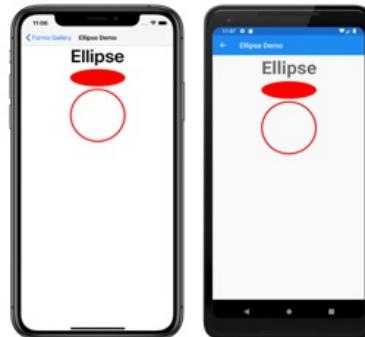
8/4/2022 • 10 minutes to read • [Edit Online](#)

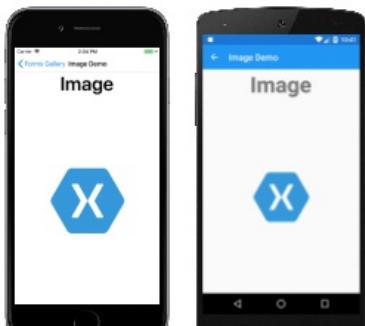
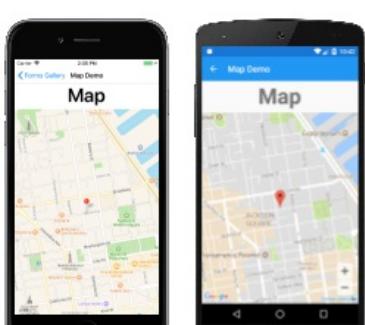
 [Download the sample](#)

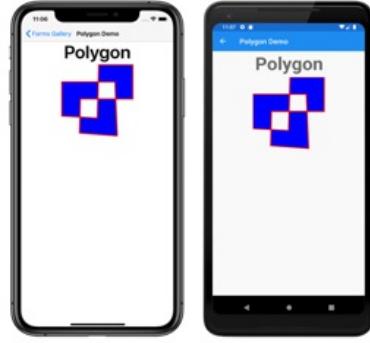
Xamarin.Forms views are the building blocks of cross-platform mobile user interfaces.

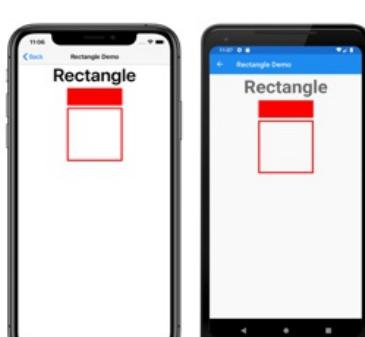
Views are user-interface objects such as labels, buttons, and sliders that are commonly known as *controls* or *widgets* in other graphical programming environments. The views supported by Xamarin.Forms all derive from the `View` class. They can be divided into several categories:

Views for presentation

Type	Description	Appearance
<code>BoxView</code>	<p><code>BoxView</code> displays a solid rectangle colored by the <code>Color</code> property. <code>BoxView</code> has a default size request of 40x40. For other sizes, assign the <code>WidthRequest</code> and <code>HeightRequest</code> properties.</p> <p>API Documentation / Guide / Sample 1, 2, 3, 4, 5, and 6</p>	 C# code for this page / XAML page
<code>Ellipse</code>	<p><code>Ellipse</code> displays an ellipse or circle of size <code>WidthRequest</code> x <code>HeightRequest</code>. To paint the inside of the ellipse, set its <code>Fill</code> property to a <code>Color</code>. To give the ellipse an outline, set its <code>Stroke</code> property to a <code>Color</code>.</p> <p>API Documentation / Guide / Sample</p>	 C# code for this page / XAML page
<code>Label</code>	<p><code>Label</code> displays single-line text strings or multi-line blocks of text, either with constant or variable formatting. Set the <code>Text</code> property to a string for constant formatting, or set the <code>FormattedText</code> property to a <code>FormattedString</code> object for variable formatting.</p> <p>API Documentation / Guide / Sample</p>	 C# code for this page / XAML page

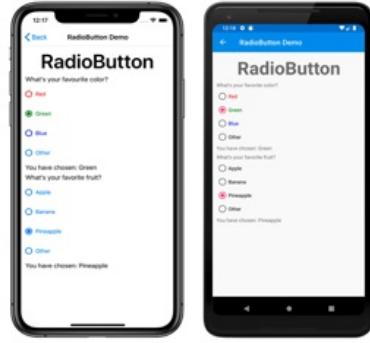
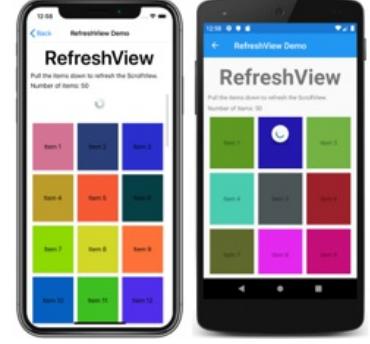
TYPE	DESCRIPTION	APPEARANCE
Line	<p>Line displays a line from a start point to an end point. The start point is represented by the <code>x1</code> and <code>y1</code> properties, while the end point is represented by the <code>x2</code> and <code>y2</code> properties. To color the line, set its <code>Stroke</code> property to a Color.</p> <p>API Documentation / Guide / Sample</p>	
Image	<p>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 <code>Stream</code> object.</p> <p>API Documentation / Guide / Sample</p>	
Map	<p>Map displays a map. The Xamarin.Forms.Maps NuGet package must be installed. Android and Universal Windows Platform require a map authorization key.</p> <p>API Documentation / Guide / Sample</p>	

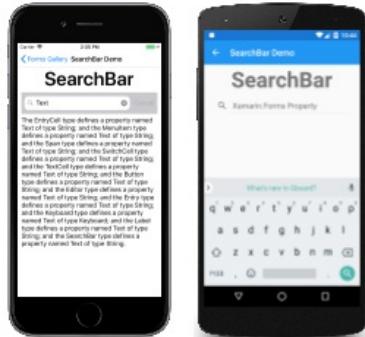
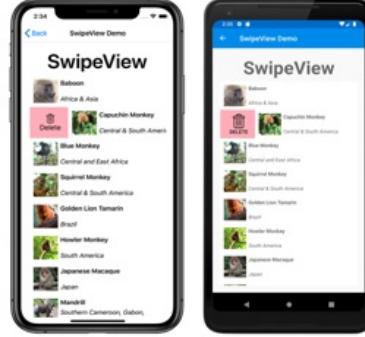
TYPE	DESCRIPTION	APPEARANCE
OpenGLView	<p>OpenGLView displays OpenGL graphics in iOS and Android projects. There is no support for the Universal Windows Platform. The iOS and Android projects require a reference to the OpenTK-1.0 assembly or the OpenTK version 1.0.0.0 assembly.</p> <p>OpenGLView is easier 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).</p> <p>This is the only graphics facility that is built into Xamarin.Forms, but a Xamarin.Forms application can also render graphics using SkiaSharp, or UrhoSharp.</p>	 <p>C# code for this page / XAML page with code-behind</p>
Path	<p>Path displays curves and complex shapes. The Data property specifies the shape to be drawn. To color the shape, set its Stroke property to a Color.</p>	 <p>API Documentation / Guide / Sample</p> <p>C# code for this page / XAML page</p>
Polygon	<p>Polygon displays a polygon. The Points property specifies the vertex points of the polygon, while the FillRule property specifies how the interior fill of the polygon is determined. To paint the inside of the polygon, set its Fill property to a Color. To give the polygon an outline, set its Stroke property to a Color.</p>	 <p>API Documentation / Guide / Sample</p> <p>C# code for this page / XAML page</p>

TYPE	DESCRIPTION	APPEARANCE
Polyline	<p><code>Polyline</code> displays a series of connected straight lines. The <code>Points</code> property specifies the vertex points of the polyline, while the <code>FillRule</code> property specifies how the interior fill of the polyline is determined. To paint the inside of the polyline, set its <code>Fill</code> property to a <code>Color</code>. To give the polyline an outline, set its <code>Stroke</code> property to a <code>Color</code>.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page</p>
Rectangle	<p><code>Rectangle</code> displays a rectangle or square. To paint the inside of the rectangle, set its <code>Fill</code> property to a <code>Color</code>. To give the rectangle an outline, set its <code>Stroke</code> property to a <code>Color</code>.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page</p>
WebView	<p><code>WebView</code> displays Web pages or HTML content, based on whether the <code>Source</code> property is set to a <code>UriWebViewSource</code> or an <code>HtmlWebViewSource</code> object.</p> <p>API Documentation / Guide / Sample 1 and 2</p>	 <p>C# code for this page / XAML page</p>

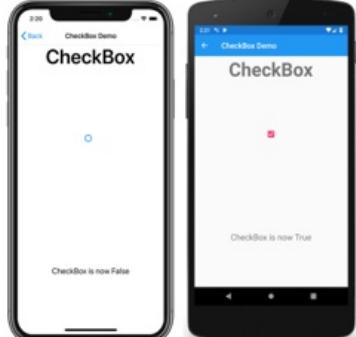
Views that initiate commands

TYPE	DESCRIPTION	APPEARANCE
------	-------------	------------

TYPE	DESCRIPTION	APPEARANCE
Button	<p><code>Button</code> is a rectangular object that displays text, and which fires a <code>Clicked</code> event when it's been pressed.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page with code-behind</p>
ImageButton	<p><code>ImageButton</code> is a rectangular object that displays an image, and which fires a <code>Clicked</code> event when it's been pressed.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page with code-behind</p>
RadioButton	<p><code>RadioButton</code> allows the selection of one option from a set, and fires a <code>CheckedChanged</code> event when selection occurs.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page with code-behind</p>
RefreshView	<p><code>RefreshView</code> is a container control that provides pull-to-refresh functionality for scrollable content. The <code>ICommand</code> defined by the <code>Command</code> property is executed when a refresh is triggered, and the <code>IsRefreshing</code> property indicates the current state of the control.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page with code-behind</p>

TYPE	DESCRIPTION	APPEARANCE
<code>SearchBar</code>	<p><code>SearchBar</code> 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 <code>Text</code> property provides access to the text, and the <code>SearchButtonPressed</code> event indicates that the button has been pressed.</p> <p>API Documentation / Guide / Sample</p>	
<code>SwipeView</code>	<p><code>SwipeView</code> is a container control that wraps around an item of content, and provides context menu items that are revealed by a swipe gesture. Each menu item is represented by a <code>SwipeItem</code>, which has a <code>Command</code> property that executes an <code>ICommand</code> when the item is tapped.</p> <p>API Documentation / Guide / Sample</p>	

Views for setting values

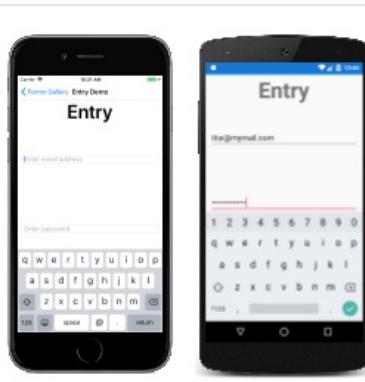
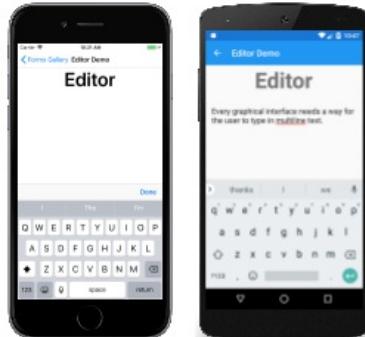
TYPE	DESCRIPTION	APPEARANCE
<code>CheckBox</code>	<p><code>CheckBox</code> allows the user to select a Boolean value using a type of button that can either be checked or empty. The <code>.IsChecked</code> property is the state of the <code>CheckBox</code>, and the <code>CheckedChanged</code> event is fired when the state changes.</p> <p>API Documentation / Guide / Sample</p>	

TYPE	DESCRIPTION	APPEARANCE
Slider	Slider allows the user to select a <code>double</code> value from a continuous range specified with the <code>Minimum</code> and <code>Maximum</code> properties.	 C# code for this page / XAML page
Stepper	Stepper allows the user to select a <code>double</code> value from a range of incremental values specified with the <code>Minimum</code> , <code>Maximum</code> , and <code>Increment</code> properties.	 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 <code>IsToggled</code> property is the state of the switch, and the <code>Toggled</code> event is fired when the state changes.	 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 <code>MinimumDate</code> and <code>MaximumDate</code> properties. The <code>Date</code> property is the selected date, and the <code>DateSelected</code> event is fired when that property changes.	 C# code for this page / XAML page

TYPE	DESCRIPTION	APPEARANCE
<code>TimePicker</code>	<p><code>TimePicker</code> allows the user to select a time with the platform time picker. The <code>Time</code> property is the selected time. An application can monitor changes in the <code>Time</code> property by installing a handler for the <code>PropertyChanged</code> event.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page</p>

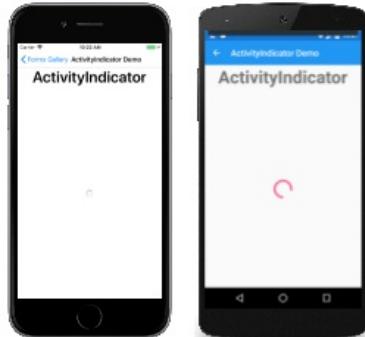
Views for editing text

These two classes derive from the `InputView` class, which defines the `Keyboard` property:

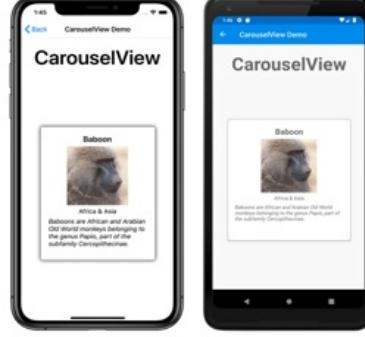
TYPE	DESCRIPTION	APPEARANCE
<code>Entry</code>	<p><code>Entry</code> allows the user to enter and edit a single line of text. The text is available as the <code>Text</code> property, and the <code>TextChanged</code> and <code>Completed</code> events are fired when the text changes or the user signals completion by tapping the enter key.</p> <p>Use an <code>Editor</code> for entering and editing multiple lines of text.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page</p>
<code>Editor</code>	<p><code>Editor</code> allows the user to enter and edit multiple lines of text. The text is available as the <code>Text</code> property, and the <code>TextChanged</code> and <code>Completed</code> events are fired when the text changes or the user signals completion.</p> <p>Use an <code>Entry</code> view for entering and editing a single line of text.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page</p>

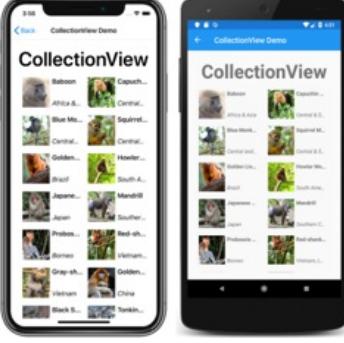
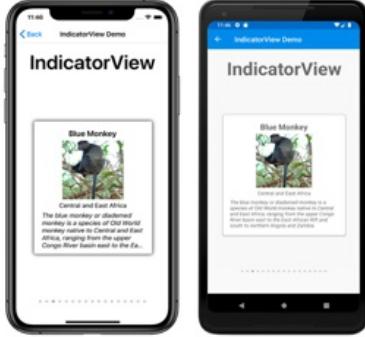
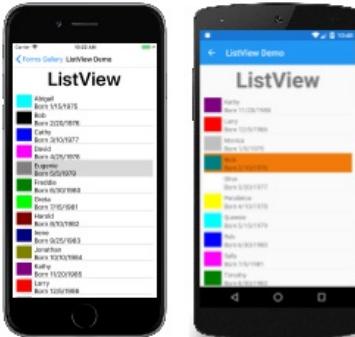
Views to indicate activity

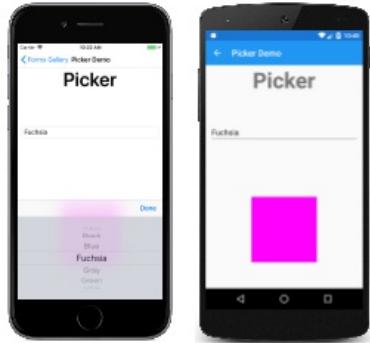
TYPE	DESCRIPTION	APPEARANCE
------	-------------	------------

TYPE	DESCRIPTION	APPEARANCE
ActivityIndicator	<p>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.</p> <p>If the activity's progress is known, use a ProgressBar instead.</p> <p>API Documentation / Guide / Sample</p>	
ProgressBar	<p>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.</p> <p>If the activity's progress is not known, use an ActivityIndicator instead.</p> <p>API Documentation / Guide / Sample</p>	

Views that display collections

TYPE	DESCRIPTION	APPEARANCE
CarouselView	<p>CarouselView displays a scrollable list of 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 CurrentItemChanged event signals that the currently displayed item has changed, which is available as the CurrentItem property.</p> <p>Guide / Sample</p>	

TYPE	DESCRIPTION	APPEARANCE
CollectionView	<p>CollectionView displays a scrollable list of selectable data items, using different layout specifications. It aims to provide a more flexible, and performant alternative to ListView. 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 SelectionChanged event signals that a selection has been made, which is available as the SelectedItem property.</p> <p>Guide / Sample</p>	
IndicatorView	<p>IndicatorView displays indicators that represent the number of items in a CarouselView. Set the CarouselView.IndicatorView property to the IndicatorView object to display indicators for the CarouselView.</p> <p>API Documentation / Guide / Sample</p>	
ListView	<p>ListView derives from ItemsView 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.</p> <p>API Documentation / Guide / Sample</p>	

TYPE	DESCRIPTION	APPEARANCE
<code>Picker</code>	<p><code>Picker</code> displays a selected item from a list of text strings, and allows selecting that item when the view is tapped. Set the <code>Items</code> property to a list of strings, or the <code>ItemsSource</code> property to a collection of objects. The <code>SelectedIndexChanged</code> event is fired when an item is selected.</p> <p>The <code>Picker</code> displays the list of items only when it's selected. Use <code>ListView</code> or <code>TableView</code> for a scrollable list that remains on the page.</p> <p>API Documentation / Guide</p>	 <p>C# code for this page / XAML page with code-behind</p>
<code>TableView</code>	<p><code>TableView</code> displays a list of rows of type <code>Cell</code> with optional headers and subheaders. Set the <code>Root</code> property to an object of type <code>TableRoot</code>, and add <code>TableSection</code> objects to that <code>TableRoot</code>. Each <code>TableSection</code> is a collection of <code>Cell</code> objects.</p> <p>API Documentation / Guide / Sample</p>	 <p>C# code for this page / XAML page</p>

Related links

- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API Documentation](#)

Xamarin.Forms Cells

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

Xamarin.Forms cells can be added to ListViews 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:

TYPE	DESCRIPTION	APPEARANCE
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	
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](#)

[C# code for this page](#) / [XAML page](#)

TYPE	DESCRIPTION	APPEARANCE
SwitchCell	<p>The <code>SwitchCell</code> contains text set with the <code>Text</code> property and an on/off switch initially set with the Boolean <code>On</code> property. Handle the <code>OnChanged</code> event to be notified when the <code>On</code> property changes.</p> <p>API Documentation / Guide</p>	
EntryCell	<p>The <code>EntryCell</code> defines a <code>Label</code> property that identifies the cell and a single line of editable text in the <code>Text</code> property. Handle the <code>Completed</code> event to be notified when the user has completed the text entry.</p> <p>API Documentation / Guide</p>	

Related links

- [Xamarin.Forms FormsGallery sample](#)
- [Xamarin.Forms Samples](#)
- [Xamarin.Forms API Documentation](#)

Xamarin.Forms common control properties, methods, and events

8/4/2022 • 12 minutes to read • [Edit Online](#)

The Xamarin.Forms `VisualElement` class is the base class for most of the controls used in a Xamarin.Forms application. The `VisualElement` class defines many [properties](#), [methods](#), and [events](#) that are used in deriving classes.

Properties

The following properties are available on `VisualElement` objects.

`AnchorX`

The `AnchorX` property is a `double` value that defines the center point on the X axis for transforms such as scale and rotation. The default value is 0.5.

`AnchorY`

The `AnchorY` property is a `double` value that defines the center point on the Y axis for transforms such as scale and rotation. The default value is 0.5.

`Background`

The `Background` property is a `Brush` value that enables brushes to be used as the background in any control. The default value is `Brush.Default`.

`BackgroundColor`

The `BackgroundColor` property is a `Color` that determines the background color of the control. If unset, the background will be the default `Color` object, which renders as transparent.

`Behaviors`

The `Behaviors` property is a `List` of `Behavior` objects. Behaviors enable you to attach reusable functionality to elements by adding them to the `Behaviors` list. For more information about the `Behavior` class, see [Xamarin.Forms Behaviors](#).

`Bounds`

The `Bounds` property is a read-only `Rectangle` object that represents the space occupied by the control. The `Bounds` property value is assigned during the layout cycle. The `Rectangle` `struct` contains useful properties and methods for testing intersection and containment of rectangles. For more information, see the [Xamarin.Forms Rectangle API](#).

`Clip`

The `clip` property is a `Geometry` object that defines the outline of the contents of an element. To define a clip, use a `Geometry` object such as `EllipseGeometry` to set the element's `clip` property. Only the area that is within the region of the geometry will be visible. For more information, see [Clip with a Geometry](#).

`Effects`

The `Effects` property is a `List` of `Effect` objects, inherited from the `Element` class. Effects allow native controls to be customized, and are typically used for small styling changes. For more information about the `Effect` class, see [Xamarin.Forms Effects](#).

FlowDirection

The `FlowDirection` property is a `FlowDirection` enum value. Flow direction can be set to `MatchParent`, `LeftToRight`, or `RightToLeft` and determines the layout order and direction. The `FlowDirection` property is typically used to support languages that read right-to-left.

Height

The `Height` property is a read-only `double` value that describes the rendered height of the control. The `Height` property is calculated during the layout cycle and can't be directly set. The height of a control can be requested using the [HeightRequest property](#).

HeightRequest

The `HeightRequest` property is a `double` value that determines the desired height of the control. The absolute height of the control may not match the requested value. For more information, see [Request properties](#).

InputTransparent

The `InputTransparent` property is a `bool` that determines whether the control receives user input. The default value is `false`, ensuring that the element receives input. This property transfers to child elements when it's set. Setting the `InputTransparent` property to `true` on a layout class will result in all elements within the layout not receiving input.

.IsEnabled

The `.IsEnabled` property is a `bool` value that determines whether the control reacts to user input. The default value is `true`. Setting this property to false will prevent the control from accepting user input.

IsFocused

The `IsFocused` property is a `bool` value that describes whether the control is currently the focused object. Calling the `Focus` method on the control will result in the `IsFocused` value being set to true. Calling the `Unfocus` method will set this property to false.

IsTabStop

The `IsTabStop` property is a `bool` value that defines whether the control receives focus when the user is advancing through controls with the tab key. If this property is false, the `TabIndex` property will have no effect.

.isVisible

The `isVisible` property is a `bool` value that determines whether the control is rendered. Controls with the `isVisible` property set to false won't be displayed, won't be considered for space calculations during the layout cycle, and can't accept user input.

MinimumHeightRequest

The `MinimumHeightRequest` property is a `double` value that determines how overflow is handled when two elements are competing for limited space. Setting the `MinimumHeightRequest` property allows the layout process to scale the element down to the minimum dimension requested. If no `MinimumHeightRequest` is specified, the default value is -1 and the layout process will consider the `HeightRequest` to be the minimum value. This means that elements with no `MinimumHeightRequest` value will not have scalable height.

For more information, see [Minimum request properties](#).

MinimumWidthRequest

The `MinimumWidthRequest` property is a `double` value that determines how overflow is handled when two elements are competing for limited space. Setting the `MinimumWidthRequest` property allows the layout process to scale the element down to the minimum dimension requested. If no `MinimumWidthRequest` is specified, the default value is -1 and the layout process will consider the `WidthRequest` to be the minimum value. This means

that elements with no `MinimumWidthRequest` value will not have scalable width.

For more information, see [Minimum request properties](#).

Opacity

The `Opacity` property is a `double` value from zero to one that determines the opacity of the control during rendering. The default value for this property is 1.0. Values outside of the range from 0 to 1 will be clamped. The `Opacity` property is only applied if the `IsVisible` property is `true`. Opacity is applied iteratively. Therefore if a parent control has 0.5 opacity and its child has 0.5 opacity, the child will render with an effective 0.25 opacity value. Setting the `Opacity` property of an input control to 0 has undefined behavior.

Parent

The `Parent` property is inherited from the `Element` class. This property is an `Element` object that is the parent of control. The `Parent` property is typically set automatically on an element when it's added as a child of another element.

Resources

The `Resources` property is a `ResourceDictionary` instance that is populated with key/value pairs that are typically populated at runtime from XAML. This dictionary allows application developers to reuse objects defined in XAML at both compile time and run time. The keys in the dictionary are populated from the `x:Key` attribute of the XAML tag. The object created from XAML is inserted into the `ResourceDictionary` for the specified key. once it has been initialized.

For more information, see [Resource Dictionaries](#).

Rotation

The `Rotation` property is a `double` value between zero and 360 that defines the rotation about the Z axis in degrees. The default value of this property is 0. Rotation is applied relative to the `AnchorX` and `AnchorY` values.

RotationX

The `RotationX` property is a `double` value between zero and 360 that defines the rotation about the X axis in degrees. The default value of this property is 0. Rotation is applied relative to the `AnchorX` and `AnchorY` values.

RotationY

The `RotationY` property is a `double` value between zero and 360 that defines the rotation about the Y axis in degrees. The default value of this property is 0. Rotation is applied relative to the `AnchorX` and `AnchorY` values.

Scale

The `Scale` property is a `double` value that defines the scale of the control. The default value of this property is 1.0. Scale is applied relative to the `AnchorX` and `AnchorY` values.

ScaleX

The `ScaleX` property is a `double` value that defines the scale of the control along the X axis. The default value of this property is 1.0. The `scaleX` property is applied relative to the `AnchorX` value.

ScaleY

The `ScaleY` property is a `double` value that defines the scale of the control along the Y axis. The default value of this property is 1.0. The `scaleY` property is applied relative to the `AnchorY` value.

Style

The `Style` property is inherited from the `NavigableElement` class. This property is an instance of the `Style` class. The `Style` class contains triggers, setters, and behaviors that define the appearance and behavior of visual elements. For more information, see [Xamarin.Forms XAML Styles](#).

StyleClass

The `styleClass` property is a list of `string` objects that represent the names of `Style` classes. This property is inherited from the `NavigableElement` class. The `StyleClass` property allows multiple style attributes to be applied to a `VisualElement` instance. For more information, see [Xamarin.Forms Style Classes](#).

TabIndex

The `TabIndex` property is an `int` value that defines the control order when advancing through controls with the tab key. The `TabIndex` property is the implementation for the property defined on the `ITabStopElement` interface, which the `VisualElement` class implements.

TranslationX

The `TranslationX` property is a `double` value that defines the delta translation to be applied on the X axis. Translation is applied after layout and is typically used for applying animations. Translating an element outside the bounds of its parent container may prevent inputs from working.

For more information, see [Animation in Xamarin.Forms](#).

TranslationY

The `TranslationY` property is a `double` value that defines the delta translation to be applied on the Y axis. Translation is applied after layout and is typically used for applying animations. Translating an element outside the bounds of its parent container may prevent inputs from working.

For more information, see [Animation in Xamarin.Forms](#).

Triggers

The `Triggers` property is a read-only `List` of `TriggerBase` objects. Triggers allow application developers to express actions in XAML that change the visual appearance of controls in response to event or property changes. For more information, see [Xamarin.Forms Triggers](#).

Visual

The `visual` property is an `IVisual` instance that enables renderers to be created and selectively applied to `VisualElement` instances. The `visual` property is set to match its parent so defining a renderer on a component will also apply to any children of that component. If no custom renderer is set on a control or its ancestors, the default Xamarin.Forms renderer will be used. For more information, see [Xamarin.Forms Visual](#).

Width

The `width` property is a read-only `double` value that describes the rendered width of the control. The `width` property is calculated during the layout cycle and can't be directly set. The width of a control can be requested using the `WidthRequest` property.

WidthRequest

The `WidthRequest` property is a `double` value that determines the desired width of the control. The absolute width of the control may not match the requested value. For more information, see [Request properties](#).

X

The `x` property is a read-only `double` value that describes the current X position of the control.

Y

The `y` property is a read-only `double` value that describes the current Y position of the control.

Methods

The following methods are available on the `VisualElement` class. For a complete list, see [VisualElement API](#)

Methods.

FindByName

The `FindByName` method is inherited from the `Element` class and has the following signature:

```
public object FindByName (string name)
```

This method searches all child elements for the provided `name` argument and returns the element that has the specified name. If no match is found, `null` is returned.

Focus

The `Focus` method attempts to set focus on the element. This method has the following signature:

```
public bool Focus ()
```

The `Focus` method returns `true` if keyboard focus was successfully set and `false` if the method call did not result in a focus change. The element must be able to receive focus for this method to work. Calling the `Focus` method on elements that are offscreen or unrealized has undefined behavior.

Unfocus

The `Unfocus` method attempts to remove focus on the element. This method has the following signature:

```
public void Unfocus ()
```

The element must already have focus for this method to work.

Events

The following events are available on the `VisualElement` class. For a complete list, see [Xamarin.Forms VisualElement Events](#).

Focused

The `Focused` event is raised whenever the `VisualElement` instance receives focus. This event is not bubbled through the Xamarin.Forms stack, it's received directly from the native control. This event is emitted by the `IsFocused` property setter.

SizeChanged

The `SizeChanged` event is raised whenever the `VisualElement` instance `Height` or `Width` properties change. If developers wish to respond directly to the size change, instead of responding to the post-change event, they should implement the `OnSizeAllocated` virtual method instead.

Unfocused

The `Unfocused` event is raised whenever the `VisualElement` instance loses focus. This event is not bubbled through the Xamarin.Forms stack, it's received directly from the native control. This event is emitted by the `IsFocused` property setter.

Units of Measurement

Android, iOS, and UWP platforms all have different measurement units that can vary across devices. Xamarin.Forms uses a platform-independent unit of measurement that normalizes units across devices and platforms. There are 160 units per inch, or 64 units per centimeter, in Xamarin.Forms.

Request properties

Properties whose names contain "request" define a desired value, which may not match the actual rendered value. For example, `HeightRequest` might be set to 150 but if the layout only allows room for 100 units, the rendered `Height` of the control will only be 100. Rendered size is affected by available space and contained components.

Minimum request properties

Minimum request properties include `MinimumHeightRequest` and `MinimumWidthRequest`, and are intended to enable more precise control over how elements handle overflow relative to each other. However, layout behavior related to these properties has some important considerations.

Unspecified minimum property values

If a minimum value is not set, the minimum property defaults to -1. The layout process ignores this value and considers the absolute value to be the minimum. The practical consequence of this behavior is that an element with no minimum value specified **will not** shrink. An element with a minimum value specified **will** shrink.

The following XAML shows two `BoxView` elements in a horizontal `StackLayout`:

```
<StackLayout Orientation="Horizontal">
    <BoxView HeightRequest="100" BackgroundColor="Purple" WidthRequest="500"></BoxView>
    <BoxView HeightRequest="100" BackgroundColor="Green" WidthRequest="500" MinimumWidthRequest="250">
    </BoxView>
</StackLayout>
```

The first `BoxView` instance requests a width of 500 and does not specify a minimum width. The second `BoxView` instance requests a width of 500 and a minimum width of 250. If the parent `StackLayout` element is not wide enough to contain both components at their requested width, the first `BoxView` instance will be considered by the layout process to have a minimum width of 500 because no other valid minimum is specified. The second `BoxView` instance is allowed to scale down to 250 and it will shrink to fit until its width hits 250 units.

If the desired behavior is for the first `BoxView` instance to scale down with no minimum width, the `MinimumWidthRequest` must be set to a valid value, such as 0.

Minimum and absolute property values

The behavior is undefined when the minimum value is greater than the absolute value. For example, if `WidthRequest` is set to 100, the `MinimumWidthRequest` property should never exceed 100. When specifying a minimum property value, you should always specify an absolute value to ensure the absolute value is greater than the minimum value.

Minimum properties within a Grid

`Grid` layouts have their own system for relative sizing of rows and columns. Using `MinimumWidthRequest` or `MinimumHeightRequest` within a `Grid` layout will not have an effect. For more information, see [Xamarin.Forms Grid](#).

Related links

- [VisualElement API](#)

Xamarin.Forms Third-Party Controls

8/4/2022 • 2 minutes to read • [Edit Online](#)

In addition to the controls supplied with Xamarin.Forms, third-party controls are available from the following companies:

- [Telerik](#)
- [Syncfusion](#)
- [DevExpress](#)
- [Infragistics](#)
- [ComponentOne](#)
- [Steema](#)

These controls provide additional support for Xamarin.Forms developers by augmenting the standard controls with custom controls and services.

Xamarin.Forms BoxView

8/4/2022 • 18 minutes to read • [Edit Online](#)



[Download the sample](#)

`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.

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:

```

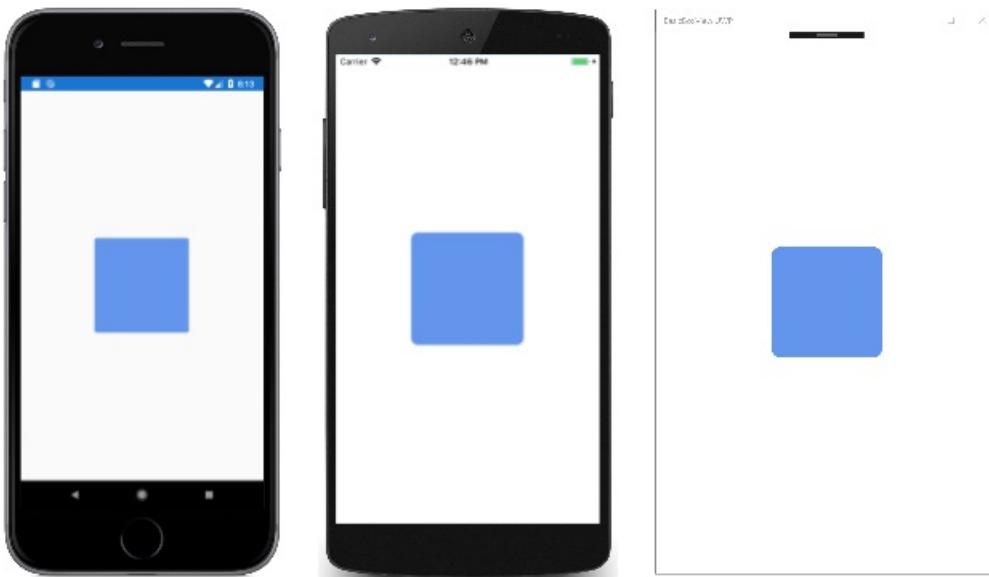
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:BasicBoxView"
    x:Class="BasicBoxView.MainPage">

    <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>

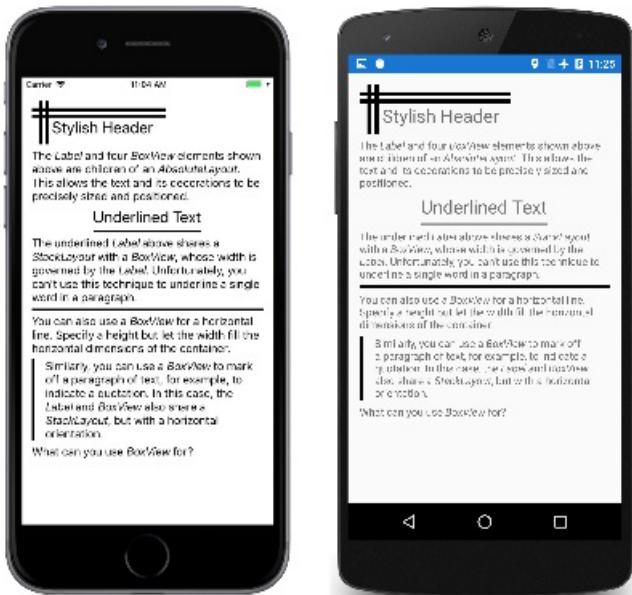
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="BoxView">
                <Setter Property="Color" Value="Black" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <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`:

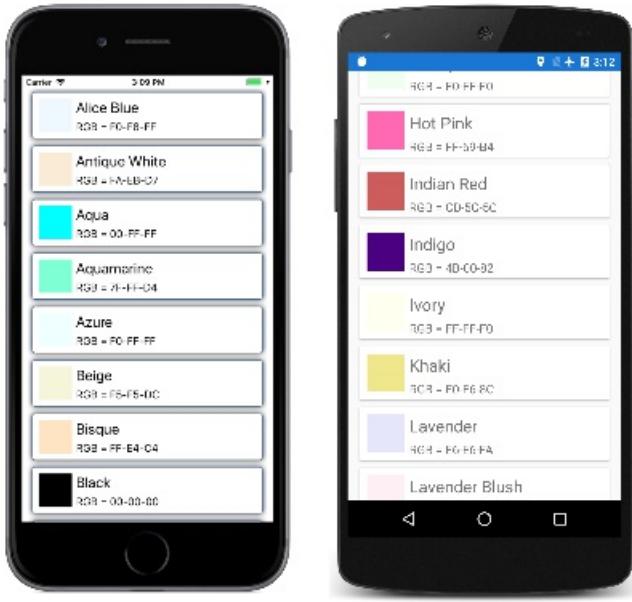
```
<BoxView HeightRequest="3" />
```

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 the `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; }

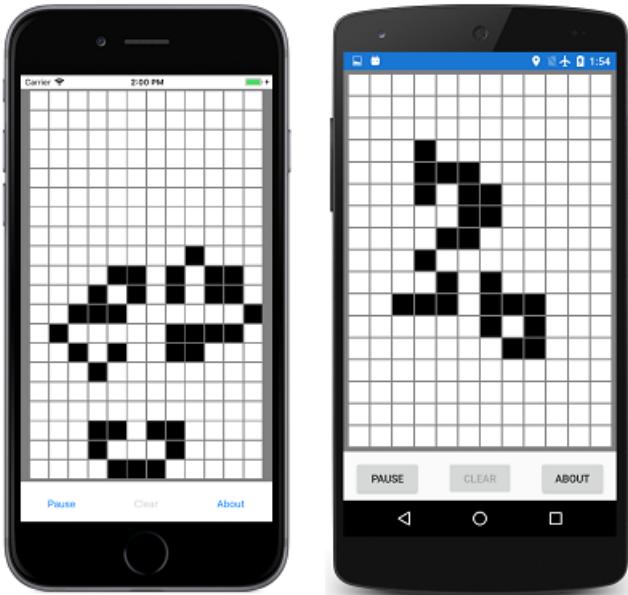
    public bool IsAlive
    {
        set
        {
            if (isAlive != value)
            {
                isAlive = value;
                BackgroundColor = isAlive ? Color.Black : Color.White;
            }
        }
        get
        {
            return isAlive;
        }
    }
}

```

`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.

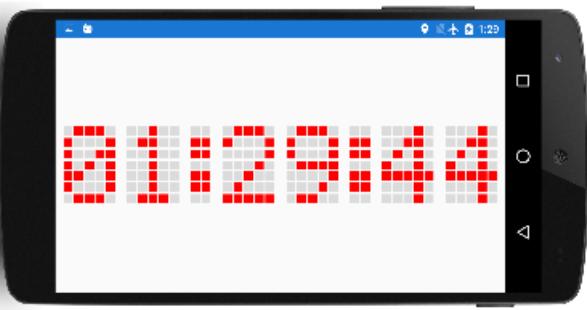
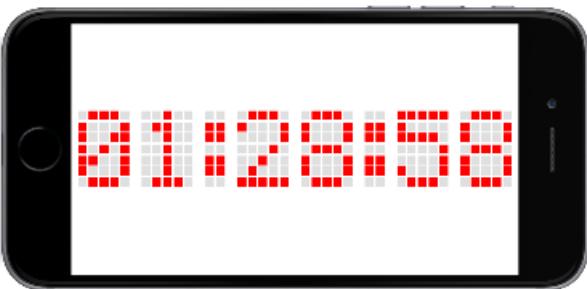
`LifeCell` also installs a `TapGestureRecognizer` to allow the user to toggle the state of cells by tapping them. The class translates the `Tapped` event from the gesture recognizer into its own `Tapped` event.

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, 0, 0, 0, 1}, { 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;

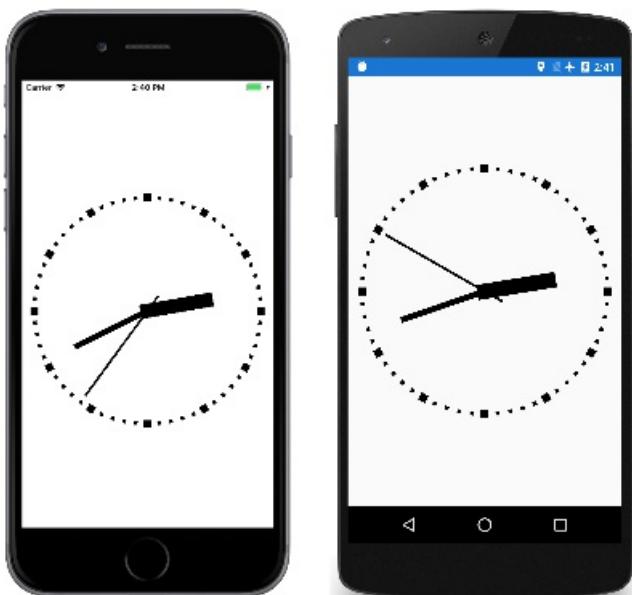
        // Set the dot colors for each digit separately.
        SetDotMatrix(0, hour / 10);
        SetDotMatrix(1, hour % 10);
        SetDotMatrix(2, dateTime.Minute / 10);
        SetDotMatrix(3, dateTime.Minute % 10);
        SetDotMatrix(4, dateTime.Second / 10);
        SetDotMatrix(5, dateTime.Second % 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.

Related Links

- [Basic BoxView \(sample\)](#)
- [Text Decoration \(sample\)](#)
- [ListView Colors \(sample\)](#)
- [Game of Life \(sample\)](#)
- [Dot-Matrix Clock \(sample\)](#)
- [BoxView Clock \(sample\)](#)
- [BoxView](#)

Images in Xamarin.Forms

8/4/2022 • 12 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

Display images

Xamarin.Forms uses the `Image` view to display images on a page. It has several 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 (i.e. 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](#), [downloaded](#), or loaded from a stream. In addition, font icons can be displayed by the `Image` view by specifying the font icon data in a `FontImageSource` object. For more information, see [Display font icons](#) in the [Fonts](#) guide.

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 (i.e. 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)** - By default, images should be placed in the application's root directory with **Build Action: Content**. Alternatively, images can be placed in a different directory which is then specified with a platform-specific. For more information, see [Default image directory on Windows](#).

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](#).

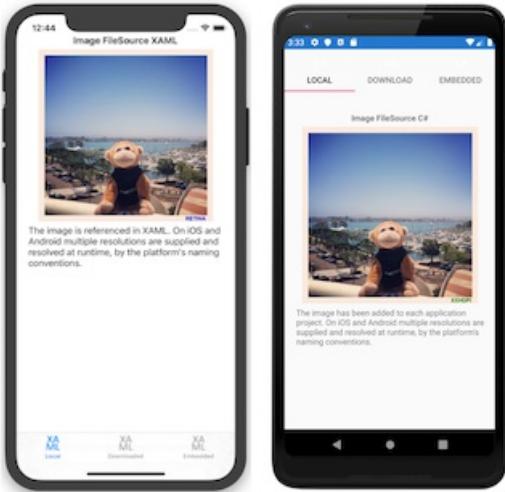
Adhering to these rules for file naming and placement allows the following XAML to load and display the image on all platforms:

```
<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)

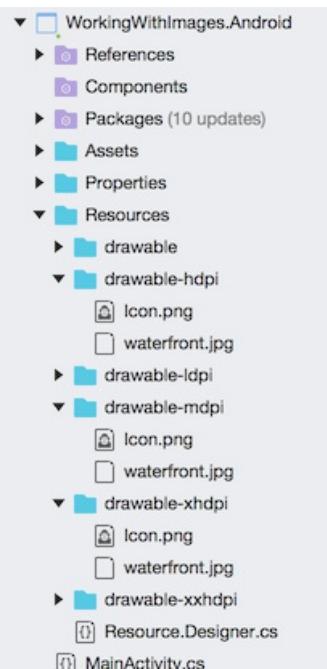
iOS, Android, and UWP include support for different image resolutions, where the operating system chooses the appropriate image at runtime based on the device's capabilities. Xamarin.Forms uses the native platforms' APIs for loading local images, so it automatically supports alternate resolutions if 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:

- `Button` has an `ImageSource` property that can be set to a bitmap image to be displayed on the `Button`. For more information, see [Using bitmaps with buttons](#).
- `ImageButton` has a `Source` property that can be set to the image to display in the `ImageButton`. For more

information, see [Setting the image source](#).

- `ToolbarItem` has an `IconImageSource` property that can be set to an image that's loaded from a file, embedded resource, URI, or stream.
- `ImageCell` has an `ImageSource` property that can be set to an image retrieved from a file, embedded resource, URI, or stream.
- `Page`. Any page type that derives from `Page` has `IconImageSource` and `BackgroundImageSource` properties, which can be assigned a file, embedded resource, URI, or stream. Under certain circumstances, such as when a `NavigationPage` is displaying a `ContentPage`, the icon will be displayed if supported by the platform.

IMPORTANT

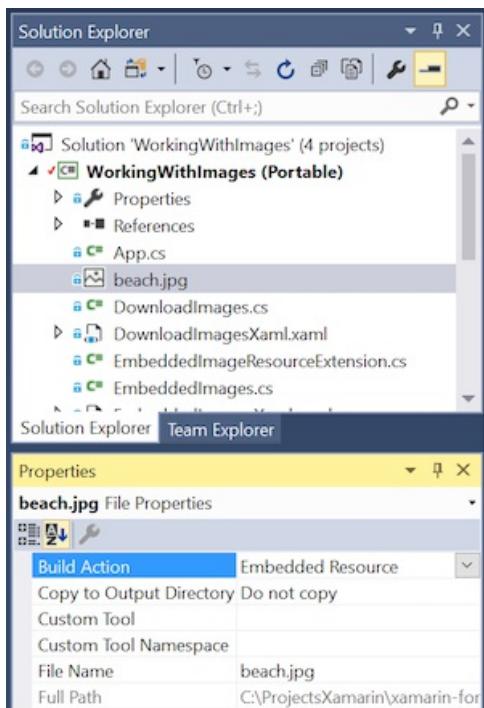
On iOS, the `Page.IconImageSource` property can't be populated from an image in an asset catalog image set. Instead, load icon images for the `Page.IconImageSource` property from a file, embedded resource, URI, or stream.

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:

```
Image embeddedImage = new Image  
{  
    Source = ImageSource.FromResource("WorkingWithImages.beach.jpg", typeof(MyClass).GetTypeInfo().Assembly)  
};
```

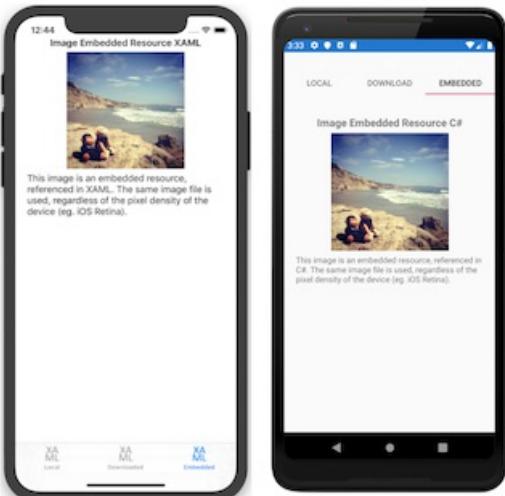
NOTE

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:



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>

```

Troubleshoot embedded images

Debug 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(MyClass).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);
```

Download images

Images can be automatically downloaded for display, as shown in the following XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="WorkingWithImages.DownloadImagesXaml">
    <StackLayout VerticalOptions="Center" HorizontalOptions="Center">
        <Label Text="Image UriSource Xaml" />
        <Image Source="https://aka.ms/campus.jpg" />
        <Label Text="campus.jpg gets downloaded from microsoft.com" />
    </StackLayout>
</ContentPage>
```

The equivalent C# code is as follows:

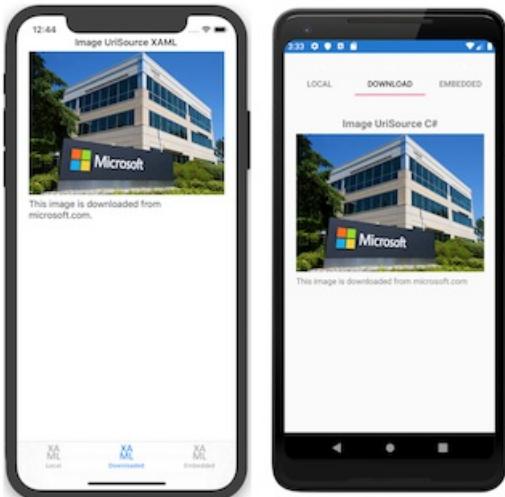
```
var webImage = new Image {
    Source = ImageSource.FromUri(
        new Uri("https://aka.ms/campus.jpg")
    );
}
```

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://aka.ms/campus.jpg";
```

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 = new Uri("https://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://aka.ms/campus.jpg"),
    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.

Animated GIFs

Xamarin.Forms includes support for displaying small, animated GIFs. This is accomplished by setting the `Image.Source` property to an animated GIF file:

```
<Image Source="demo.gif" />
```

IMPORTANT

While the animated GIF support in Xamarin.Forms includes the ability to download files, it does not support caching or streaming animated GIFs.

By default, when an animated GIF is loaded it will not be played. This is because the `IsAnimationPlaying` property, that controls whether an animated GIF is playing or stopped, has a default value of `false`. This property, of type `bool`, is backed by a `BindableProperty` object, which means that it can be the target of a data binding, and styled.

Therefore, when an animated GIF is loaded it will not be played until the `IsAnimationPlaying` property is set to `true`. Playback can then be stopped by setting the `IsAnimationPlaying` property to `false`. Note that this property has no effect when displaying a non-GIF image source.

NOTE

On Android, animated GIF support requires that your application is using fast renderers, and won't work if you've opted into using the legacy renderers. On UWP, animated GIF support requires a minimum release of Windows 10 Anniversary Update (version 1607).

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 [UWP Guidelines for tile and icon assets](#) for more information on creating these application resources.

In addition, font icons can be displayed by the [Image](#) view by specifying the font icon data in a [FontImageSource](#) object. For more information, see [Display font icons](#) in the [Fonts](#) guide.

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.

Related links

- [WorkingWithImages \(sample\)](#)
- [iOS Working with Images](#)
- [Android Iconography](#)
- [Guidelines for tile and icon assets](#)

Xamarin.Forms Label

8/4/2022 • 14 minutes to read • [Edit Online](#)

 [Download the sample](#)

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`

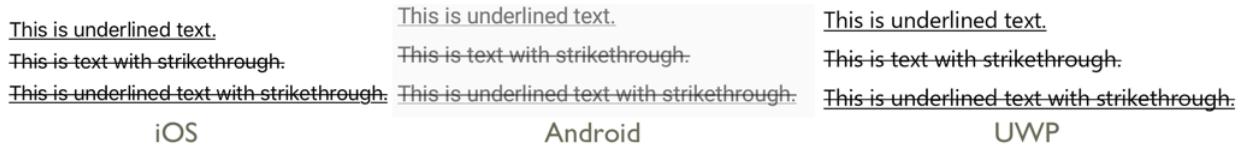
The following XAML example demonstrates setting the `Label.TextDecorations` property:

```
<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:



NOTE

Text decorations can also be applied to `Span` instances. For more information about the `Span` class, see [Formatted Text](#).

Transform text

A `Label` can transform the casing of its text, stored in the `Text` property, by setting the `TextTransform` property to a value of the `TextTransform` enumeration. This enumeration has four values:

- `None` indicates that the text won't be transformed.

- `Default` indicates that the default behavior for the platform will be used. This is the default value of the `TextTransform` property.
- `Lowercase` indicates that the text will be transformed to lowercase.
- `Uppercase` indicates that the text will be transformed to uppercase.

The following example shows transforming text to uppercase:

```
<Label Text="This text will be displayed in uppercase."
       TextTransform="Uppercase" />
```

The equivalent C# code is:

```
Label label = new Label
{
    Text = "This text will be displayed in uppercase.",
    TextTransform = TextTransform.Uppercase
};
```

Character spacing

Character spacing can be applied to `Label` instances by setting the `Label.CharacterSpacing` property to a `double` value:

```
<Label Text="Character spaced text"
       CharacterSpacing="10" />
```

The equivalent C# code is:

```
Label label = new Label { Text = "Character spaced text", CharacterSpacing = 10 };
```

The result is that characters in the text displayed by the `Label` are spaced `CharacterSpacing` device-independent units apart.

New lines

There are two main techniques for forcing text in a `Label` onto a new line, from XAML:

1. Use the unicode line feed character, which is "
".
2. Specify your text using *property element* syntax.

The following code shows an example of both techniques:

```
<!-- Unicode line feed character -->
<Label Text="First line &#10; Second line" />

<!-- Property element syntax -->
<Label>
    <Label.Text>
        First line
        Second line
    </Label.Text>
</Label>
```

In C#, text can be forced onto a new line with the "\n" character:

```
Label label = new Label { Text = "First line\nSecond line" };
```

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 replace 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.

Display 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 -1, which is its default value, 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 0, the `Label` isn't displayed.
- 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:



Display HTML

The `Label` class has a `TextType` property, which determines whether the `Label` instance should display plain text, or HTML text. This property should be set to one of the members of the `TextType` enumeration:

- `Text` indicates that the `Label` will display plain text, and is the default value of the `Label.TextType` property.
- `Html` indicates that the `Label` will display HTML text.

Therefore, `Label` instances can display HTML by setting the `Label.TextType` property to `Html`, and the `Label.Text` property to a HTML string:

```
Label label = new Label
{
    Text = "This is <strong style=\"color:red\">HTML</strong> text.",
    TextType = TextType.Html
};
```

In the example above, the double quote characters in the HTML have to be escaped using the `\` symbol.

In XAML, HTML strings can become unreadable due to additionally escaping the `<` and `>` symbols:

```
<Label Text="This is &lt;strong style="color:red"&gt;HTML&lt;/strong&gt; text."
       TextType="Html" />
```

Alternatively, for greater readability the HTML can be inlined in a `CDATA` section:

```
<Label TextType="Html">
    <![CDATA[
        This is <strong style="color:red">HTML</strong> text.
    ]]>
</Label>
```

In this example, the `Label.Text` property is set to the HTML string that's inlined in the `CDATA` section. This works because the `Text` property is the `ContentProperty` for the `Label` class.

The following screenshots show a `Label` displaying HTML:



IMPORTANT

Displaying HTML in a `Label` is limited to the HTML tags that are supported by the underlying platform.

Formatted text

Labels expose a `FormattedText` property that 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.
- `CharacterSpacing`, of type `double`, is the spacing between characters of the `Span` text.
- `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](#).

The `BackgroundColor`, `Text`, and `Text` bindable properties have a default binding mode of `OneWay`. For more information about this binding mode, see [The Default Binding Mode](#) in the [Binding Mode](#) guide.

In addition, the `GestureRecognizers` property can be used to define a collection of gesture recognizers that will respond to gestures on the `Span`.

NOTE

It's not possible to display HTML in a `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:

Three screenshots showing the result of setting `Label.LineHeight` to 1.8 on iOS, Android, and UWP. The text is wrapped and has a line height of 1.8.

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

Padding

Padding represents the space between an element and its child elements, and is used to separate the element from its own content. Padding can be applied to `Label` instances by setting the `Label.Padding` property to a `Thickness` value:

```
<Label Padding="10">
    <Label.FormattedText>
        <FormattedString>
            <Span Text="Lorem ipsum" />
            <Span Text="dolor sit amet." />
        </FormattedString>
    </Label.FormattedText>
</Label>
```

The equivalent C# code is:

```
FormattedString formattedString = new FormattedString();
formattedString.Spans.Add(new Span
{
    Text = "Lorem ipsum"
});
formattedString.Spans.Add(new Span
{
    Text = "dolor sit amet."
});
Label label = new Label
{
    FormattedText = formattedString,
    Padding = new Thickness(20)
};
```

IMPORTANT

On iOS, when a `Label` is created that sets the `Padding` property, padding will be applied and the padding value can be updated later. However, when a `Label` is created that doesn't set the `Padding` property, attempting to set it later will have no effect.

On Android and the Universal Windows Platform, the `Padding` property value can be specified when the `Label` is created, or later.

For more information about padding, see [Margins and Padding](#).

Hyperlinks

The text displayed by `Label` and `Span` instances can be turned into hyperlinks with the following approach:

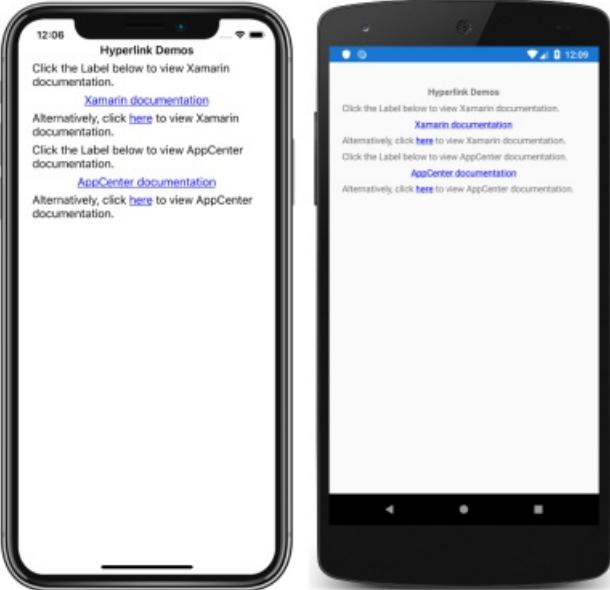
1. Set the `TextColor` and `TextDecoration` properties of the `Label` or `Span`.

2. Add a `TapGestureRecognizer` to the `GestureRecognizers` collection of the `Label` or `Span`, whose `Command` property binds to a `ICommand`, and whose `CommandParameter` property contains the URL to open.
3. Define the `ICommand` that will be executed by the `TapGestureRecognizer`.
4. Write the code that will be executed by the `ICommand`.

The following code example, taken from the [Hyperlink Demos](#) sample, shows a `Label` whose content is set from multiple `Span` instances:

```
<Label>
    <Label.FormattedText>
        <FormattedString>
            <Span Text="Alternatively, click " />
            <Span Text="here"
                  TextColor="Blue"
                  TextDecorations="Underline">
                <Span.GestureRecognizers>
                    <TapGestureRecognizer Command="{Binding TapCommand}"
                                         CommandParameter="https://docs.microsoft.com/xamarin/" />
                </Span.GestureRecognizers>
            </Span>
            <Span Text=" to view Xamarin documentation." />
        </FormattedString>
    </Label.FormattedText>
</Label>
```

In this example, the first and third `Span` instances comprise text, while the second `Span` represents a tappable hyperlink. It has its text color set to blue, and has an underline text decoration. This creates the appearance of a hyperlink, as shown in the following screenshots:



When the hyperlink is tapped, the `TapGestureRecognizer` will respond by executing the `ICommand` defined by its `Command` property. In addition, the URL specified by the `CommandParameter` property will be passed to the `ICommand` as a parameter.

The code-behind for the XAML page contains the `TapCommand` implementation:

```

public partial class MainPage : ContentPage
{
    // Launcher.OpenAsync is provided by Xamarin.Essentials.
    public ICommand TapCommand => new Command<string>(async (url) => await Launcher.OpenAsync(url));

    public MainPage()
    {
        InitializeComponent();
        BindingContext = this;
    }
}

```

The `TapCommand` executes the `Launcher.OpenAsync` method, passing the `TapGestureRecognizer.CommandParameter` property value as a parameter. The `Launcher.OpenAsync` method is provided by `Xamarin.Essentials`, and opens the URL in a web browser. Therefore, the overall effect is that when the hyperlink is tapped on the page, a web browser appears and the URL associated with the hyperlink is navigated to.

Creating a reusable hyperlink class

The previous approach to creating a hyperlink requires writing repetitive code every time you require a hyperlink in your application. However, both the `Label` and `Span` classes can be subclassed to create `HyperlinkLabel` and `HyperlinkSpan` classes, with the gesture recognizer and text formatting code added there.

The following code example, taken from the [Hyperlink Demos](#) sample, shows a `HyperlinkSpan` class:

```

public class HyperlinkSpan : Span
{
    public static readonly BindableProperty UrlProperty =
        BindableProperty.Create(nameof(Url), typeof(string), typeof(HyperlinkSpan), null);

    public string Url
    {
        get { return (string)GetValue(UrlProperty); }
        set { SetValue(UrlProperty, value); }
    }

    public HyperlinkSpan()
    {
        TextDecorations = TextDecorations.Underline;
        TextColor = Color.Blue;
        GestureRecognizers.Add(new TapGestureRecognizer
        {
            // Launcher.OpenAsync is provided by Xamarin.Essentials.
            Command = new Command(async () => await Launcher.OpenAsync(Url))
        });
    }
}

```

The `HyperlinkSpan` class defines a `url` property, and associated `BindableProperty`, and the constructor sets the hyperlink appearance and the `TapGestureRecognizer` that will respond when the hyperlink is tapped. When a `HyperlinkSpan` is tapped, the `TapGestureRecognizer` will respond by executing the `Launcher.OpenAsync` method to open the URL, specified by the `url` property, in a web browser.

The `HyperlinkSpan` class can be consumed by adding an instance of the class to the XAML:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:HyperlinkDemo"
    x:Class="HyperlinkDemo.MainPage">
    <StackLayout>
        ...
        <Label>
            <Label.FormattedText>
                <FormattedString>
                    <Span Text="Alternatively, click " />
                    <local:HyperlinkSpan Text="here"
                        Url="https://docs.microsoft.com/appcenter/" />
                    <Span Text=" to view AppCenter documentation." />
                </FormattedString>
            </Label.FormattedText>
        </Label>
    </StackLayout>
</ContentPage>
```

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 \(sample\)](#)
- [Hyperlinks \(sample\)](#)
- [Creating Mobile Apps with Xamarin.Forms, Chapter 3 free download](#)
- [Label API](#)
- [Span API](#)

Xamarin.Forms Map

8/4/2022 • 2 minutes to read • [Edit Online](#)

Initialization and Configuration

The [Xamarin.Forms.Maps](#) NuGet package is required to use maps functionality in an application. In addition, accessing the user's location requires location permissions to have been granted to the application.

Map Control

The [Map](#) control is a cross-platform view for displaying and annotating maps. It uses the native map control for each platform, providing a fast and familiar maps experience for users.

Position and Distance

The [Position](#) struct is typically used when positioning a map and its pins, and the [Distance](#) struct that can optionally be used when positioning a map.

Pins

The [Map](#) control allows locations to be marked with [Pin](#) objects. A [Pin](#) is a map marker that opens an information window when tapped.

Polygons, Polylines, and Circles

[Polygon](#), [Polyline](#), and [Circle](#) elements allow you to highlight specific areas on a map. A [Polygon](#) is a fully enclosed shape that can have a stroke and fill color. A [Polyline](#) is a line that does not fully enclose an area. A [Circle](#) highlights a circular area of the map.

Geocoding

The [Geocoder](#) class converts between string addresses and latitude and longitude coordinates that are stored in [Position](#) objects.

Launch the Native Map App

The native map app on each platform can be launched from a Xamarin.Forms application by the [Xamarin.Essentials](#) [Launcher](#) class.

Xamarin.Forms Map Initialization and Configuration

8/4/2022 • 7 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Map` control uses the native map control 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 API requirements.

Map initialization

The `Map` control is provided by the `Xamarin.Forms.Maps` NuGet package, which should be added to every project in the solution.

After installing the `Xamarin.Forms.Maps` NuGet package, it must be initialized in each platform project.

On iOS, this should occur in `AppDelegate.cs` by invoking the `Xamarin.FormsMaps.Init` method *after* the `Xamarin.Forms.Forms.Init` method:

```
Xamarin.FormsMaps.Init();
```

On Android, this should occur in `MainActivity.cs` by invoking the `Xamarin.FormsMaps.Init` method *after* the `Xamarin.Forms.Forms.Init` method:

```
Xamarin.FormsMaps.Init(this, savedInstanceState);
```

On the Universal Windows Platform (UWP), this should occur in `MainPage.xaml.cs` by invoking the `Xamarin.FormsMaps.Init` method from the `MainPage` constructor:

```
Xamarin.FormsMaps.Init("INSERT_AUTHENTICATION_TOKEN_HERE");
```

For information about the authentication token required on UWP, see [Universal Windows Platform](#).

Once the NuGet package has been added and the initialization method called inside each application, `Xamarin.Forms.Maps` APIs can be used in the shared code project.

Platform configuration

Additional configuration is required on Android and the Universal Windows Platform (UWP) before the map will display. In addition, on iOS, Android, and UWP, accessing the user's location requires location permissions to have been granted to the application.

iOS

Displaying and interacting with a map on iOS doesn't require any additional configuration. However, to access location services, you must set the following keys in `Info.plist`:

- iOS 11 and later
 - `NSLocationWhenInUseUsageDescription` – for using location services when the application is in use
 - `NSLocationAlwaysAndWhenInUseUsageDescription` – for using location services at all times
- iOS 10 and earlier

- `NSLocationWhenInUseUsageDescription` – for using location services when the application 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 application 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

A prompt is then displayed when the application attempts to access the user's location, requesting access:



Android

The configuration process for displaying and interacting with a map on Android is:

1. Get a Google Maps API key and add it to the manifest.
2. Specify the Google Play services version number in the manifest.
3. Specify the requirement for Apache HTTP Legacy library in the manifest.
4. [optional] Specify the `WRITE_EXTERNAL_STORAGE` permission in the manifest.
5. [optional] Specify location permissions in the manifest.
6. [optional] Request runtime location permissions in the `MainActivity` class.

For an example of a correctly configured manifest file, see [AndroidManifest.xml](#) from the sample application.

Get a Google Maps API key

To use the [Google Maps API](#) on Android you must generate an API key. To do this, follow the instructions in [Obtaining a Google Maps API key](#).

Once you've obtained an API key it must be added within the `<application>` element of the `Properties/AndroidManifest.xml` file:

```
<application ...>
    <meta-data android:name="com.google.android.geo.API_KEY" android:value="PASTE-YOUR-API-KEY-HERE" />
</application>
```

This embeds the API key into the manifest. Without a valid API key the `Map` control will display a blank grid.

NOTE

`com.google.android.geo.API_KEY` is the recommended metadata name for the API key. For backwards compatibility, the `com.google.android.maps.v2.API_KEY` metadata name can be used, but only allows authentication to the Android Maps API v2.

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 key](#).

Specify the Google Play services version number

Add the following declaration within the `<application>` element of `AndroidManifest.xml`:

```
<meta-data android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />
```

This embeds the version of Google Play services that the application was compiled with, into the manifest.

Specify the requirement for the Apache HTTP legacy library

If your Xamarin.Forms application targets API 28 or higher, you must add the following declaration within the `<application>` element of `AndroidManifest.xml`:

```
<uses-library android:name="org.apache.http.legacy" android:required="false" />
```

This tells the application to use the Apache Http client library, which has been removed from the `bootclasspath` in Android 9.

Specify the WRITE_EXTERNAL_STORAGE permission

If your application targets API 22 or lower, it may be necessary to add the `WRITE_EXTERNAL_STORAGE` permission to the manifest, as a child of the `<manifest>` element:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

This is not required if your application targets API 23 or greater.

Specify location permissions

If your application needs to access the user's location, you must request permission by adding the `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permissions to the manifest (or both), as a child of the `<manifest>` element:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1"
    android:versionName="1.0" package="com.companyname.myapp">
    ...
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

The `ACCESS_COARSE_LOCATION` permission allows the API to use WiFi or mobile data, or both, to determine the device's location. The `ACCESS_FINE_LOCATION` permissions allows the API to use the Global Positioning System (GPS), WiFi, or mobile data to determine a precise a location as possible.

Alternatively, these permissions can be enabled by using the manifest editor to add the following permissions:

- `AccessCoarseLocation`
- `AccessFineLocation`

These are shown in the screenshot below:



Request runtime location permissions

If your application targets API 23 or later and needs to access the user's location, it must check to see if it has the required permission at runtime, and request it if it does not have it. This can be accomplished as follows:

1. In the `MainActivity` class, add the following fields:

```
const int RequestLocationId = 0;

readonly string[] LocationPermissions =
{
    Manifest.Permission.AccessCoarseLocation,
    Manifest.Permission.AccessFineLocation
};
```

2. In the `MainActivity` class, add the following `OnStart` override:

```
protected override void OnStart()
{
    base.OnStart();

    if ((int)Build.VERSION.SdkInt >= 23)
    {
        if (CheckSelfPermission(Manifest.Permission.AccessFineLocation) != Permission.Granted)
        {
            RequestPermissions(LocationPermissions, RequestLocationId);
        }
        else
        {
            // Permissions already granted - display a message.
        }
    }
}
```

Provided that the application is targeting API 23 or greater, this code performs a runtime permission check for the `AccessFineLocation` permission. If permission has not been granted, a permission request is made by calling the `RequestPermissions` method.

3. In the `MainActivity` class, add the following `OnRequestPermissionsResult` override:

```

public override void OnRequestPermissionsResult(int requestCode, string[] permissions,
[GeneratedEnum] Permission[] grantResults)
{
    if (requestCode == RequestLocationId)
    {
        if ((grantResults.Length == 1) && (grantResults[0] == (int)Permission.Granted))
            // Permissions granted - display a message.
        else
            // Permissions denied - display a message.
    }
    else
    {
        base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
    }
}

```

This override handles the result of the permission request.

The overall effect of this code is that when the application requests the user's location, the following dialog is displayed which requests permission:



Universal Windows Platform

On UWP, your application must be authenticated before it can display a map and consume map services. To authenticate your application, you must specify a maps authentication key. For more information, see [Request a maps authentication key](#). The authentication token should then be specified in the

`FormsMaps.Init("AUTHORIZATION_TOKEN")` method call, to authenticate the application with Bing Maps.

NOTE

On UWP, to use map services such as geocoding you must also set the `MapService.ServiceToken` property to the authentication key value. This can be accomplished with the following line of code:

```
Windows.Services.Maps.MapService.ServiceToken = "INSERT_AUTH_TOKEN_HERE";
```

In addition, if your application needs to access the user's location, you must enable the location capability in the package manifest. This can be accomplished as follows:

1. In Solution Explorer, double-click `package.appxmanifest` and select the Capabilities tab.
2. In the Capabilities list, check the box for Location. This adds the `location` device capability to the package manifest file.

```
<Capabilities>
    <!-- DeviceCapability elements must follow Capability elements (if present) -->
    <DeviceCapability Name="location"/>
</Capabilities>
```

Release builds

UWP release builds use .NET native compilation to compile the application directly to native code. However, a consequence of this is that the renderer for the `Map` control on UWP may be linked out of the executable. This can be fixed by using a UWP-specific overload of the `Forms.Init` method in `App.xaml.cs`:

```
var assembliesToInclude = new [] { typeof(Xamarin.Forms.Maps.UWP.MapRenderer).GetTypeInfo().Assembly };
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

This code passes the assembly in which the `Xamarin.Forms.Maps.UWP.MapRenderer` class resides, to the `Forms.Init` method. This ensures that the assembly isn't linked out of the executable by the .NET native compilation process.

IMPORTANT

Failure to do this will result in the `Map` control not appearing when running a release build.

Related links

- [Maps Sample](#)
- [Xamarin.Forms.Maps Pins.](#)
- [Maps API](#)
- [Map Custom Renderer](#)

Xamarin.Forms Map Control

8/4/2022 • 8 minutes to read • [Edit Online](#)

 [Download the sample](#)

The `Map` control is a cross-platform view for displaying and annotating maps. It uses the native map control for each platform, providing a fast and familiar maps experience for users:



The `Map` class defines the following properties that control map appearance and behavior:

- `IsShowingUser`, of type `bool`, indicates whether the map is showing the user's current location.
- `ItemsSource`, of type `IEnumerable`, which specifies the collection of `IEnumerable` items to be displayed.
- `ItemTemplate`, of type `DataTemplate`, which specifies the `DataTemplate` to apply to each item in the collection of displayed items.
- `ItemTemplateSelector`, of type `DataTemplateSelector`, which specifies the `DataTemplateSelector` that will be used to choose a `DataTemplate` for an item at runtime.
- `HasScrollEnabled`, of type `bool`, determines whether the map is allowed to scroll.
- `HasZoomEnabled`, of type `bool`, determines whether the map is allowed to zoom.
- `MapElements`, of type `IList<MapElement>`, represents the list of elements on the map, such as polygons and polylines.
- `MapType`, of type `MapType`, indicates the display style of the map.
- `MoveToLastRegionOnLayoutChange`, of type `bool`, controls whether the displayed map region will move from its current region to its previously set region when a layout change occurs.
- `Pins`, of type `IList<Pin>`, represents the list of pins on the map.
- `TrafficEnabled`, of type `bool`, indicates whether traffic data is overlaid on the map.
- `VisibleRegion`, of type `MapSpan`, returns the currently displayed region of the map.

These properties, with the exception of the `MapElements`, `Pins`, and `VisibleRegion` properties, are backed by `BindableProperty` objects, which mean they can be targets of data bindings.

The `Map` class also defines a `MapClicked` event that's fired when the map is tapped. The `MapClickedEventArgs` object that accompanies the event has a single property named `Position`, of type `Position`. When the event is

fired, the `Position` property is set to the map location that was tapped. For information about the `Position` struct, see [Map Position and Distance](#).

For information about the `ItemsSource`, `ItemTemplate`, and `ItemTemplateSelector` properties, see [Display a pin collection](#).

Display a map

A `Map` can be displayed by adding it to a layout or page:

```
<ContentPage ...>
    <xmllns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps">
        <maps:Map x:Name="map" />
    </ContentPage>
```

NOTE

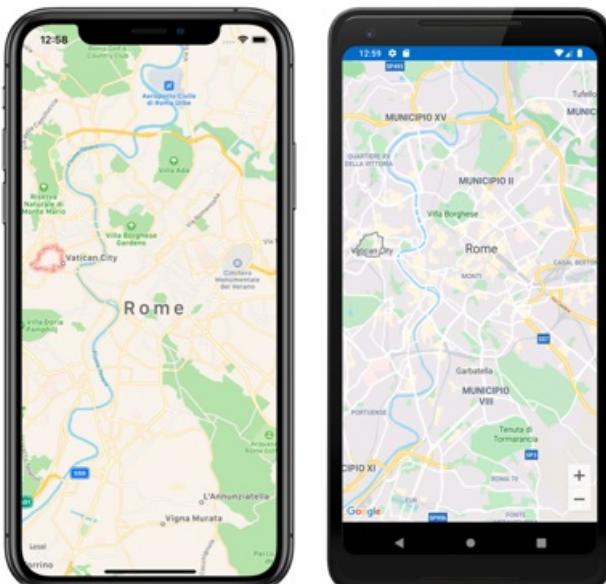
An additional `xmllns` namespace definition is required to reference the `Xamarin.Forms.Maps` controls. In the previous example the `Xamarin.Forms.Maps` namespace is referenced through the `maps` keyword.

The equivalent C# code is:

```
using Xamarin.Forms;
using Xamarin.Forms.Maps;

namespace WorkingWithMaps
{
    public class MapTypesPageCode : ContentPage
    {
        public MapTypesPageCode()
        {
            Map map = new Map();
            Content = map;
        }
    }
}
```

This example calls the default `Map` constructor, which centers the map on Rome:



Alternatively, a `MapSpan` argument can be passed to a `Map` constructor to set the center point and zoom level of the map when it's loaded. For more information, see [Display a specific location on a map](#).

Map types

The `Map.MapType` property can be set to a `MapType` enumeration member to define the display style of the map.

The `MapType` enumeration defines the following members:

- `Street` specifies that a street map will be displayed.
- `Satellite` specifies that a map containing satellite imagery will be displayed.
- `Hybrid` specifies that a map combining street and satellite data will be displayed.

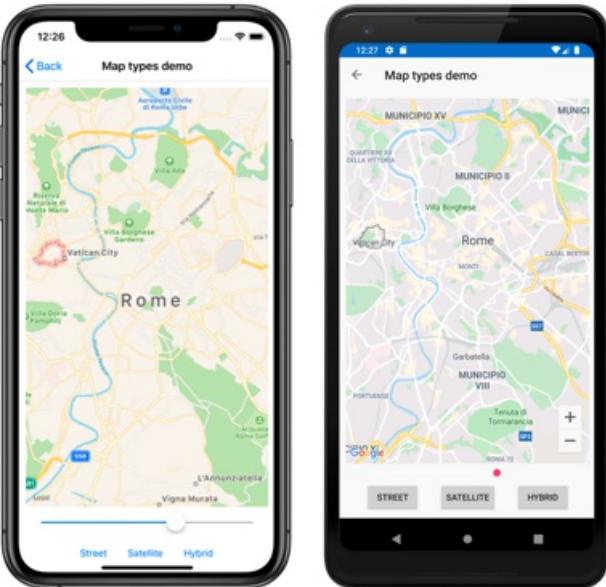
By default, a `Map` will display a street map if the `MapType` property is undefined. Alternatively, the `MapType` property can be set to one of the `MapType` enumeration members:

```
<maps:Map MapType="Satellite" />
```

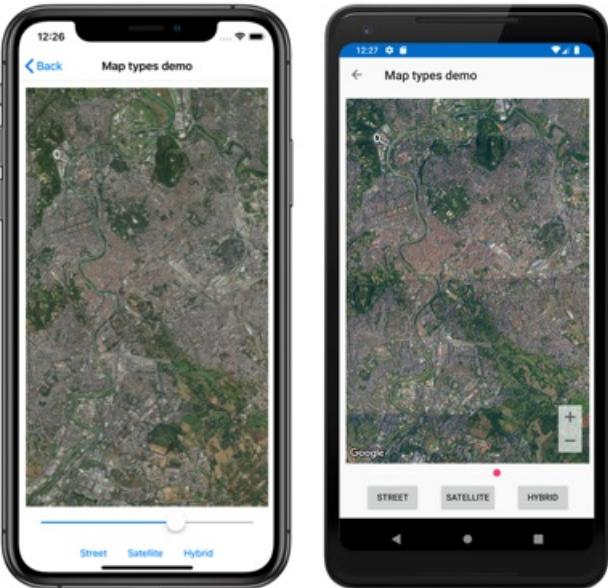
The equivalent C# code is:

```
Map map = new Map
{
    MapType = MapType.Satellite
};
```

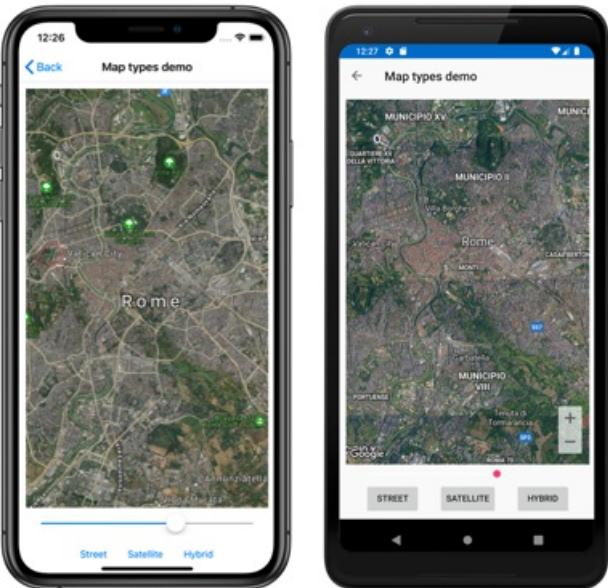
The following screenshots show a `Map` when the `MapType` property is set to `Street`:



The following screenshots show a `Map` when the `MapType` property is set to `Satellite`:



The following screenshots show a `Map` when the `MapType` property is set to `Hybrid`:



Display a specific location on a map

The region of a map to display when a map is loaded can be set by passing a `MapSpan` argument to the `Map` constructor:

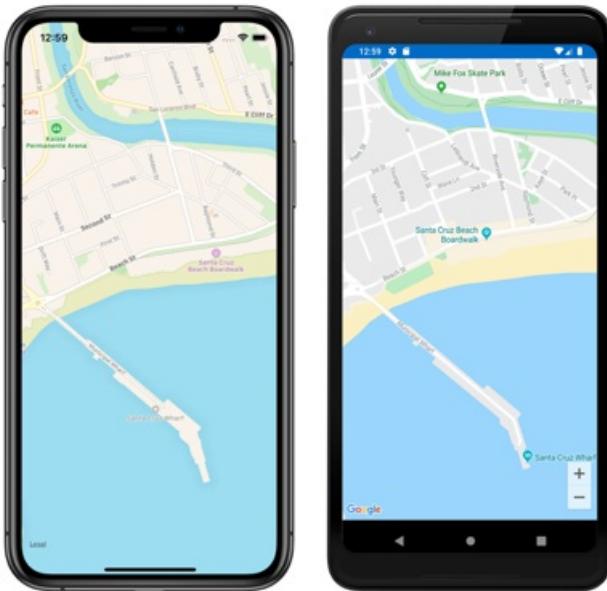
```
<maps:Map>
  <x:Arguments>
    <maps:MapSpan>
      <x:Arguments>
        <maps:Position>
          <x:Arguments>
            <x:Double>36.9628066</x:Double>
            <x:Double>-122.0194722</x:Double>
          </x:Arguments>
        </maps:Position>
        <x:Double>0.01</x:Double>
        <x:Double>0.01</x:Double>
      </x:Arguments>
    </maps:MapSpan>
  </x:Arguments>
</maps:Map>
```

The equivalent C# code is:

```
Position position = new Position(36.9628066, -122.0194722);
MapSpan mapSpan = new MapSpan(position, 0.01, 0.01);
Map map = new Map(mapSpan);
```

This example creates a `Map` object that shows the region that is specified by the `MapSpan` object. The `MapSpan` object is centered on the latitude and longitude represented by a `Position` object, and spans 0.01 latitude and 0.01 longitude degrees. For information about the `Position` struct, see [Map Position and Distance](#). For information about passing arguments in XAML, see [Passing Arguments in XAML](#).

The result is that when the map is displayed, it's centered on a specific location, and spans a specific number of latitude and longitude degrees:



Create a MapSpan object

There are a number of approaches for creating `MapSpan` objects. A common approach is supply the required arguments to the `MapSpan` constructor. These are a latitude and longitude represented by a `Position` object, and `double` values that represent the degrees of latitude and longitude that are spanned by the `MapSpan`. For information about the `Position` struct, see [Map Position and Distance](#).

Alternatively, there are three methods in the `MapSpan` class that return new `MapSpan` objects:

1. `ClampLatitude` returns a `MapSpan` with the same `LongitudeDegrees` as the method's class instance, and a radius defined by its `north` and `south` arguments.
2. `FromCenterAndRadius` returns a `MapSpan` that is defined by its `Position` and `Distance` arguments.
3. `WithZoom` returns a `MapSpan` with the same center as the method's class instance, but with a radius multiplied by its `double` argument.

For information about the `Distance` struct, see [Map Position and Distance](#).

Once a `MapSpan` has been created, the following properties can be accessed to retrieve data about it:

- `Center`, which represents the `Position` in the geographical center of the `MapSpan`.
- `LatitudeDegrees`, which represents the degrees of latitude that are spanned by the `MapSpan`.
- `LongitudeDegrees`, which represents the degrees of longitude that are spanned by the `MapSpan`.
- `Radius`, which represents the `MapSpan` radius.

Move the map

The `Map.MoveToRegion` method can be called to change the position and zoom level of a map. This method accepts a `MapSpan` argument that defines the region of the map to display, and its zoom level.

The following code shows an example of moving the displayed region on a map:

```
MapSpan mapSpan = MapSpan.FromCenterAndRadius(position, Distance.FromKilometers(0.444));
map.MoveToRegion(mapSpan);
```

Zoom the map

The zoom level of a `Map` can be changed without altering its location. This can be accomplished using the map UI, or programmatically by calling the `MoveToRegion` method with a `MapSpan` argument that uses the current location as the `Position` argument:

```
double zoomLevel = 0.5;
double latlongDegrees = 360 / (Math.Pow(2, zoomLevel));
if (map.VisibleRegion != null)
{
    map.MoveToRegion(new MapSpan(map.VisibleRegion.Center, latlongDegrees, latlongDegrees));
}
```

In this example, the `MoveToRegion` method is called with a `MapSpan` argument that specifies the current location of the map, via the `Map.VisibleRegion` property, and the zoom level as degrees of latitude and longitude. The overall result is that the zoom level of the map is changed, but its location isn't. An alternative approach for implementing zoom on a map is to use the `MapSpan.WithZoom` method to control the zoom factor.

IMPORTANT

Zooming a map, whether via the map UI or programmatically, requires that the `Map.HasZoomEnabled` property is `true`. For more information about this property, see [Disable zoom](#).

Customize map behavior

The behavior of a `Map` can be customized by setting some of its properties, and by handling the `MapClicked` event.

NOTE

Additional map behavior customization can be achieved by creating a map custom renderer. For more information, see [Customizing a Xamarin.Forms Map](#).

Show traffic data

The `Map` class defines a `TrafficEnabled` property of type `bool`. By default this property is `false`, which indicates that traffic data won't be overlaid on the map. When this property is set to `true`, traffic data is overlaid on the map. The following example shows setting this property:

```
<maps:Map TrafficEnabled="true" />
```

The equivalent C# code is:

```
Map map = new Map
{
    TrafficEnabled = true
};
```

Disable scroll

The `Map` class defines a `HasScrollEnabled` property of type `bool`. By default this property is `true`, which indicates that the map is allowed to scroll. When this property is set to `false`, the map will not scroll. The following example shows setting this property:

```
<maps:Map HasScrollEnabled="false" />
```

The equivalent C# code is:

```
Map map = new Map
{
    HasScrollEnabled = false
};
```

Disable zoom

The `Map` class defines a `HasZoomEnabled` property of type `bool`. By default this property is `true`, which indicates that zoom can be performed on the map. When this property is set to `false`, the map can't be zoomed. The following example shows setting this property:

```
<maps:Map HasZoomEnabled="false" />
```

The equivalent C# code is:

```
Map map = new Map
{
    HasZoomEnabled = false
};
```

Show the user's location

The `Map` class defines a `IsShowingUser` property of type `bool`. By default this property is `false`, which indicates that the map is not showing the user's current location. When this property is set to `true`, the map shows the user's current location. The following example shows setting this property:

```
<maps:Map IsShowingUser="true" />
```

The equivalent C# code is:

```
Map map = new Map
{
    IsShowingUser = true
};
```

IMPORTANT

On iOS, Android, and the Universal Windows Platform, accessing the user's location requires location permissions to have been granted to the application. For more information, see [Platform configuration](#).

Maintain map region on layout change

The `Map` class defines a `MoveToLastRegionOnLayoutChange` property of type `bool`. By default this property is `true`, which indicates that the displayed map region will move from its current region to its previously set region when a layout change occurs, such as on device rotation. When this property is set to `false`, the displayed map region will remain centered when a layout change occurs. The following example shows setting this property:

```
<maps:Map MoveToLastRegionOnLayoutChange="false" />
```

The equivalent C# code is:

```
Map map = new Map
{
    MoveToLastRegionOnLayoutChange = false
};
```

Map clicks

The `Map` class defines a `MapClicked` event that's fired when the map is tapped. The `MapClickedEventArgs` object that accompanies the event has a single property named `Position`, of type `Position`. When the event is fired, the `Position` property is set to the map location that was tapped. For information about the `Position` struct, see [Map Position and Distance](#).

The following code example shows an event handler for the `MapClicked` event:

```
void OnMapClicked(object sender, MapClickedEventArgs e)
{
    System.Diagnostics.Debug.WriteLine($"MapClick: {e.Position.Latitude}, {e.Position.Longitude}");
}
```

In this example, the `OnMapClicked` event handler outputs the latitude and longitude that represents the tapped map location. The event handler can be registered with the `MapClicked` event as follows:

```
<maps:Map MapClicked="OnMapClicked" />
```

The equivalent C# code is:

```
Map map = new Map();
map.MapClicked += OnMapClicked;
```

Related links

- [Maps Sample](#)
- [Map Position and Distance](#)
- [Customizing a Xamarin.Forms Map](#)
- [Passing Arguments in XAML](#)

Xamarin.Forms Map Position and Distance

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Xamarin.Forms.Maps` namespace contains a `Position` struct that's typically used when positioning a map and its pins, and a `Distance` struct that can optionally be used when positioning a map.

Position

The `Position` struct encapsulates a position stored as latitude and longitude values. This struct defines two read-only properties:

- `Latitude`, of type `double`, which represents the latitude of the position in decimal degrees.
- `Longitude`, of type `double`, which represents the longitude of the position in decimal degrees.

`Position` objects are created with the `Position` constructor, which requires latitude and longitude arguments specified as `double` values:

```
Position position = new Position(36.9628066, -122.0194722);
```

When creating a `Position` object, the latitude value will be clamped between -90.0 and 90.0, and the longitude value will be clamped between -180.0 and 180.0.

NOTE

The `GeographyUtils` class has a `ToRadians` extension method that converts a `double` value from degrees to radians, and a `ToDegrees` extension method that converts a `double` value from radians to degrees.

Distance

The `Distance` struct encapsulates a distance stored as a `double` value, which represents the distance in meters. This struct defines three read-only properties:

- `Kilometers`, of type `double`, which represents the distance in kilometers that's spanned by the `Distance`.
- `Meters`, of type `double`, which represents the distance in meters that's spanned by the `Distance`.
- `Miles`, of type `double`, which represents the distance in miles that's spanned by the `Distance`.

`Distance` objects can be created with the `Distance` constructor, which requires a meters argument specified as a `double`:

```
Distance distance = new Distance(1450.5);
```

Alternatively, `Distance` objects can be created with the `FromKilometers`, `FromMeters`, `FromMiles`, and `BetweenPositions` factory methods:

```
Distance distance1 = Distance.FromKilometers(1.45); // argument represents the number of kilometers
Distance distance2 = Distance.FromMeters(1450.5);   // argument represents the number of meters
Distance distance3 = Distance.FromMiles(0.969);     // argument represents the number of miles
Distance distance4 = Distance.BetweenPositions(position1, position2);
```

Related links

- [Maps Sample](#)

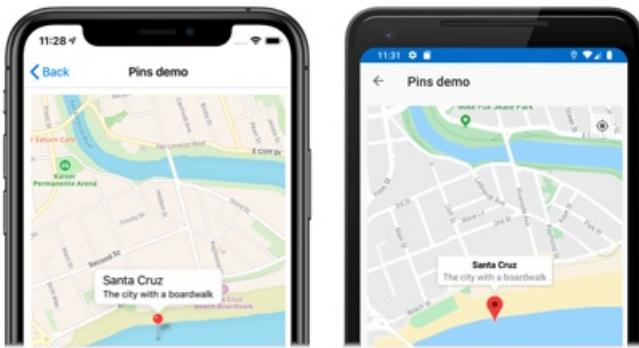
Xamarin.Forms Map Pins

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

The Xamarin.Forms `Map` control allows locations to be marked with `Pin` objects. A `Pin` is a map marker that opens an information window when tapped:



When a `Pin` object is added to the `Map.Pins` collection, the pin is rendered on the map.

The `Pin` class has the following properties:

- `Address`, of type `string`, which typically represents the address for the pin location. However, it can be any `string` content, not just an address.
- `Label`, of type `string`, which typically represents the pin title.
- `Position`, of type `Position`, which represents the latitude and longitude of the pin.
- `Type`, of type `PinType`, which represents the type of pin.

These properties are backed by `BindableProperty` objects, which means a `Pin` can be the target of data bindings. For more information about data binding `Pin` objects, see [Display a pin collection](#).

In addition, the `Pin` class defines `MarkerClicked` and `InfoWindowClicked` events. The `MarkerClicked` event is fired when a pin is tapped, and the `InfoWindowClicked` event is fired when the information window is tapped.

The `PinClickedEventArgs` object that accompanies both events has a single `HideInfoWindow` property, of type `bool`.

Display a pin

A `Pin` can be added to a `Map` in XAML:

```

<ContentPage ...>
    <x:Arguments>
        <maps:Map x:Name="map"
            IsShowingUser="True"
            MoveToLastRegionOnLayoutChange="False">
            <x:Arguments>
                <maps:MapSpan>
                    <x:Arguments>
                        <maps:Position>
                            <x:Arguments>
                                <x:Double>36.9628066</x:Double>
                                <x:Double>-122.0194722</x:Double>
                            </x:Arguments>
                        </maps:Position>
                        <x:Double>0.01</x:Double>
                        <x:Double>0.01</x:Double>
                    </x:Arguments>
                </maps:MapSpan>
            </x:Arguments>
            <maps:Map.Pins>
                <maps:Pin Label="Santa Cruz"
                    Address="The city with a boardwalk"
                    Type="Place">
                    <maps:Pin.Position>
                        <maps:Position>
                            <x:Arguments>
                                <x:Double>36.9628066</x:Double>
                                <x:Double>-122.0194722</x:Double>
                            </x:Arguments>
                        </maps:Position>
                    </maps:Pin.Position>
                </maps:Pin>
            </maps:Map.Pins>
        </maps:Map>
    </ContentPage>

```

This XAML creates a [Map](#) object that shows the region that is specified by the [MapSpan](#) object. The [MapSpan](#) object is centered on the latitude and longitude represented by a [Position](#) object, which extends 0.01 latitude and longitude degrees. A [Pin](#) object is added to the [Map.Pins](#) collection, and drawn on the [Map](#) at the location specified by its [Position](#) property. For information about the [Position](#) struct, see [Map Position and Distance](#). For information about passing arguments in XAML to objects that lack default constructors, see [Passing Arguments in XAML](#).

The equivalent C# code is:

```

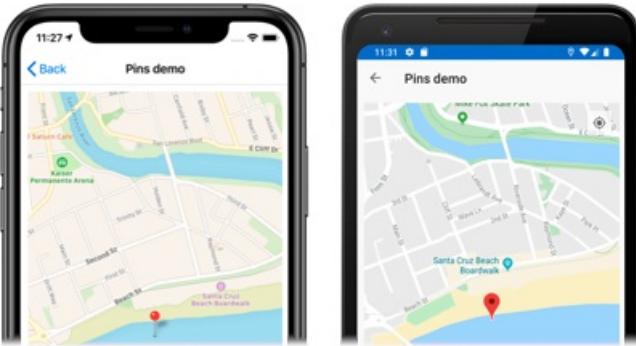
using Xamarin.Forms.Maps;
// ...
Map map = new Map
{
    // ...
};
Pin pin = new Pin
{
    Label = "Santa Cruz",
    Address = "The city with a boardwalk",
    Type = PinType.Place,
    Position = new Position(36.9628066, -122.0194722)
};
map.Pins.Add(pin);

```

WARNING

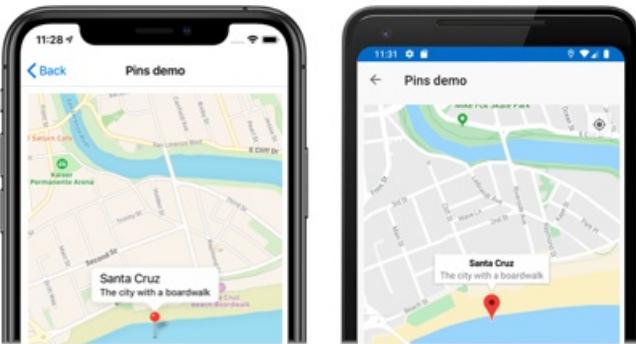
Failure to set the `Pin.Label` property will result in an `ArgumentException` being thrown when the `Pin` is added to a `Map`.

This example code results in a single pin being rendered on a map:



Interact with a pin

By default, when a `Pin` is tapped its information window is displayed:



Tapping elsewhere on the map closes the information window.

The `Pin` class defines a `MarkerClicked` event, which is fired when a `Pin` is tapped. It's not necessary to handle this event to display the information window. Instead, this event should be handled when there's a requirement to be notified that a specific pin has been tapped.

The `Pin` class also defines a `InfoWindowClicked` event that's fired when an information window is tapped. This event should be handled when there's a requirement to be notified that a specific information window has been tapped.

The following code shows an example of handling these events:

```

using Xamarin.Forms.Maps;
// ...
Pin boardwalkPin = new Pin
{
    Position = new Position(36.9641949, -122.0177232),
    Label = "Boardwalk",
    Address = "Santa Cruz",
    Type = PinType.Place
};
boardwalkPin.MarkerClicked += async (s, args) =>
{
    args.HideInfoWindow = true;
    string pinName = ((Pin)s).Label;
    await DisplayAlert("Pin Clicked", $"{pinName} was clicked.", "Ok");
};

Pin wharfPin = new Pin
{
    Position = new Position(36.9571571, -122.0173544),
    Label = "Wharf",
    Address = "Santa Cruz",
    Type = PinType.Place
};
wharfPin.InfoWindowClicked += async (s, args) =>
{
    string pinName = ((Pin)s).Label;
    await DisplayAlert("Info Window Clicked", $"The info window was clicked for {pinName}.", "Ok");
};

```

The `PinClickedEventArgs` object that accompanies both events has a single `HideInfoWindow` property, of type `bool`. When this property is set to `true` inside an event handler, the information window will be hidden.

Pin types

`Pin` objects include a `Type` property, of type `PinType`, which represents the type of pin. The `PinType` enumeration defines the following members:

- `Generic`, represents a generic pin.
- `Place`, represents a pin for a place.
- `SavedPin`, represents a pin for a saved location.
- `SearchResult`, represents a pin for a search result.

However, setting the `Pin.Type` property to any `PinType` member does not change the appearance of the rendered pin. Instead, you must create a custom renderer to customize pin appearance. For more information, see [Customizing a map pin](#).

Display a pin collection

The `Map` class defines the following properties:

- `ItemsSource`, of type `IEnumerable`, which specifies the collection of `IEnumerable` items to be displayed.
- `ItemTemplate`, of type `DataTemplate`, which specifies the `DataTemplate` to apply to each item in the collection of displayed items.
- `ItemTemplateSelector`, of type `DataTemplateSelector`, which specifies the `DataTemplateSelector` that will be used to choose a `DataTemplate` for an item at runtime.

IMPORTANT

The `ItemTemplate` property takes precedence when both the `ItemTemplate` and `ItemTemplateSelector` properties are set.

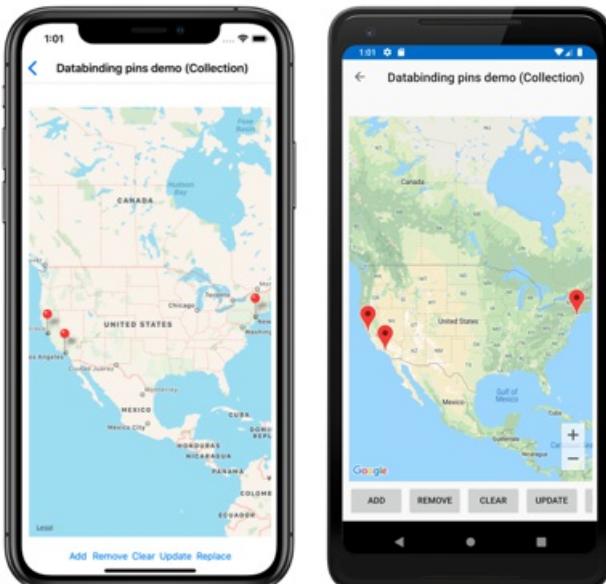
A `Map` can be populated with pins by using data binding to bind its `ItemsSource` property to an `IEnumerable` collection:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
    x:Class="WorkingWithMaps.PinItemsSourcePage">
    <Grid>
        ...
        <maps:Map x:Name="map"
            ItemsSource="{Binding Locations}">
            <maps:Map.ItemTemplate>
                <DataTemplate>
                    <maps:Pin Position="{Binding Position}"
                        Address="{Binding Address}"
                        Label="{Binding Description}" />
                </DataTemplate>
            </maps:Map.ItemTemplate>
        </maps:Map>
        ...
    </Grid>
</ContentPage>
```

The `ItemsSource` property data binds to the `Locations` property of the connected.viewmodel, which returns an `ObservableCollection` of `Location` objects, which is a custom type. Each `Location` object defines `Address` and `Description` properties, of type `string`, and a `Position` property, of type `Position`.

The appearance of each item in the `IEnumerable` collection is defined by setting the `ItemTemplate` property to a `DataTemplate` that contains a `Pin` object that data binds to appropriate properties.

The following screenshots show a `Map` displaying a `Pin` collection using data binding:



Choose item appearance at runtime

The appearance of each item in the `IEnumerable` collection can be chosen at runtime, based on the item value, by setting the `ItemTemplateSelector` property to a `DataTemplateSelector`:

```

<ContentPage ...>
    xmlns:local="clr-namespace:WorkingWithMaps"
    xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps">
<ContentPage.Resources>
    <local:MapItemTemplateSelector x:Key="MapItemTemplateSelector">
        <local:MapItemTemplateSelector.DefaultTemplate>
            <DataTemplate>
                <maps:Pin Position="{Binding Position}"
                           Address="{Binding Address}"
                           Label="{Binding Description}" />
            </DataTemplate>
        </local:MapItemTemplateSelector.DefaultTemplate>
        <local:MapItemTemplateSelector.XamarinTemplate>
            <DataTemplate>
                <!-- Change the property values, or the properties that are bound to. -->
                <maps:Pin Position="{Binding Position}"
                           Address="{Binding Address}"
                           Label="Xamarin!" />
            </DataTemplate>
        </local:MapItemTemplateSelector.XamarinTemplate>
    </local:MapItemTemplateSelector>
</ContentPage.Resources>

<Grid>
    ...
    <maps:Map x:Name="map"
              ItemsSource="{Binding Locations}"
              ItemTemplateSelector="{StaticResource MapItemTemplateSelector}" />
    ...
</Grid>
</ContentPage>

```

The following example shows the `MapItemTemplateSelector` class:

```

public class MapItemTemplateSelector : DataTemplateSelector
{
    public DataTemplate DefaultTemplate { get; set; }
    public DataTemplate XamarinTemplate { get; set; }

    protected override DataTemplate OnSelectTemplate(object item, BindableObject container)
    {
        return ((Location)item).Address.Contains("San Francisco") ? XamarinTemplate : DefaultTemplate;
    }
}

```

The `MapItemTemplateSelector` class defines `DefaultTemplate` and `XamarinTemplate` `DataTemplate` properties that are set to different data templates. The `OnSelectTemplate` method returns the `XamarinTemplate`, which displays "Xamarin" as a label when a `Pin` is tapped, when the item has an address that contains "San Francisco". When the item doesn't have an address that contains "San Francisco", the `OnSelectTemplate` method returns the `DefaultTemplate`.

NOTE

A use case for this functionality is binding properties of sub-classed `Pin` objects to different properties, based on the `Pin` sub-type.

For more information about data template selectors, see [Creating a Xamarin.Forms DataTemplateSelector](#).

Related links

- [Maps Sample](#)
- [Map Custom Renderer](#)
- [Passing Arguments in XAML](#)
- [Creating a Xamarin.Forms DataTemplateSelector](#)

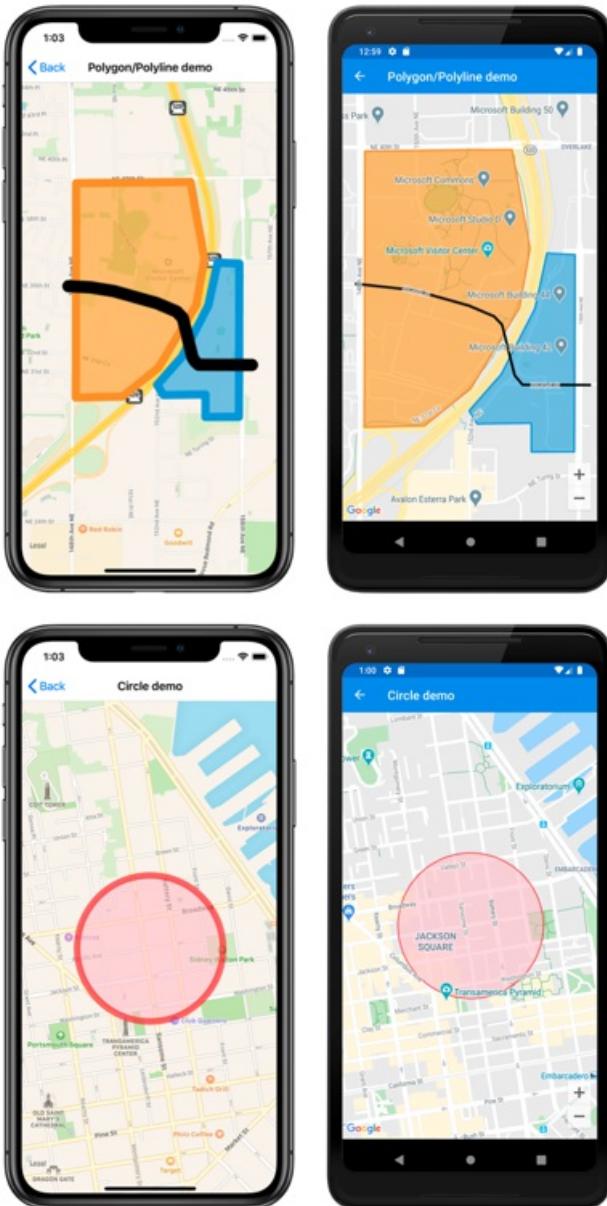
Xamarin.Forms Map Polygons and Polylines

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

`Polygon`, `Polyline`, and `Circle` elements allow you to highlight specific areas on a map. A `Polygon` is a fully enclosed shape that can have a stroke and fill color. A `Polyline` is a line that does not fully enclose an area. A `Circle` highlights a circular area of the map:



The `Polygon`, `Polyline`, and `Circle` classes derive from the `MapElement` class, which exposes the following bindable properties:

- `StrokeColor` is a `Color` object that determines the line color.
- `StrokeWidth` is a `float` object that determines the line width.

The `Polygon` class defines an additional bindable property:

- `FillColor` is a `Color` object that determines the polygon's background color.

In addition, the `Polygon` and `Polyline` classes both define a `GeoPath` property, which is a list of `Position` objects that specify the points of the shape.

The `circle` class defines the following bindable properties:

- `Center` is a `Position` object that defines the center of the circle, in latitude and longitude.
- `Radius` is a `Distance` object that defines the radius of the circle in meters, kilometers, or miles.
- `FillColor` is a `Color` property that determines the color within the circle perimeter.

NOTE

If the `StrokeColor` property is not specified the stroke will default to black. If the `FillColor` property is not specified, the fill will default to transparent. Therefore, if neither property is specified the shape will have a black outline with no fill.

Create a polygon

A `Polygon` object can be added to a map by instantiating it and adding it to the map's `MapElements` collection.

This can be accomplished in XAML as follows:

```
<ContentPage ...>
    xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps">
    <maps:Map>
        <maps:Map.MapElements>
            <maps:Polygon StrokeColor="#FF9900"
                           StrokeWidth="8"
                           FillColor="#88FF9900">
                <maps:Polygon.GeoPath>
                    <maps:Position>
                        <x:Arguments>
                            <x:Double>47.6368678</x:Double>
                            <x:Double>-122.137305</x:Double>
                        </x:Arguments>
                    </maps:Position>
                    ...
                </maps:Polygon.GeoPath>
            </maps:Polygon>
        </maps:Map.MapElements>
    </maps:Map>
</ContentPage>
```

The equivalent C# code is:

```

using Xamarin.Forms.Maps;
// ...
Map map = new Map
{
    // ...
};

// instantiate a polygon
Polygon polygon = new Polygon
{
    StrokeWidth = 8,
    StrokeColor = Color.FromHex("#1BA1E2"),
    FillColor = Color.FromHex("#881BA1E2"),
    Geopath =
    {
        new Position(47.6368678, -122.137305),
        new Position(47.6368894, -122.134655),
        new Position(47.6359424, -122.134655),
        new Position(47.6359496, -122.1325521),
        new Position(47.6424124, -122.1325199),
        new Position(47.642463, -122.1338932),
        new Position(47.6406414, -122.1344833),
        new Position(47.6384943, -122.1361248),
        new Position(47.6372943, -122.1376912)
    }
};

// add the polygon to the map's MapElements collection
map.MapElements.Add(polygon);

```

The `StrokeColor` and `StrokeWidth` properties are specified to customize the polygon's outline. The `FillColor` property value matches the `StrokeColor` property value but has an alpha value specified to make it transparent, allowing the underlying map to be visible through the shape. The `GeoPath` property contains a list of `Position` objects defining the geographic coordinates of the polygon points. A `Polygon` object is rendered on the map once it has been added to the `MapElements` collection of the `Map`.

NOTE

A `Polygon` is a fully enclosed shape. The first and last points will automatically be connected if they do not match.

Create a polyline

A `Polyline` object can be added to a map by instantiating it and adding it to the map's `MapElements` collection. This can be accomplished in XAML as follows:

```

<ContentPage ...
    xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps">
    <maps:Map>
        <maps:MapElements>
            <maps:Polyline StrokeColor="Blue"
                StrokeWidth="12">
                <maps:Polyline.Geopath>
                    <maps:Position>
                        <x:Arguments>
                            <x:Double>47.6381401</x:Double>
                            <x:Double>-122.1317367</x:Double>
                        </x:Arguments>
                    </maps:Position>
                    ...
                </maps:Polyline.Geopath>
            </maps:Polyline>
        </maps:MapElements>
    </maps:Map>
</ContentPage>

```

```

using Xamarin.Forms.Maps;
// ...
Map map = new Map
{
    // ...
};
// instantiate a polyline
Polyline polyline = new Polyline
{
    StrokeColor = Color.Blue,
    StrokeWidth = 12,
    Geopath =
    {
        new Position(47.6381401, -122.1317367),
        new Position(47.6381473, -122.1350841),
        new Position(47.6382847, -122.1353094),
        new Position(47.6384582, -122.1354703),
        new Position(47.6401136, -122.1360819),
        new Position(47.6403883, -122.1364681),
        new Position(47.6407426, -122.1377019),
        new Position(47.6412558, -122.1404056),
        new Position(47.6414148, -122.1418647),
        new Position(47.6414654, -122.1432702)
    }
};

// add the polyline to the map's MapElements collection
map.MapElements.Add(polyline);

```

The `StrokeColor` and `StrokeWidth` properties are specified to customize the line. The `GeoPath` property contains a list of `Position` objects defining the geographic coordinates of the polyline points. A `Polyline` object is rendered on the map once it has been added to the `MapElements` collection of the `Map`.

Create a circle

A `Circle` object can be added to a map by instantiating it and adding it to the map's `MapElements` collection. This can be accomplished in XAML as follows:

```

<ContentPage ...
    xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps">
    <maps:Map>
        <maps:Map.MapElements>
            <maps:Circle StrokeColor="#88FF0000"
                StrokeWidth="8"
                FillColor="#88FFC0CB">
                <maps:Circle.Center>
                    <maps:Position>
                        <x:Arguments>
                            <x:Double>37.79752</x:Double>
                            <x:Double>-122.40183</x:Double>
                        </x:Arguments>
                    </maps:Position>
                </maps:Circle.Center>
                <maps:Circle.Radius>
                    <maps:Distance>
                        <x:Arguments>
                            <x:Double>250</x:Double>
                        </x:Arguments>
                    </maps:Distance>
                </maps:Circle.Radius>
            </maps:Circle>
        </maps:Map.MapElements>
        ...
    </maps:Map>
</ContentPage>

```

The equivalent C# code is:

```

using Xamarin.Forms.Maps;
// ...
Map map = new Map();

// Instantiate a Circle
Circle circle = new Circle
{
    Center = new Position(37.79752, -122.40183),
    Radius = new Distance(250),
    StrokeColor = Color.FromHex("#88FF0000"),
    StrokeWidth = 8,
    FillColor = Color.FromHex("#88FFC0CB")
};

// Add the Circle to the map's MapElements collection
map.MapElements.Add(circle);

```

The location of the `Circle` on the Map is determined by the value of the `Center` and `Radius` properties. The `Center` property defines the center of the circle, in latitude and longitude, while the `Radius` property defines the radius of the circle in meters. The `StrokeColor` and `StrokeWidth` properties are specified to customize the circle's outline. The `FillColor` property value specifies the color within the circle perimeter. Both of the color values specify an alpha channel, allowing the underlying map to be visible through the circle. The `Circle` object is rendered on the map once it has been added to the `MapElements` collection of the `Map`.

NOTE

The `GeographyUtils` class has a `ToCircumferencePositions` extension method that converts a `Circle` object (that defines `Center` and `Radius` property values) to a list of `Position` objects that make up the latitude and longitude coordinates of the circle perimeter.

Related links

- [Maps Sample](#)

Xamarin.Forms Map Geocoding

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Xamarin.Forms.Maps` namespace provides a `Geocoder` class, which converts between string addresses and latitude and longitude coordinates that are stored in `Position` objects. For more information about the `Position` struct, see [Map Position and Distance](#).

NOTE

An alternative geocoding API is available in `Xamarin.Essentials`. The `Xamarin.Essentials.Geocoding` API offers structured address data when geocoding addresses, as opposed to the strings returned by this API. For more information, see [Xamarin.Essentials: Geocoding](#).

Geocode an address

A street address can be geocoded into latitude and longitude coordinates by creating a `Geocoder` instance and calling the `GetPositionsForAddressAsync` method on the `Geocoder` instance:

```
using Xamarin.Forms.Maps;
// ...
Geocoder geoCoder = new Geocoder();

IEnumerable<Position> approximateLocations = await geoCoder.GetPositionsForAddressAsync("Pacific Ave, San
Francisco, California");
Position position = approximateLocations.FirstOrDefault();
string coordinates = $"{position.Latitude}, {position.Longitude}";
```

The `GetPositionsForAddressAsync` method takes a `string` argument that represents the address, and asynchronously returns a collection of `Position` objects that could represent the address.

Reverse geocode an address

Latitude and longitude coordinates can be reverse geocoded into a street address by creating a `Geocoder` instance and calling the `GetAddressesForPositionAsync` method on the `Geocoder` instance:

```
using Xamarin.Forms.Maps;
// ...
Geocoder geoCoder = new Geocoder();

Position position = new Position(37.8044866, -122.4324132);
IEnumerable<string> possibleAddresses = await geoCoder.GetAddressesForPositionAsync(position);
string address = possibleAddresses.FirstOrDefault();
```

The `GetAddressesForPositionAsync` method takes a `Position` argument comprised of latitude and longitude coordinates, and asynchronously returns a collection of strings that represent the addresses near the position.

Related links

- [Maps Sample](#)

- [Xamarin.Forms Map Position and Distance](#)
- [Geocoder API](#)

Launch the Native Map App from Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The native map app on each platform can be launched from a Xamarin.Forms application by the `Xamarin.Essentials Launcher` class. This class enables an application to open another app through its custom URI scheme. The launcher functionality can be invoked with the `OpenAsync` method, passing in a `string` or `Uri` argument that represents the custom URL scheme to open. For more information about `Xamarin.Essentials`, see [Xamarin.Essentials](#).

NOTE

An alternative to using the `Xamarin.Essentials Launcher` class is to use its `Map` class. For more information, see [Xamarin.Essentials: Map](#).

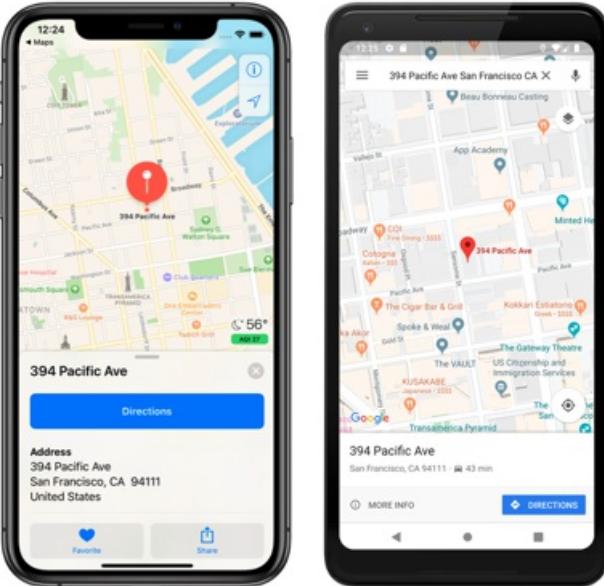
The maps app on each platform uses a unique custom URI scheme. For information about the maps URI scheme on iOS, see [Map Links](#) on developer.apple.com. For information about the maps URI scheme on Android, see [Maps Developer Guide](#) and [Google Maps Intents for Android](#) on developers.android.com. For information about the maps URI scheme on the Universal Windows Platform (UWP), see [Launch the Windows Maps app](#).

Launch the map app at a specific location

A location in the native maps app can be opened by adding appropriate query parameters to the custom URI scheme for each map app:

```
if (Device.RuntimePlatform == Device.iOS)
{
    //
    https://developer.apple.com/library/ios/featuredarticles/iPhoneURLScheme_Reference/MapLinks/MapLinks.html
    await Launcher.OpenAsync("http://maps.apple.com/?q=394+Pacific+Ave+San+Francisco+CA");
}
else if (Device.RuntimePlatform == Device.Android)
{
    // open the maps app directly
    await Launcher.OpenAsync("geo:0,0?q=394+Pacific+Ave+San+Francisco+CA");
}
else if (Device.RuntimePlatform == Device.UWP)
{
    await Launcher.OpenAsync("bingmaps:?where=394 Pacific Ave San Francisco CA");
}
```

This example code results in the native map app being launched on each platform, with the map centered on a pin representing the specified location:

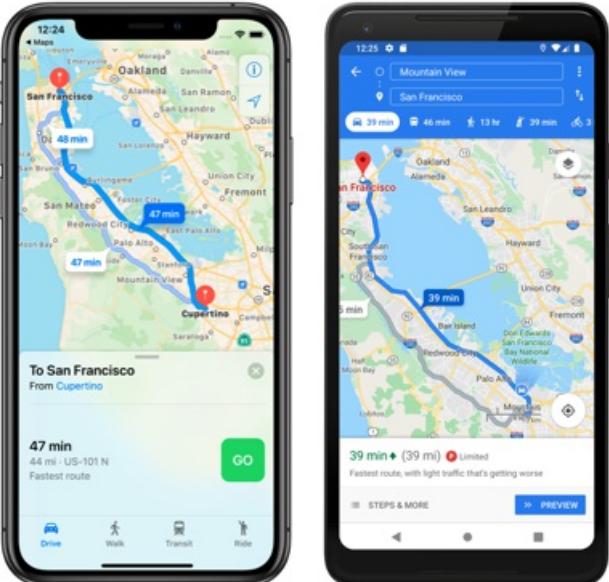


Launch the map app with directions

The native maps app can be launched displaying directions, by adding appropriate query parameters to the custom URI scheme for each map app:

```
if (Device.RuntimePlatform == Device.iOS)
{
    //
    https://developer.apple.com/library/ios/featuredarticles/iPhoneURLScheme_Reference/MapLinks/MapLinks.html
    await Launcher.OpenAsync("http://maps.apple.com/?daddr=San+Francisco,+CA&saddr=cupertino");
}
else if (Device.RuntimePlatform == Device.Android)
{
    // opens the 'task chooser' so the user can pick Maps, Chrome or other mapping app
    await Launcher.OpenAsync("http://maps.google.com/?daddr=San+Francisco,+CA&saddr=Mountain+View");
}
else if (Device.RuntimePlatform == Device.UWP)
{
    await Launcher.OpenAsync("bingmaps:?rtp=adr.394 Pacific Ave San Francisco CA~adr.One Microsoft Way
Redmond WA 98052");
}
```

This example code results in the native map app being launched on each platform, with the map centered on a route between the specified locations:



Related links

- [Maps Sample](#)
- [Xamarin.Essentials](#)
- [Map Links](#)
- [Maps Developer Guide](#)
- [Google Maps Intents for Android](#)
- [Launch the Windows Maps app](#)

Xamarin.Forms Shapes

8/4/2022 • 5 minutes to read • [Edit Online](#)

A `Shape` is a type of `View` that enables you to draw a shape to the screen. `Shape` objects can be used inside layout classes and most controls, because the `Shape` class derives from the `View` class.

Xamarin.Forms Shapes is available in the `Xamarin.Forms.Shapes` namespace on iOS, Android, macOS, the Universal Windows Platform (UWP), and the Windows Presentation Foundation (WPF).

`Shape` defines the following properties:

- `Aspect`, of type `Stretch`, describes how the shape fills its allocated space. The default value of this property is `Stretch.None`.
- `Fill`, of type `Brush`, indicates the brush used to paint the shape's interior.
- `Stroke`, of type `Brush`, indicates the brush used to paint the shape's outline.
- `StrokeDashArray`, of type `DoubleCollection`, which represents a collection of `double` values that indicate the pattern of dashes and gaps that are used to outline a shape.
- `StrokeDashOffset`, of type `double`, specifies the distance within the dash pattern where a dash begins. The default value of this property is 0.0.
- `StrokeLineCap`, of type `PenLineCap`, describes the shape at the start and end of a line or segment. The default value of this property is `PenLineCap.Flat`.
- `StrokeLineJoin`, of type `PenLineJoin`, specifies the type of join that is used at the vertices of a shape. The default value of this property is `PenLineJoin.Miter`.
- `StrokeMiterLimit`, of type `double`, specifies the limit on the ratio of the miter length to half the `StrokeThickness` of a shape. The default value of this property is 10.0.
- `StrokeThickness`, of type `double`, indicates the width of the shape outline. The default value of this property is 1.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

Xamarin.Forms defines a number of objects that derive from the `Shape` class. These are `Ellipse`, `Line`, `Path`, `Polygon`, `Polyline`, and `Rectangle`.

Paint shapes

`Brush` objects are used to paint a shapes's `Stroke` and `Fill`:

```
<Ellipse Fill="DarkBlue"
        Stroke="Red"
        StrokeThickness="4"
        WidthRequest="150"
        HeightRequest="50"
        HorizontalOptions="Start" />
```

In this example, the stroke and fill of an `Ellipse` are specified:



IMPORTANT

`Brush` objects use a type converter that enables `Color` values to be specified for the `Stroke` property.

If you don't specify a `Brush` object for `Stroke`, or if you set `StrokeThickness` to 0, then the border around the shape is not drawn.

For more information about `Brush` objects, see [Xamarin.Forms Brushes](#). For more information about valid `Color` values, see [Colors in Xamarin.Forms](#).

Stretch shapes

`Shape` objects have an `Aspect` property, of type `Stretch`. This property determines how a `Shape` object's contents is stretched to fill the `Shape` object's layout space. A `Shape` object's layout space is the amount of space the `Shape` is allocated by the Xamarin.Forms layout system, because of either an explicit `WidthRequest` and `HeightRequest` setting or because of its `HorizontalOptions` and `VerticalOptions` settings.

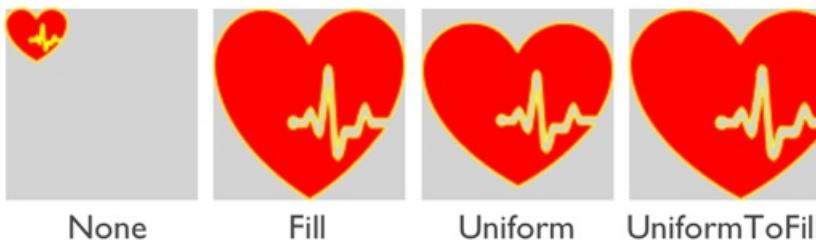
The `Stretch` enumeration defines the following members:

- `None`, which indicates that the content preserves its original size. This is the default value of the `Shape.Aspect` property.
- `Fill`, which indicates that the content is resized to fill the destination dimensions. The aspect ratio is not preserved.
- `Uniform`, which indicates that the content is resized to fit the destination dimensions, while preserving the aspect ratio.
- `UniformToFill`, indicates that the content is resized to fill the destination dimensions, while preserving the aspect ratio. If the aspect ratio of the destination rectangle differs from the source, the source content is clipped to fit in the destination dimensions.

The following XAML shows how to set the `Aspect` property:

```
<Path Aspect="Uniform"
      Stroke="Yellow"
      Fill="Red"
      BackgroundColor="LightGray"
      HorizontalOptions="Start"
      HeightRequest="100"
      WidthRequest="100">
    <Path.Data>
      <!-- Path data goes here -->
    </Path.Data>
</Path>
```

In this example, a `Path` object draws a heart. The `Path` object's `WidthRequest` and `HeightRequest` properties are set to 100 device-independent units, and its `Aspect` property is set to `Uniform`. As a result, the object's contents are resized to fit the destination dimensions, while preserving the aspect ratio:



Draw dashed shapes

`Shape` objects have a `StrokeDashArray` property, of type `DoubleCollection`. This property represents a collection of `double` values that indicate the pattern of dashes and gaps that are used to outline a shape. A `DoubleCollection` is an `ObservableCollection` of `double` values. Each `double` in the collection specifies the length of a dash or gap. The first item in the collection, which is located at index 0, specifies the length of a dash. The second item in the collection, which is located at index 1, specifies the length of a gap. Therefore, objects with an even index value specify dashes, while objects with an odd index value specify gaps.

`Shape` objects also have a `StrokeDashOffset` property, of type `double`, which specifies the distance within the dash pattern where a dash begins. Failure to set this property will result in the `Shape` having a solid outline.

Dashed shapes can be drawn by setting both the `StrokeDashArray` and `StrokeDashOffset` properties. The `StrokeDashArray` property should be set to one or more `double` values, with each pair delimited by a single comma and/or one or more spaces. For example, "0.5 1.0" and "0.5,1.0" are both valid.

The following XAML example shows how to draw a dashed rectangle:

```
<Rectangle Fill="DarkBlue"
           Stroke="Red"
           StrokeThickness="4"
           StrokeDashArray="1,1"
           StrokeDashOffset="6"
           WidthRequest="150"
           HeightRequest="50"
           HorizontalOptions="Start" />
```

In this example, a filled rectangle with a dashed stroke is drawn:



Control line ends

A line has three parts: start cap, line body, and end cap. The start and end caps describe the shape at the start and end of a line, or segment.

`Shape` objects have a `StrokeLineCap` property, of type `PenLineCap`, that describes the shape at the start and end of a line, or segment. The `PenLineCap` enumeration defines the following members:

- `Flat`, which represents a cap that doesn't extend past the last point of the line. This is comparable to no line cap, and is the default value of the `StrokeLineCap` property.
- `Square`, which represents a rectangle that has a height equal to the line thickness and a length equal to half the line thickness.
- `Round`, which represents a semicircle that has a diameter equal to the line thickness.

IMPORTANT

The `StrokeLineCap` property has no effect if you set it on a shape that has no start or end points. For example, this property has no effect if you set it on an `Ellipse`, or `Rectangle`.

The following XAML shows how to set the `StrokeLineCap` property:

```
<Line X1="0"  
      Y1="20"  
      X2="300"  
      Y2="20"  
      StrokeLineCap="Round"  
      Stroke="Red"  
      StrokeThickness="12" />
```

In this example, the red line is rounded at the start and end of the line:



Flat



Square



Round

Control line joins

`Shape` objects have a `StrokeLineJoin` property, of type `PenLineJoin`, that specifies the type of join that is used at the vertices of the shape. The `PenLineJoin` enumeration defines the following members:

- `Miter`, which represents regular angular vertices. This is the default value of the `StrokeLineJoin` property.
- `Bevel`, which represents beveled vertices.
- `Round`, which represents rounded vertices.

NOTE

When the `StrokeLineJoin` property is set to `Miter`, the `StrokeMiterLimit` property can be set to a `double` to limit the miter length of line joins in the shape.

The following XAML shows how to set the `StrokeLineJoin` property:

```
<Polyline Points="20 20,250 50,20 120"  
          Stroke="DarkBlue"  
          StrokeThickness="20"  
          StrokeLineJoin="Round" />
```

In this example, the dark blue polyline has rounded joins at its vertices:



Miter



Bevel



Round

Related links

- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Brushes](#)
- [Colors in Xamarin.Forms](#)

Xamarin.Forms Shapes: Ellipse

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Ellipse` class derives from the `Shape` class, and can be used to draw ellipses and circles. For information on the properties that the `Ellipse` class inherits from the `Shape` class, see [Xamarin.Forms Shapes](#).

The `Ellipse` class sets the `Aspect` property, inherited from the `Shape` class, to `Stretch.Fill`. For more information about the `Aspect` property, see [Stretch shapes](#).

Create an Ellipse

To draw an ellipse, create an `Ellipse` object and set its `WidthRequest` and `HeightRequest` properties. To paint the inside of the ellipse, set its `Fill` property to a `Brush`-derived object. To give the ellipse an outline, set its `Stroke` property to a `Brush`-derived object. The `StrokeThickness` property specifies the thickness of the ellipse outline. For more information about `Brush` objects, see [Xamarin.Forms Brushes](#).

To draw a circle, make the `WidthRequest` and `HeightRequest` properties of the `Ellipse` object equal.

The following XAML example shows how to draw a filled ellipse:

```
<Ellipse Fill="Red"
        WidthRequest="150"
        HeightRequest="50"
        HorizontalOptions="Start" />
```

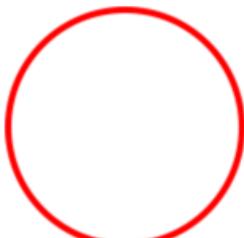
In this example, a red filled ellipse with dimensions 150x50 (device-independent units) is drawn:



The following XAML example shows how to draw a circle:

```
<Ellipse Stroke="Red"
        StrokeThickness="4"
        WidthRequest="150"
        HeightRequest="150"
        HorizontalOptions="Start" />
```

In this example, a red circle with dimensions 150x150 (device-independent units) is drawn:



For information about drawing a dashed ellipse, see [Draw dashed shapes](#).

Related links

- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Shapes](#)
- [Xamarin.Forms Brushes](#)

Xamarin.Forms Shapes: Fill rules

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

Several Xamarin.Forms Shapes classes have `FillRule` properties, of type `FillRule`. These include `Polygon`, `Polyline`, and `GeometryGroup`.

The `FillRule` enumeration defines `EvenOdd` and `Nonzero` members. Each member represents a different rule for determining whether a point is in the fill region of a shape.

IMPORTANT

All shapes are considered closed for the purposes of fill rules.

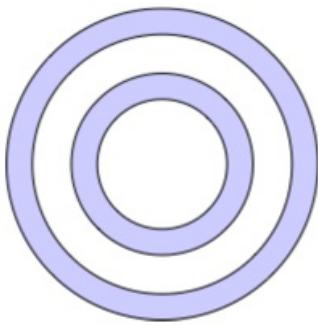
EvenOdd

The `EvenOdd` fill rule draws a ray from the point to infinity in any direction and counts the number of segments within the shape that the ray crosses. If this number is odd, the point is inside. If this number is even, the point is outside.

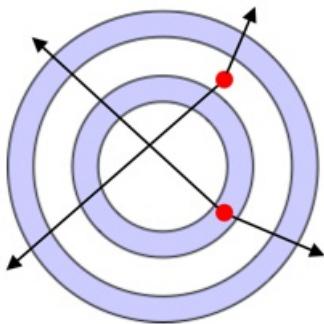
The following XAML example creates and renders a composite shape, with the `FillRule` defaulting to `EvenOdd`:

```
<Path Stroke="Black"
      Fill="#CCCCFF"
      Aspect="Uniform"
      HorizontalOptions="Start">
    <Path.Data>
        <!-- FillRule doesn't need to be set, because EvenOdd is the default. -->
        <GeometryGroup>
            <EllipseGeometry RadiusX="50"
                            RadiusY="50"
                            Center="75,75" />
            <EllipseGeometry RadiusX="70"
                            RadiusY="70"
                            Center="75,75" />
            <EllipseGeometry RadiusX="100"
                            RadiusY="100"
                            Center="75,75" />
            <EllipseGeometry RadiusX="120"
                            RadiusY="120"
                            Center="75,75" />
        </GeometryGroup>
    </Path.Data>
</Path>
```

In this example, a composite shape made up of a series of concentric rings is displayed:



In the composite shape, notice that the center and third rings are not filled. This is because a ray drawn from any point within either of those two rings passes through an even number of segments:



In the image above, the red circles represent points, and the lines represent arbitrary rays. For the upper point, the two arbitrary rays each pass through an even number of line segments. Therefore, the ring the point is in isn't filled. For the lower point, the two arbitrary rays each pass through an odd number of line segments. Therefore, the ring the point is in is filled.

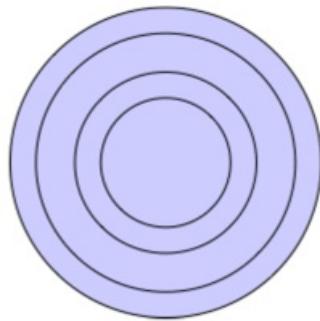
Nonzero

The `Nonzero` fill rule draws a ray from the point to infinity in any direction and then examines the places where a segment of the shape crosses the ray. Starting with a count of zero, the count is incremented each time a segment crosses the ray from left to right and decremented each time a segment crosses the ray from right to left. After counting the crossings, if the result is zero then the point is outside the polygon. Otherwise, it's inside.

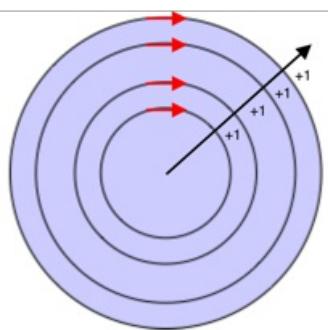
The following XAML example creates and renders a composite shape, with the `FillRule` set to `Nonzero`:

```
<Path Stroke="Black"
      Fill="#CCCCFF"
      Aspect="Uniform"
      HorizontalOptions="Start">
    <Path.Data>
      <GeometryGroup FillRule="Nonzero">
        <EllipseGeometry RadiusX="50"
                          RadiusY="50"
                          Center="75,75" />
        <EllipseGeometry RadiusX="70"
                          RadiusY="70"
                          Center="75,75" />
        <EllipseGeometry RadiusX="100"
                          RadiusY="100"
                          Center="75,75" />
        <EllipseGeometry RadiusX="120"
                          RadiusY="120"
                          Center="75,75" />
      </GeometryGroup>
    </Path.Data>
  </Path>
```

In this example, a composite shape made up of a series of concentric rings is displayed:



In the composite shape, notice that all rings are filled. This is because all the segments are running in the same direction, and so a ray drawn from any point will cross one or more segments and the sum of the crossings will not equal zero:



In the image above the red arrows represent the direction the segments are drawn, and black arrow represents an arbitrary ray running from a point in the innermost ring. Starting with a value of zero, for each segment that the ray crosses, a value of one is added because the segment crosses the ray from left to right.

A more complex shape with segments running in different directions is required to better demonstrate the behavior of the `Nonzero` fill rule. The following XAML example creates a similar shape to the previous example, except that it's created with a `PathGeometry` rather than an `EllipseGeometry`:

```

<Path Stroke="Black"
      Fill="#CCCCFF">
  <Path.Data>
    <GeometryGroup FillRule="Nonzero">
      <PathGeometry>
        <PathGeometry.Figures>
          <!-- Inner ring -->
          <PathFigure StartPoint="120,120">
            <PathFigure.Segments>
              <PathSegmentCollection>
                <ArcSegment Size="50,50"
                            IsLargeArc="True"
                            SweepDirection="CounterClockwise"
                            Point="140,120" />
              </PathSegmentCollection>
            </PathFigure.Segments>
          </PathFigure>

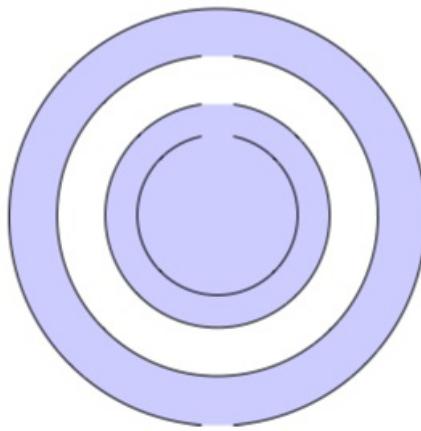
          <!-- Second ring -->
          <PathFigure StartPoint="120,100">
            <PathFigure.Segments>
              <PathSegmentCollection>
                <ArcSegment Size="70,70"
                            IsLargeArc="True"
                            SweepDirection="CounterClockwise"
                            Point="140,100" />
              </PathSegmentCollection>
            </PathFigure.Segments>
          </PathFigure>

          <!-- Third ring -->
          <PathFigure StartPoint="120,70">
            <PathFigure.Segments>
              <PathSegmentCollection>
                <ArcSegment Size="100,100"
                            IsLargeArc="True"
                            SweepDirection="CounterClockwise"
                            Point="140,70" />
              </PathSegmentCollection>
            </PathFigure.Segments>
          </PathFigure>

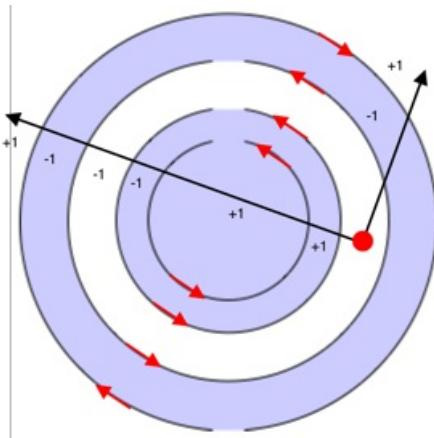
          <!-- Outer ring -->
          <PathFigure StartPoint="120,300">
            <PathFigure.Segments>
              <ArcSegment Size="130,130"
                            IsLargeArc="True"
                            SweepDirection="Clockwise"
                            Point="140,300" />
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </GeometryGroup>
  <Path.Data>
</Path>

```

In this example, a series of arc segments are drawn, that aren't closed:



In the image above, the third arc from the center is not filled. This is because the sum of the values from a given ray crossing the segments in its path is zero:



In the image above, the red circle represents a point, the black lines represent arbitrary rays that move out from the point in the non-filled region, and the red arrows represent the direction the segments are drawn. As can be seen, the sum of the values from the rays crossing the segments is zero:

- The arbitrary ray that travels diagonally right crosses two segments that run in different directions. Therefore, the segments cancel each other out giving a value of zero.
- The arbitrary ray that travels diagonally left crosses a total of six segments. However, the crossings cancel each other out so that zero is the final sum.

A sum of zero results in the ring not being filled.

Related links

- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Shapes](#)

Xamarin.Forms Shapes: Geometries

8/4/2022 • 17 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Geometry` class, and the classes that derive from it, enable you to describe the geometry of a 2D shape. `Geometry` objects can be simple, such as rectangles and circles, or composite, created from two or more geometry objects. In addition, more complex geometries can be created that include arcs and curves.

The `Geometry` class is the parent class for several classes that define different categories of geometries:

- `EllipseGeometry`, which represents the geometry of an ellipse or circle.
- `GeometryGroup`, which represents a container that can combine multiple geometry objects into a single object.
- `LineGeometry`, which represents the geometry of a line.
- `PathGeometry`, which represents the geometry of a complex shape that can be composed of arcs, curves, ellipses, lines, and rectangles.
- `RectangleGeometry`, which represents the geometry of a rectangle or square.

NOTE

There's also a `RoundedRectangleGeometry` class that derives from the `GeometryGroup` class. For more information, see [RoundRectangleGeometry](#).

The `Geometry` and `Shape` classes seem similar, in that they both describe 2D shapes, but have an important difference. The `Geometry` class derives from the `BindableObject` class, while the `Shape` class derives from the `View` class. Therefore, `Shape` objects can render themselves and participate in the layout system, while `Geometry` objects cannot. While `Shape` objects are more readily usable than `Geometry` objects, `Geometry` objects are more versatile. While a `Shape` object is used to render 2D graphics, a `Geometry` object can be used to define the geometric region for 2D graphics, and define a region for clipping.

The following classes have properties that can be set to `Geometry` objects:

- The `Path` class uses a `Geometry` to describe its contents. You can render a `Geometry` by setting the `Path.Data` property to a `Geometry` object, and setting the `Path` object's `Fill` and `Stroke` properties.
- The `VisualElement` class has a `Clip` property, of type `Geometry`, that defines the outline of the contents of an element. When the `Clip` property is set to a `Geometry` object, only the area that is within the region of the `Geometry` will be visible. For more information, see [Clip with a Geometry](#).

The classes that derive from the `Geometry` class can be grouped into three categories: simple geometries, path geometries, and composite geometries.

Simple geometries

The simple geometry classes are `EllipseGeometry`, `LineGeometry`, and `RectangleGeometry`. They are used to create basic geometric shapes, such as circles, lines, and rectangles. These same shapes, as well as more complex shapes, can be created using a `PathGeometry` or by combining geometry objects together, but these classes provide a simpler approach for producing these basic geometric shapes.

EllipseGeometry

An ellipse geometry represents the geometry of an ellipse or circle, and is defined by a center point, an x-radius, and a y-radius.

The `EllipseGeometry` class defines the following properties:

- `Center`, of type `Point`, which represents the center point of the geometry.
- `RadiusX`, of type `double`, which represents the x-radius value of the geometry. The default value of this property is 0.0.
- `RadiusY`, of type `double`, which represents the y-radius value of the geometry. The default value of this property is 0.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The following example shows how to create and render an `EllipseGeometry` in a `Path` object:

```
<Path Fill="Blue"
      Stroke="Red">
  <Path.Data>
    <EllipseGeometry Center="50,50"
                      RadiusX="50"
                      RadiusY="50" />
  </Path.Data>
</Path>
```

In this example, the center of the `EllipseGeometry` is set to (50,50) and the x-radius and y-radius are both set to 50. This creates a red circle with a diameter of 100 device-independent units, whose interior is painted blue:



LineGeometry

A line geometry represents the geometry of a line, and is defined by specifying the start point of the line and the end point.

The `LineGeometry` class defines the following properties:

- `StartPoint`, of type `Point`, which represents the start point of the line.
- `EndPoint`, of type `Point`, which represents the end point of the line.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The following example shows how to create and render a `LineGeometry` in a `Path` object:

```
<Path Stroke="Black">
  <Path.Data>
    <LineGeometry StartPoint="10,20"
                  EndPoint="100,130" />
  </Path.Data>
</Path>
```

In this example, a `LineGeometry` is drawn from (10,20) to (100,130):

NOTE

Setting the `Fill` property of a `Path` that renders a `LineGeometry` will have no effect, because a line has no interior.

RectangleGeometry

A rectangle geometry represents the geometry of a rectangle or square, and is defined with a `Rect` structure that specifies its relative position and its height and width.

The `RectangleGeometry` class defines the `Rect` property, of type `Rect`, which represents the dimensions of the rectangle. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

The following example shows how to create and render a `RectangleGeometry` in a `Path` object:

```
<Path Fill="Blue"  
      Stroke="Red">  
  <Path.Data>  
    <RectangleGeometry Rect="10,10,150,100" />  
  </Path.Data>  
</Path>
```

The position and dimensions of the rectangle are defined by a `Rect` structure. In this example, the position is (10,10), the width is 150, and the height is 100 device-independent units:



Path geometries

A path geometry describes a complex shape that can be composed of arcs, curves, ellipses, lines, and rectangles.

The `PathGeometry` class defines the following properties:

- `Figures`, of type `PathFigureCollection`, which represents the collection of `PathFigure` objects that describe the path's contents.
- `FillRule`, of type `FillRule`, which determines how the intersecting areas contained in the geometry are combined. The default value of this property is `FillRule.EvenOdd`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

For more information about the `FillRule` enumeration, see [Xamarin.Forms Shapes: Fill rules](#).

NOTE

The `Figures` property is the `ContentProperty` of the `PathGeometry` class, and so does not need to be explicitly set from XAML.

A `PathGeometry` is made up of a collection of `PathFigure` objects, with each `PathFigure` describing a shape in the geometry. Each `PathFigure` is itself comprised of one or more `PathSegment` objects, each of which describes a segment of the shape. There are many types of segments:

- `ArcSegment`, which creates an elliptical arc between two points.
- `BezierSegment`, which creates a cubic Bezier curve between two points.
- `LineSegment`, which creates a line between two points.
- `PolyBezierSegment`, which creates a series of cubic Bezier curves.
- `PolyLineSegment`, which creates a series of lines.
- `PolyQuadraticBezierSegment`, which creates a series of quadratic Bezier curves.
- `QuadraticBezierSegment`, which creates a quadratic Bezier curve.

All the above classes derive from the abstract `PathSegment` class.

The segments within a `PathFigure` are combined into a single geometric shape with the end point of each segment being the start point of the next segment. The `StartPoint` property of a `PathFigure` specifies the point from which the first segment is drawn. Each subsequent segment starts at the end point of the previous segment. For example, a vertical line from `10,50` to `10,150` can be defined by setting the `StartPoint` property to `10,50` and creating a `LineSegment` with a `Point` property setting of `10,150`:

```
<Path Stroke="Black">
    <Path.Data>
        <PathGeometry>
            <PathGeometry.Figures>
                <PathFigureCollection>
                    <PathFigure StartPoint="10,50">
                        <PathFigure.Segments>
                            <PathSegmentCollection>
                                <LineSegment Point="10,150" />
                            </PathSegmentCollection>
                        </PathFigure.Segments>
                    </PathFigure>
                </PathFigureCollection>
            </PathGeometry.Figures>
        </PathGeometry>
    </Path.Data>
</Path>
```

More complex geometries can be created by using a combination of `PathSegment` objects, and by using multiple `PathFigure` objects within a `PathGeometry`.

Create an ArcSegment

An `ArcSegment` creates an elliptical arc between two points. An elliptical arc is defined by its start and end points, x- and y-radius, x-axis rotation factor, a value indicating whether the arc should be greater than 180 degrees, and a value describing the direction in which the arc is drawn.

The `ArcSegment` class defines the following properties:

- `Point`, of type `Point`, which represents the endpoint of the elliptical arc. The default value of this property is `(0,0)`.

- `Size`, of type `Size`, which represents the x- and y-radius of the arc. The default value of this property is `(0,0)`.
- `RotationAngle`, of type `double`, which represents the amount in degrees by which the ellipse is rotated around the x-axis. The default value of this property is `0`.
- `SweepDirection`, of type `SweepDirection`, which specifies the direction in which the arc is drawn. The default value of this property is `SweepDirection.CounterClockwise`.
- `IsLargeArc`, of type `bool`, which indicates whether the arc should be greater than 180 degrees. The default value of this property is `false`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

NOTE

The `ArcSegment` class does not contain a property for the starting point of the arc. It only defines the end point of the arc it represents. The start point of the arc is the current point of the `PathFigure` to which the `ArcSegment` is added.

The `SweepDirection` enumeration defines the following members:

- `CounterClockwise`, which specifies that arcs are drawn in a clockwise direction.
- `Clockwise`, which specifies that arcs are drawn in a counter clockwise direction.

The following example shows how to create and render an `ArcSegment` in a `Path` object:

```
<Path Stroke="Black">
    <Path.Data>
        <PathGeometry>
            <PathGeometry.Figures>
                <PathFigureCollection>
                    <PathFigure StartPoint="10,10">
                        <PathFigure.Segments>
                            <PathSegmentCollection>
                                <ArcSegment Size="100,50"
                                           RotationAngle="45"
                                           IsLargeArc="True"
                                           SweepDirection="CounterClockwise"
                                           Point="200,100" />
                            </PathSegmentCollection>
                        </PathFigure.Segments>
                    </PathFigure>
                </PathFigureCollection>
            </PathGeometry.Figures>
        </PathGeometry>
    </Path.Data>
</Path>
```

In this example, an elliptical arc is drawn from `(10,10)` to `(200,100)`.

Create a BezierSegment

A `BezierSegment` creates a cubic Bezier curve between two points. A cubic Bezier curve is defined by four points: a start point, an end point, and two control points.

The `BezierSegment` class defines the following properties:

- `Point1`, of type `Point`, which represents the first control point of the curve. The default value of this property is `(0,0)`.
- `Point2`, of type `Point`, which represents the second control point of the curve. The default value of this

property is (0,0).

- `Point3`, of type `Point`, which represents the end point of the curve. The default value of this property is (0,0).

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

NOTE

The `BezierSegment` class does not contain a property for the starting point of the curve. The start point of the curve is the current point of the `PathFigure` to which the `BezierSegment` is added.

The two control points of a cubic Bezier curve behave like magnets, attracting portions of what would otherwise be a straight line toward themselves and producing a curve. The first control point affects the start portion of the curve. The second control point affects the end portion of the curve. The curve doesn't necessarily pass through either of the control points. Instead, each control point moves its portion of the line toward itself, but not through itself.

The following example shows how to create and render a `BezierSegment` in a `Path` object:

```
<Path Stroke="Black">
    <Path.Data>
        <PathGeometry>
            <PathGeometry.Figures>
                <PathFigureCollection>
                    <PathFigure StartPoint="10,10">
                        <PathFigure.Segments>
                            <PathSegmentCollection>
                                <BezierSegment Point1="100,0"
                                               Point2="200,200"
                                               Point3="300,10" />
                            </PathSegmentCollection>
                        </PathFigure.Segments>
                    </PathFigure>
                </PathFigureCollection>
            </PathGeometry.Figures>
        </PathGeometry>
    </Path.Data>
</Path>
```

In this example, a cubic Bezier curve is drawn from (10,10) to (300,10). The curve has two control points at (100,0) and (200,200):



Create a LineSegment

A `LineSegment` creates a line between two points.

The `LineSegment` class defines the `Point` property, of type `Point`, which represents the end point of the line segment. The default value of this property is (0,0), and it's backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

NOTE

The `LineSegment` class does not contain a property for the starting point of the line. It only defines the end point. The start point of the line is the current point of the `PathFigure` to which the `LineSegment` is added.

The following example shows how to create and render `LineSegment` objects in a `Path` object:

```
<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Start">
    <Path.Data>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigureCollection>
            <PathFigure IsClosed="True"
                       StartPoint="10,100">
              <PathFigure.Segments>
                <PathSegmentCollection>
                  <LineSegment Point="100,100" />
                  <LineSegment Point="100,50" />
                </PathSegmentCollection>
              </PathFigure.Segments>
            </PathFigure>
          </PathFigureCollection>
        </PathGeometry.Figures>
      </PathGeometry>
    </Path.Data>
  </Path>
```

In this example, a line segment is drawn from (10,100) to (100,100), and from (100,100) to (100,50). In addition, the `PathFigure` is closed because its `IsClosed` property is set to `true`. This results in a triangle being drawn:



Create a PolyBezierSegment

A `PolyBezierSegment` creates one or more cubic Bezier curves.

The `PolyBezierSegment` class defines the `Points` property, of type `PointCollection`, which represents the points that define the `PolyBezierSegment`. A `PointCollection` is an `ObservableCollection` of `Point` objects. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

NOTE

The `PolyBezierSegment` class does not contain a property for the starting point of the curve. The start point of the curve is the current point of the `PathFigure` to which the `PolyBezierSegment` is added.

The following example shows how to create and render a `PolyBezierSegment` in a `Path` object:

```

<Path Stroke="Black">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigureCollection>
          <PathFigure StartPoint="10,10">
            <PathFigure.Segments>
              <PathSegmentCollection>
                <PolyBezierSegment Points="0,0 100,0 150,100 150,0 200,0 300,10" />
              </PathSegmentCollection>
            </PathFigure.Segments>
          </PathFigure>
        </PathFigureCollection>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>

```

In this example, the `PolyBezierSegment` specifies two cubic Bezier curves. The first curve is from (10,10) to (150,100) with a control point of (0,0), and another control point of (100,0). The second curve is from (150,100) to (300,10) with a control point of (150,0) and another control point of (200,0):



Create a PolyLineSegment

A `PolyLineSegment` creates one or more line segments.

The `PolyLineSegment` class defines the `Points` property, of type `PointCollection`, which represents the points that define the `PolyLineSegment`. A `PointCollection` is an `ObservableCollection` of `Point` objects. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

NOTE

The `PolyLineSegment` class does not contain a property for the starting point of the line. The start point of the line is the current point of the `PathFigure` to which the `PolyLineSegment` is added.

The following example shows how to create and render a `PolyLineSegment` in a `Path` object:

```

<Path Stroke="Black">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="10,10">
          <PathFigure.Segments>
            <PolyLineSegment Points="50,10 50,50" />
          </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>

```

In this example, the `PolyLineSegment` specifies two lines. The first line is from (10,10) to (50,10), and the second line is from (50,10) to (50,50):

Create a PolyQuadraticBezierSegment

A `PolyQuadraticBezierSegment` creates one or more quadratic Bezier curves.

The `PolyQuadraticBezierSegment` class defines the `Points` property, of type `PointCollection`, which represents the points that define the `PolyQuadraticBezierSegment`. A `PointCollection` is an `ObservableCollection` of `Point` objects. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

NOTE

The `PolyQuadraticBezierSegment` class does not contain a property for the starting point of the curve. The start point of the curve is the current point of the `PathFigure` to which the `PolyQuadraticBezierSegment` is added.

The following example shows to create and render a `PolyQuadraticBezierSegment` in a `Path` object:

```
<Path Stroke="Black">
    <Path.Data>
        <PathGeometry>
            <PathGeometry.Figures>
                <PathFigureCollection>
                    <PathFigure StartPoint="10,10">
                        <PathFigure.Segments>
                            <PathSegmentCollection>
                                <PolyQuadraticBezierSegment Points="100,100 150,50 0,100 15,200" />
                            </PathSegmentCollection>
                        </PathFigure.Segments>
                    </PathFigure>
                </PathFigureCollection>
            </PathGeometry.Figures>
        </PathGeometry>
    </Path.Data>
</Path>
```

In this example, the `PolyQuadraticBezierSegment` specifies two Bezier curves. The first curve is from (10,10) to (150,50) with a control point at (100,100). The second curve is from (100,100) to (15,200) with a control point at (0,100):



Create a QuadraticBezierSegment

A `QuadraticBezierSegment` creates a quadratic Bezier curve between two points.

The `QuadraticBezierSegment` class defines the following properties:

- `Point1`, of type `Point`, which represents the control point of the curve. The default value of this property is (0,0).

- `Point2`, of type `Point`, which represents the end point of the curve. The default value of this property is `(0,0)`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

NOTE

The `QuadraticBezierSegment` class does not contain a property for the starting point of the curve. The start point of the curve is the current point of the `PathFigure` to which the `QuadraticBezierSegment` is added.

The following example shows how to create and render a `QuadraticBezierSegment` in a `Path` object:

```
<Path Stroke="Black">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigureCollection>
          <PathFigure StartPoint="10,10">
            <PathFigure.Segments>
              <PathSegmentCollection>
                <QuadraticBezierSegment Point1="200,200"
                                         Point2="300,10" />
              </PathSegmentCollection>
            </PathFigure.Segments>
          </PathFigure>
        </PathFigureCollection>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

In this example, a quadratic Bezier curve is drawn from `(10,10)` to `(300,10)`. The curve has a control point at `(200,200)`:



Create complex geometries

More complex geometries can be created by using a combination of `PathSegment` objects. The following example creates a shape using a `BezierSegment`, a `LineSegment`, and an `ArcSegment`:

```

<Path Stroke="Black">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="10,50">
          <PathFigure.Segments>
            <BezierSegment Point1="100,0"
                           Point2="200,200"
                           Point3="300,100"/>
            <LineSegment Point="400,100" />
            <ArcSegment Size="50,50"
                        RotationAngle="45"
                        IsLargeArc="True"
                        SweepDirection="Clockwise"
                        Point="200,100"/>
          </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>

```

In this example, a `BezierSegment` is first defined using four points. The example then adds a `LineSegment`, which is drawn between the end point of the `BezierSegment` to the point specified by the `LineSegment`. Finally, an `ArcSegment` is drawn from the end point of the `LineSegment` to the point specified by the `ArcSegment`.

Even more complex geometries can be created by using multiple `PathFigure` objects within a `PathGeometry`. The following example creates a `PathGeometry` from seven `PathFigure` objects, some of which contain multiple `PathSegment` objects:

```

<Path Stroke="Red"
      StrokeThickness="12"
      StrokeLineJoin="Round">
  <Path.Data>
    <PathGeometry>
      <!-- H -->
      <PathFigure StartPoint="0,0">
        <LineSegment Point="0,100" />
      </PathFigure>
      <PathFigure StartPoint="0,50">
        <LineSegment Point="50,50" />
      </PathFigure>
      <PathFigure StartPoint="50,0">
        <LineSegment Point="50,100" />
      </PathFigure>

      <!-- E -->
      <PathFigure StartPoint="125, 0">
        <BezierSegment Point1="60, -10"
                      Point2="60, 60"
                      Point3="125, 50" />
        <BezierSegment Point1="60, 40"
                      Point2="60, 110"
                      Point3="125, 100" />
      </PathFigure>

      <!-- L -->
      <PathFigure StartPoint="150, 0">
        <LineSegment Point="150, 100" />
        <LineSegment Point="200, 100" />
      </PathFigure>

      <!-- L -->
      <PathFigure StartPoint="225, 0">
        <LineSegment Point="225, 100" />
        <LineSegment Point="275, 100" />
      </PathFigure>

      <!-- O -->
      <PathFigure StartPoint="300, 50">
        <ArcSegment Size="25, 50"
                    Point="300, 49.9"
                    IsLargeArc="True" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>

```

In this example, the word "Hello" is drawn using a combination of `LineSegment` and `BezierSegment` objects, along with a single `ArcSegment` object:



Composite geometries

Composite geometry objects can be created using a `GeometryGroup`. The `GeometryGroup` class creates a composite geometry from one or more `Geometry` objects. Any number of `Geometry` objects can be added to a `GeometryGroup`.

The `GeometryGroup` class defines the following properties:

- `Children`, of type `GeometryCollection`, which specifies the objects that define the `GeometryGroup`. A `GeometryCollection` is an `ObservableCollection` of `Geometry` objects.
- `FillRule`, of type `FillRule`, which specifies how the intersecting areas in the `GeometryGroup` are combined. The default value of this property is `FillRule.EvenOdd`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

NOTE

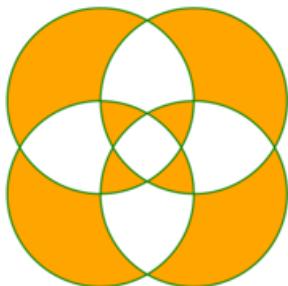
The `Children` property is the `ContentProperty` of the `GeometryGroup` class, and so does not need to be explicitly set from XAML.

For more information about the `FillRule` enumeration, see [Xamarin.Forms Shapes: Fill rules](#).

To draw a composite geometry, set the required `Geometry` objects as the children of a `GeometryGroup`, and display them with a `Path` object. The following XAML shows an example of this:

```
<Path Stroke="Green"
      StrokeThickness="2"
      Fill="Orange">
    <Path.Data>
        <GeometryGroup>
            <EllipseGeometry RadiusX="100"
                            RadiusY="100"
                            Center="150,150" />
            <EllipseGeometry RadiusX="100"
                            RadiusY="100"
                            Center="250,150" />
            <EllipseGeometry RadiusX="100"
                            RadiusY="100"
                            Center="150,250" />
            <EllipseGeometry RadiusX="100"
                            RadiusY="100"
                            Center="250,250" />
        </GeometryGroup>
    </Path.Data>
</Path>
```

In this example, four `EllipseGeometry` objects with identical x-radius and y-radius coordinates, but with different center coordinates, are combined. This creates four overlapping circles, whose interiors are filled orange due to the default `EvenOdd` fill rule:



RoundRectangleGeometry

A round rectangle geometry represents the geometry of a rectangle, or square, with rounded corners, and is defined by a corner radius and a `Rect` structure that specifies its relative position and its height and width.

The `RoundRectangleGeometry` class, which derives from the `GeometryGroup` class, defines the following properties:

- `CornerRadius`, of type `CornerRadius`, which is the corner radius of the geometry.
- `Rect`, of type `Rect`, which represents the dimensions of the rectangle.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

NOTE

The fill rule used by the `RoundRectangleGeometry` is `FillRule.Nonzero`. For more information about fill rules, see [Xamarin.Forms Shapes: Fill rules](#).

The following example shows how to create and render a `RoundRectangleGeometry` in a `Path` object:

```
<Path Fill="Blue"  
      Stroke="Red">  
  <Path.Data>  
    <RoundRectangleGeometry CornerRadius="5"  
                          Rect="10,10,150,100" />  
  </Path.Data>  
</Path>
```

The position and dimensions of the rectangle are defined by a `Rect` structure. In this example, the position is (10,10), the width is 150, and the height is 100 device-independent units. In addition, the rectangle corners are rounded with a radius of 5 device-independent units.

Clip with a Geometry

The `visualElement` class has a `Clip` property, of type `Geometry`, that defines the outline of the contents of an element. When the `Clip` property is set to a `Geometry` object, only the area that is within the region of the `Geometry` will be visible.

The following example shows how to use a `Geometry` object as the clip region for an `Image`:

```
<Image Source="monkeyface.png">  
  <Image.Clip>  
    <EllipseGeometry RadiusX="100"  
                     RadiusY="100"  
                     Center="180,180" />  
  </Image.Clip>  
</Image>
```

In this example, an `EllipseGeometry` with `RadiusX` and `RadiusY` values of 100, and a `Center` value of (180,180) is set to the `Clip` property of an `Image`. Only the part of the image that is within the area of the ellipse will be displayed:



NOTE

Simple geometries, path geometries, and composite geometries can all be used to clip `VisualElement` objects.

Other features

The `GeometryHelper` class provides the following helper methods:

- `FlattenGeometry`, which flattens a `Geometry` into a `PathGeometry`.
- `FlattenCubicBezier`, which flattens a cubic Bezier curve into a `List<Point>` collection.
- `FlattenQuadraticBezier`, which flattens a quadratic Bezier curve into a `List<Point>` collection.
- `FlattenArc`, which flattens an elliptical arc into a `List<Point>` collection.

Related links

- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Shapes](#)
- [Xamarin.Forms Shapes: Fill rules](#)

Xamarin.Forms Shapes: Line

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Line` class derives from the `Shape` class, and can be used to draw lines. For information on the properties that the `Line` class inherits from the `Shape` class, see [Xamarin.Forms Shapes](#).

`Line` defines the following properties:

- `x1`, of type double, indicates the x-coordinate of the start point of the line. The default value of this property is 0.0.
- `y1`, of type double, indicates the y-coordinate of the start point of the line. The default value of this property is 0.0.
- `x2`, of type double, indicates the x-coordinate of the end point of the line. The default value of this property is 0.0.
- `y2`, of type double, indicates the y-coordinate of the end point of the line. The default value of this property is 0.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

For information about controlling how line ends are drawn, see [Control line ends](#).

Create a Line

To draw a line, create a `Line` object and set its `x1` and `y1` properties to its start point, and its `x2` and `y2` properties to its end point. In addition, set its `Stroke` property to a `Brush`-derived object because a line without a stroke is invisible. For more information about `Brush` objects, see [Xamarin.Forms Brushes](#).

NOTE

Setting the `Fill` property of a `Line` has no effect, because a line has no interior.

The following XAML example shows how to draw a line:

```
<Line X1="40"  
      Y1="0"  
      X2="0"  
      Y2="120"  
      Stroke="Red" />
```

In this example, a red diagonal line is drawn from (40,0) to (0,120):



Because the `x1`, `y1`, `x2`, and `y2` properties have default values of 0, it's possible to draw some lines with minimal syntax:

```
<Line Stroke="Red"  
      X2="200" />
```

In this example, a horizontal line that's 200 device-independent units long is defined. Because the other properties are 0 by default, a line is drawn from (0,0) to (200,0).

The following XAML example shows how to draw a dashed line:

```
<Line X1="40"  
      Y1="0"  
      X2="0"  
      Y2="120"  
      Stroke="DarkBlue"  
      StrokeDashArray="1,1"  
      StrokeDashOffset="6" />
```

In this example, a dark blue dashed diagonal line is drawn from (40,0) to (0,120):



For more information about drawing a dashed line, see [Draw dashed shapes](#).

Related links

- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Shapes](#)
- [Xamarin.Forms Brushes](#)

Xamarin.Forms Shapes: Path

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Path` class derives from the `Shape` class, and can be used to draw curves and complex shapes. These curves and shapes are often described using `Geometry` objects. For information on the properties that the `Path` class inherits from the `Shape` class, see [Xamarin.Forms Shapes](#).

`Path` defines the following properties:

- `Data`, of type `Geometry`, which specifies the shape to be drawn.
- `RenderTransform`, of type `Transform`, which represents the transform that is applied to the geometry of a path prior to it being drawn.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

For more information about transforms, see [Xamarin.Forms Path Transforms](#).

Create a Path

To draw a path, create a `Path` object and set its `Data` property. There are two techniques for setting the `Data` property:

- You can set a string value for `Data` in XAML, using path markup syntax. With this approach, the `Path.Data` value is consuming a serialization format for graphics. Typically, you don't edit this string value by hand after it's created. Instead, you use design tools to manipulate the data, and export it as a string fragment that's consumable by the `Data` property.
- You can set the `Data` property to a `Geometry` object. This can be a specific `Geometry` object, or a `GeometryGroup` which acts as a container that can combine multiple geometry objects into a single object.

Create a Path with path markup syntax

The following XAML example shows how to draw a triangle using path markup syntax:

```
<Path Data="M 10,100 L 100,100 100,50Z"
      Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Start" />
```

The `Data` string begins with the move command, indicated by `M`, which establishes an absolute start point for the path. `L` is the line command, which creates a straight line from the start point to the specified end point. `Z` is the close command, which creates a line that connects the current point to the starting point. The result is a triangle:



For more information about path markup syntax, see [Xamarin.Forms Path markup syntax](#).

Create a Path with Geometry objects

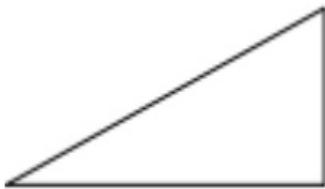
Curves and shapes can be described using `Geometry` objects, which are used to set the `Path` object's `Data` property. There are a variety of `Geometry` objects to choose from. The `EllipseGeometry`, `LineGeometry`, and `RectangleGeometry` classes describe relatively simple shapes. To create more complex shapes or create curves, use a `PathGeometry`.

`PathGeometry` objects are comprised of one or more `PathFigure` objects. Each `PathFigure` object represents a different shape. Each `PathFigure` object is itself comprised of one or more `PathSegment` objects, each representing a connection portion of the shape. Segment types include the following the `LineSegment`, `BezierSegment`, and `ArcSegment` classes.

The following XAML example shows how to draw a triangle using a `PathGeometry` object:

```
<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Start">
    <Path.Data>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigureCollection>
            <PathFigure IsClosed="True"
                       StartPoint="10,100">
              <PathFigure.Segments>
                <PathSegmentCollection>
                  <LineSegment Point="100,100" />
                  <LineSegment Point="100,50" />
                </PathSegmentCollection>
              </PathFigure.Segments>
            </PathFigure>
          </PathFigureCollection>
        </PathGeometry.Figures>
      </PathGeometry>
    </Path.Data>
  </Path>
```

In this example, the start point of the triangle is (10,100). A line segment is drawn from (10,100) to (100,100), and from (100,100) to (100,50). Then the figures first and last segments are connected, because the `PathFigure.IsClosed` property is set to `true`. The result is a triangle:



For more information about geometries, see [Xamarin.Forms Geometries](#).

Related links

- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Shapes](#)
- [Xamarin.Forms Geometries](#)
- [Xamarin.Forms Path markup syntax](#)
- [Xamarin.Forms Path transforms](#)

Xamarin.Forms Shapes: Path markup syntax

8/4/2022 • 7 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms path markup syntax enables you to compactly specify path geometries in XAML. The syntax is specified as a string value to the `Path.Data` property:

```
<Path Stroke="Black"  
      Data="M13.908992,16.207977 L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983Z" />
```

Path markup syntax is composed of an optional `FillRule` value, and one or more figure descriptions. This syntax can be expressed as: `<Path Data=" [fillRule] figureDescription [figureDescription]* " ... />`

In this syntax:

- `fillRule` is an optional `Xamarin.Forms.Shapes.FillRule` that specifies whether the geometry should use the `EvenOdd` or `Nonzero` `FillRule`. `F0` is used to specify the `EvenOdd` fill rule, while `F1` is used to specify the `Nonzero` fill rule. For more information about fill rules, see [Xamarin.Forms Shapes: Fill rules](#).
- `figureDescription` represents a figure composed of a move command, draw commands, and an optional close command. A move command specifies the start point of the figure. Draw commands describe the figure's contents, and the optional close command closes the figure.

In the example above, the path markup syntax specifies a start point using the move command (`M`), a series of straight lines using the line command (`L`), and closes the path with the close command (`Z`).

In path markup syntax, spaces are not required before or after commands. In addition, two numbers don't have to be separated by a comma or white space, but this can only be achieved when the string is unambiguous.

TIP

Path markup syntax is compatible with Scalable Vector Graphics (SVG) image path definitions, and so it can be useful for porting graphics from SVG format.

While path markup syntax is intended for consumption in XAML, it can be converted to a `Geometry` object in code by invoking the `ConvertFromInvariantString` method in the `PathGeometryConverter` class:

```
Geometry pathData = (Geometry)new PathGeometryConverter().ConvertFromInvariantString("M13.908992,16.207977  
L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983Z");
```

Move command

The move command specifies the start point of a new figure. The syntax for this command is: `M startPoint` or `m startPoint`.

In this syntax, `startPoint` is a `Point` structure that specifies the start point of a new figure. If you list multiple points after the move command, a line is drawn to those points.

`M 10,10` is an example of a valid move command.

Draw commands

A draw command can consist of several shape commands. The following draw commands are available:

- Line (`L` or `l`).
- Horizontal line (`H` or `h`).
- Vertical line (`V` or `v`).
- Elliptical arc (`A` or `a`).
- Cubic Bezier curve (`C` or `c`).
- Quadratic Bezier curve (`Q` or `q`).
- Smooth cubic Bezier curve (`S` or `s`).
- Smooth quadratic Bezier curve (`T` or `t`).

Each draw command is specified with a case-insensitive letter. When sequentially entering more than one command of the same type, you can omit the duplicate command entry. For example `L 100,200 300,400` is equivalent to `L 100,200 L 300,400`.

Line command

The line command creates a straight line between the current point and the specified end point. The syntax for this command is: `L endPoint` or `l endPoint`

In this syntax, `endPoint` is a `Point` that represents the end point of the line.

`L 20,30` and `L 20 30` are examples of valid line commands.

For information about creating a straight line as a `PathGeometry` object, see [Create a LineSegment](#).

Horizontal line command

The horizontal line command creates a horizontal line between the current point and the specified x-coordinate. The syntax for this command is: `H x` or `h x`.

In this syntax, `x` is a `double` that represents the x-coordinate of the end point of the line.

`H 90` is an example of a valid horizontal line command.

Vertical line command

The vertical line command creates a vertical line between the current point and the specified y-coordinate. The syntax for this command is: `V y` or `v y`.

In this syntax, `y` is a `double` that represents the y-coordinate of the end point of the line.

`V 90` is an example of a valid vertical line command.

Elliptical arc command

The elliptical arc command creates an elliptical arc between the current point and the specified end point. The syntax for this command is: `A size rotationAngle isLargeArcFlag sweepDirectionFlag endPoint` or `a size rotationAngle isLargeArcFlag sweepDirectionFlag endPoint`.

In this syntax:

- `size` is a `Size` that represents the x- and y-radius of the arc.
- `rotationAngle` is a `double` that represents the rotation of the ellipse, in degrees.
- `isLargeArcFlag` should be set to 1 if the angle of the arc should be 180 degrees or greater, otherwise set it to 0.
- `sweepDirectionFlag` should be set to 1 if the arc is drawn in a positive-angle direction, otherwise set it to 0.

- `endPoint` is a [Point](#) to which the arc is drawn.

`A 150,150 0 1,0 150,-150` is an example of a valid elliptical arc command.

For information about creating an elliptical arc as a [PathGeometry](#) object, see [Create an ArcSegment](#).

Cubic Bezier curve command

The cubic Bezier curve command creates a cubic Bezier curve between the current point and the specified end point by using the two specified control point. The syntax for this command is: `c controlPoint1 controlPoint2 endPoint` or `c controlPoint1 controlPoint2 endPoint`.

In this syntax:

- `controlPoint1` is a [Point](#) that represents the first control point of the curve, which determines the starting tangent of the curve.
- `controlPoint2` is a [Point](#) that represents the second control point of the curve, which determines the ending tangent of the curve.
- `endPoint` is a [Point](#) that represents the point to which the curve is drawn.

`C 100,200 200,400 300,200` is an example of a valid cubic Bezier curve command.

For information about creating a cubic Bezier curve as a [PathGeometry](#) object, see [Create a BezierSegment](#).

Quadratic Bezier curve command

The quadratic Bezier curve command creates a quadratic Bezier curve between the current point and the specified end point by using the specified control point. The syntax for this command is: `q controlPoint endPoint` or `q controlPoint endPoint`.

In this syntax:

- `controlPoint` is a [Point](#) that represents the control point of the curve, which determines the starting and ending tangents of the curve.
- `endPoint` is a [Point](#) that represents the point to which the curve is drawn.

`Q 100,200 300,200` is an example of a valid quadratic Bezier curve command.

For information about creating a quadratic Bezier curve as a [PathGeometry](#) object, see [Create a QuadraticBezierSegment](#).

Smooth cubic Bezier curve command

The smooth cubic Bezier curve command creates a cubic Bezier curve between the current point and the specified end point by using the specified control point. The syntax for this command is: `s controlPoint2 endPoint` or `s controlPoint2 endPoint`.

In this syntax:

- `controlPoint2` is a [Point](#) that represents the second control point of the curve, which determines the ending tangent of the curve.
- `endPoint` is a [Point](#) that represents the point to which the curve is drawn.

The first control point is assumed to be the reflection of the second control point of the previous command, relative to the current point. If there is no previous command, or the previous command was not a cubic Bezier curve command or a smooth cubic Bezier curve command, the first control point is assumed to be coincident with the current point.

`S 100,200 200,300` is an example of a valid smooth cubic Bezier curve command.

Smooth quadratic Bezier curve command

The smooth quadratic Bezier curve command creates a quadratic Bezier curve between the current point and the specified end point by using a control point. The syntax for this command is: `T` `endPoint` or `t` `endPoint`.

In this syntax, `endPoint` is a `Point` that represents the point to which the curve is drawn.

The control point is assumed to be the reflection of the control point of the previous command relative to the current point. If there is no previous command or if the previous command was not a quadratic Bezier curve or a smooth quadratic Bezier curve command, the control point is assumed to be coincident with the current point.

`T 100,30` is an example of a valid smooth quadratic cubic Bezier curve command.

Close command

The close command ends the current figure and creates a line that connects the current point to the starting point of the figure. Therefore, this command creates a line-join between the last segment and the first segment of the figure.

The syntax for the close command is: `z` or `Z`.

Additional values

Instead of a standard numerical value, you can also use the following case-sensitive special values:

- `Infinity` represents `double.PositiveInfinity`.
- `-Infinity` represents `double.NegativeInfinity`.
- `Nan` represents `double.NaN`.

In addition, you may also use case-insensitive scientific notation. Therefore, `+1.e17` is a valid value.

Related links

- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Shapes: Geometries](#)
- [Xamarin.Forms Shapes: Fill rules](#)

Xamarin.Forms Shapes: Path transforms

8/4/2022 • 12 minutes to read • [Edit Online](#)



[Download the sample](#)

A `Transform` defines how to transform a `Path` object from one coordinate space to another coordinate space. When a transform is applied to a `Path` object, it changes how the object is rendered in the UI.

Transforms can be categorized into four general classifications: rotation, scaling, skew, and translation. Xamarin.Forms defines a class for each of these transform classifications:

- `RotateTransform`, which rotates a `Path` by a specified `Angle`.
- `ScaleTransform`, which scales a `Path` object by specified `ScaleX` and `ScaleY` amounts.
- `SkewTransform`, which skews a `Path` object by specified `AngleX` and `AngleY` amounts.
- `TranslateTransform`, which moves a `Path` object by specified `X` and `Y` amounts.

Xamarin.Forms also provides the following classes for creating more complex transformations:

- `TransformGroup`, which represents a composite transform composed of multiple transform objects.
- `CompositeTransform`, which applies multiple transform operations to a `Path` object.
- `MatrixTransform`, which creates custom transforms that are not provided by the other transform classes.

All of these classes derive from the `Transform` class, which defines a `Value` property of type `Matrix`, which represents the current transformation as a `Matrix` object. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled. For more information about the `Matrix` struct, see [Transform matrix](#).

To apply a transform to a `Path`, you create a transform class and set it as the value of the `Path.RenderTransform` property.

Rotation transform

A rotate transform rotates a `Path` object clockwise about a specified point in a 2D x-y coordinate system.

The `RotateTransform` class, which derives from the `Transform` class, defines the following properties:

- `Angle`, of type `double`, represents the angle, in degrees, of clockwise rotation. The default value of this property is 0.0.
- `CenterX`, of type `double`, represents the x-coordinate of the rotation center point. The default value of this property is 0.0.
- `CenterY`, of type `double`, represents the y-coordinate of the rotation center point. The default value of this property is 0.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `centerX` and `centerY` properties specify the point about which the `Path` object is rotated. This center point is expressed in the coordinate space of the object that's transformed. By default, the rotation is applied to (0,0), which is the upper-left corner of the `Path` object.

The following example shows how to rotate a `Path` object:

```

<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Center"
      HeightRequest="100"
      WidthRequest="100"
      Data="M13.908992,16.207977L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983z">
  <Path.RenderTransform>
    <RotateTransform CenterX="0"
                    CenterY="0"
                    Angle="45" />
  </Path.RenderTransform>
</Path>

```

In this example, the `Path` object is rotated 45 degrees about its upper-left corner.

Scale transform

A scale transform scales a `Path` object in the 2D x-y coordinate system.

The `ScaleTransform` class, which derives from the `Transform` class, defines the following properties:

- `ScaleX`, of type `double`, which represents the x-axis scale factor. The default value of this property is 1.0.
- `ScaleY`, of type `double`, which represents the y-axis scale factor. The default value of this property is 1.0.
- `CenterX`, of type `double`, which represents the x-coordinate of the center point of this transform. The default value of this property is 0.0.
- `CenterY`, of type `double`, which represents the y-coordinate of the center point of this transform. The default value of this property is 0.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The value of `ScaleX` and `ScaleY` have a huge impact on the resulting scaling:

- Values between 0 and 1 decrease the width and height of the scaled object.
- Values greater than 1 increase the width and height of the scaled object.
- Values of 1 indicate that the object is not scaled.
- Negative values flip the scale object horizontally and vertically.
- Values between 0 and -1 flip the scale object and decrease its width and height.
- Values less than -1 flip the object and increase its width and height.
- Values of -1 flip the scaled object but do not change its horizontal or vertical size.

The `centerX` and `centerY` properties specify the point about which the `Path` object is scaled. This center point is expressed in the coordinate space of the object that's transformed. By default, scaling is applied to (0,0), which is the upper-left corner of the `Path` object. This has the effect of moving the `Path` object and making it appear larger, because when you apply a transform you change the coordinate space in which the `Path` object resides.

The following example shows how to scale a `Path` object:

```

<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Center"
      HeightRequest="100"
      WidthRequest="100"
      Data="M13.908992,16.207977L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983z">
<Path.RenderTransform>
    <ScaleTransform CenterX="0"
                    CenterY="0"
                    ScaleX="1.5"
                    ScaleY="1.5" />
</Path.RenderTransform>
</Path>

```

In this example, the `Path` object is scaled to 1.5 times the size.

Skew transform

A skew transform skews a `Path` object in the 2D x-y coordinate system, and is useful for creating the illusion of 3D depth in a 2D object.

The `SkewTransform` class, which derives from the `Transform` class, defines the following properties:

- `AngleX`, of type `double`, which represents the x-axis skew angle, which is measured in degrees counterclockwise from the y-axis. The default value of this property is 0.0.
- `AngleY`, of type `double`, which represents the y-axis skew angle, which is measured in degrees counterclockwise from the x-axis. The default value of this property is 0.0.
- `CenterX`, of type `double`, which represents the x-coordinate of the transform center. The default value of this property is 0.0.
- `CenterY`, of type `double`, which represents the y-coordinate of the transform center. The default value of this property is 0.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

To predict the effect of a skew transformation, consider that `AngleX` skews x-axis values relative to the original coordinate system. Therefore, for an `AngleX` of 30, the y-axis rotates 30 degrees through the origin and skews the values in x by 30 degrees from that origin. Similarly, an `AngleY` of 30 skews the y values of the `Path` object by 30 degrees from the origin.

NOTE

To skew a `Path` object in place, set the `CenterX` and `CenterY` properties to the object's center point.

The following example shows how to skew a `Path` object:

```

<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Center"
      HeightRequest="100"
      WidthRequest="100"
      Data="M13.908992,16.207977L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983z">
<Path.RenderTransform>
    <SkewTransform CenterX="0"
                  CenterY="0"
                  AngleX="45"
                  AngleY="0" />
</Path.RenderTransform>
</Path>

```

In this example, a horizontal skew of 45 degrees is applied to the `Path` object, from a center point of (0,0).

Translate transform

A translate transform moves an object in the 2D x-y coordinate system.

The `TranslateTransform` class, which derives from the `Transform` class, defines the following properties:

- `X`, of type `double`, which represents the distance to move along the x-axis. The default value of this property is 0.0.
- `Y`, of type `double`, which represents the distance to move along the y-axis. The default value of this property is 0.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

Negative `X` values move an object to the left, while positive values move an object to the right. Negative `Y` values move an object up, while positive values move an object down.

The following example shows how to translate a `Path` object:

```

<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Center"
      HeightRequest="100"
      WidthRequest="100"
      Data="M13.908992,16.207977L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983z">
<Path.RenderTransform>
    <TranslateTransform X="50"
                      Y="50" />
</Path.RenderTransform>
</Path>

```

In this example, the `Path` object is moved 50 device-independent units to the right, and 50 device-independent units down.

Multiple transforms

Xamarin.Forms has two classes that support applying multiple transforms to a `Path` object. These are `TransformGroup`, and `CompositeTransform`. A `TransformGroup` performs transforms in any desired order, while a `CompositeTransform` performs transforms in a specific order.

Transform groups

Transform groups represent composite transforms composed of multiple `Transform` objects.

The `TransformGroup` class, which derives from the `Transform` class, defines a `Children` property, of type `TransformCollection`, which represents a collection of `Transform` objects. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

The order of transformations is important in a composite transform that uses the `TransformGroup` class. For example, if you first rotate, then scale, then translate, you get a different result than if you first translate, then rotate, then scale. One reason order is significant is that transforms like rotation and scaling are performed respect to the origin of the coordinate system. Scaling an object that is centered at the origin produces a different result to scaling an object that has been moved away from the origin. Similarly, rotating an object that is centered at the origin produces a different result than rotating an object that has been moved away from the origin.

The following example shows how to perform a composite transform using the `TransformGroup` class:

```
<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Center"
      HeightRequest="100"
      WidthRequest="100"
      Data="M13.908992,16.207977L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983z">
<Path.RenderTransform>
    <TransformGroup>
        <ScaleTransform ScaleX="1.5"
                      ScaleY="1.5" />
        <RotateTransform Angle="45" />
    </TransformGroup>
</Path.RenderTransform>
</Path>
```

In this example, the `Path` object is scaled to 1.5 times its size, and then rotated by 45 degrees.

Composite transforms

A composite transform applies multiple transforms to an object.

The `compositeTransform` class, which derives from the `Transform` class, defines the following properties:

- `CenterX`, of type `double`, which represents the x-coordinate of the center point of this transform. The default value of this property is 0.0.
- `CenterY`, of type `double`, which represents the y-coordinate of the center point of this transform. The default value of this property is 0.0.
- `ScaleX`, of type `double`, which represents the x-axis scale factor. The default value of this property is 1.0.
- `ScaleY`, of type `double`, which represents the y-axis scale factor. The default value of this property is 1.0.
- `SkewX`, of type `double`, which represents the x-axis skew angle, which is measured in degrees counterclockwise from the y-axis. The default value of this property is 0.0.
- `SkewY`, of type `double`, which represents the y-axis skew angle, which is measured in degrees counterclockwise from the x-axis. The default value of this property is 0.0.
- `Rotation`, of type `double`, represents the angle, in degrees, of clockwise rotation. The default value of this property is 0.0.
- `TranslateX`, of type `double`, which represents the distance to move along the x-axis. The default value of this property is 0.0.
- `TranslateY`, of type `double`, which represents the distance to move along the y-axis. The default value of this property is 0.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data

bindings, and styled.

A `CompositeTransform` applies transforms in this order:

1. Scale (`ScaleX` and `ScaleY`).
2. Skew (`SkewX` and `SkewY`).
3. Rotate (`Rotation`).
4. Translate (`TranslateX`, `TranslateY`).

If you want to apply multiple transforms to an object in a different order, you should create a `TransformGroup` and insert the transforms in your intended order.

IMPORTANT

A `CompositeTransform` uses the same center points, `CenterX` and `CenterY`, for all transformations. If you want to specify different center points per transform, use a `TransformGroup`,

The following example shows how to perform a composite transform using the `CompositeTransform` class:

```
<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Center"
      HeightRequest="100"
      WidthRequest="100"
      Data="M13.908992,16.207977L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983z">
  <Path.RenderTransform>
    <CompositeTransform ScaleX="1.5"
                        ScaleY="1.5"
                        Rotation="45"
                        TranslateX="50"
                        TranslateY="50" />
  </Path.RenderTransform>
</Path>
```

In this example, the `Path` object is scaled to 1.5 times its size, then rotated by 45 degrees, and then translated by 50 device-independent units.

Transform matrix

A transform can be described in terms of a 3x3 affine transformation matrix, that performs transformations in 2D space. This 3x3 matrix is represented by the `Matrix` struct, which is a collection of three rows and three columns of `double` values.

The `Matrix` struct defines the following properties:

- `Determinant`, of type `double`, which gets the determinant of the matrix.
- `HasInverse`, of type `bool`, which indicates whether the matrix is invertible.
- `Identity`, of type `Matrix`, which gets an identity matrix.
- `HasIdentity`, of type `bool`, which indicates whether the matrix is an identity matrix.
- `M11`, of type `double`, which represents the value of the first row and first column of the matrix.
- `M12`, of type `double`, which represents the value of the first row and second column of the matrix.
- `M21`, of type `double`, which represents the value of the second row and first column of the matrix.
- `M22`, of type `double`, which represents the value of the second row and second column of the matrix.
- `offsetX`, of type `double`, which represents the value of the third row and first column of the matrix.
- `offsetY`, of type `double`, which represents the value of the third row and second column of the matrix.

The `OffsetX` and `OffsetY` properties are so named because they specify the amount to translate the coordinate space along the x-axis, and y-axis, respectively.

In addition, the `Matrix` struct exposes a series of methods that can be used to manipulate the matrix values, including `Append`, `Invert`, `Multiply`, `Prepend` and many more.

The following table shows the structure of a Xamarin.Forms matrix:

M11

M12

0.0

M21

M22

0.0

OffsetX

OffsetY

1.0

NOTE

An affine transformation matrix has its final column equal to (0,0,1), so only the members in the first two columns need to be specified.

By manipulating matrix values, you can rotate, scale, skew, and translate `Path` objects. For example, if you change the `OffsetX` value to 100, you can use it move a `Path` object 100 device-independent units along the x-axis. If you change the `M22` value to 3, you can use it to stretch a `Path` object to three times its current height. If you change both values, you move the `Path` object 100 device-independent units along the x-axis and stretch its height by a factor of 3. In addition, affine transformation matrices can be multiplied to form any number of linear transformations, such as rotation and skew, followed by translation.

Custom transforms

The `MatrixTransform` class, which derives from the `Transform` class, defines a `Matrix` property, of type `Matrix`, which represents the matrix that defines the transformation. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

Any transform that you can describe with a `TranslateTransform`, `ScaleTransform`, `RotateTransform`, or `SkewTransform` object can equally be described by a `MatrixTransform`. However, the `TranslateTransform`, `ScaleTransform`, `RotateTransform`, and `SkewTransform` classes are easier to conceptualize than setting the vector components in a `Matrix`. Therefore, the `MatrixTransform` class is typically used to create custom transformations that aren't provided by the `RotateTransform`, `ScaleTransform`, `SkewTransform`, or `TranslateTransform` classes.

The following example shows how to transform a `Path` object using a `MatrixTransform`:

```
<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Center"
      Data="M13.908992,16.207977L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983z">
  <Path.RenderTransform>
    <MatrixTransform>
      <MatrixTransform.Matrix>
        <!-- M11 stretches, M12 skews -->
        <Matrix OffsetX="10"
                 OffsetY="100"
                 M11="1.5"
                 M12="1" />
      </MatrixTransform.Matrix>
    </MatrixTransform>
  </Path.RenderTransform>
</Path>
```

In this example, the `Path` object is stretched, skewed, and offset in both the X and Y dimensions.

Alternatively, this can be written in a simplified form that uses a type converter that's built into Xamarin.Forms:

```
<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Center"
      Data="M13.908992,16.207977L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983z">
  <Path.RenderTransform>
    <MatrixTransform Matrix="1.5,1,0,1,10,100" />
  </Path.RenderTransform>
</Path>
```

In this example, the `Matrix` property is specified as a comma-delimited string consisting of six members: `M11`, `M12`, `M21`, `M22`, `OffsetX`, `OffsetY`. While the members are comma-delimited in this example, they can also be delimited by one or more spaces.

In addition, the previous example can be simplified even further by specifying the same six members as the value of the `RenderTransform` property:

```
<Path Stroke="Black"
      Aspect="Uniform"
      HorizontalOptions="Center"
      RenderTransform="1.5 1 0 1 10 100"
      Data="M13.908992,16.207977L32.000049,16.207977 32.000049,31.999985 13.908992,30.109983z" />
```

Related links

- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Shapes](#)

Xamarin.Forms Shapes: Polygon

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Polygon` class derives from the `Shape` class, and can be used to draw polygons, which are connected series of lines that form closed shapes. For information on the properties that the `Polygon` class inherits from the `Shape` class, see [Xamarin.Forms Shapes](#).

`Polygon` defines the following properties:

- `Points`, of type `PointCollection`, which is a collection of `Point` structures that describe the vertex points of the polygon.
- `FillRule`, of type `FillRule`, which specifies how the interior fill of the shape is determined. The default value of this property is `FillRule.EvenOdd`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `PointsCollection` type is an `ObservableCollection` of `Point` objects. The `Point` structure defines `x` and `y` properties, of type `double`, that represent an x- and y-coordinate pair in 2D space. Therefore, the `Points` property should be set to a list of x-coordinate and y-coordinate pairs that describe the polygon vertex points, delimited by a single comma and/or one or more spaces. For example, "40,10 70,80" and "40 10, 70 80" are both valid.

For more information about the `FillRule` enumeration, see [Xamarin.Forms Shapes: Fill rules](#).

Create a Polygon

To draw a polygon, create a `Polygon` object and set its `Points` property to the vertices of a shape. A line is automatically drawn that connects the first and last points. To paint the inside of the polygon, set its `Fill` property to a `Brush`-derived object. To give the polygon an outline, set its `Stroke` property to a `Brush`-derived object. The `StrokeThickness` property specifies the thickness of the polygon outline. For more information about `Brush` objects, see [Xamarin.Forms Brushes](#).

The following XAML example shows how to draw a filled polygon:

```
<Polygon Points="40,10 70,80 10,50"
          Fill="AliceBlue"
          Stroke="Green"
          StrokeThickness="5" />
```

In this example, a filled polygon that represents a triangle is drawn:



The following XAML example shows how to draw a dashed polygon:

```
<Polygon Points="40,10 70,80 10,50"
    Fill="AliceBlue"
    Stroke="Green"
    StrokeThickness="5"
    StrokeDashArray="1,1"
    StrokeDashOffset="6" />
```

In this example, the polygon outline is dashed:

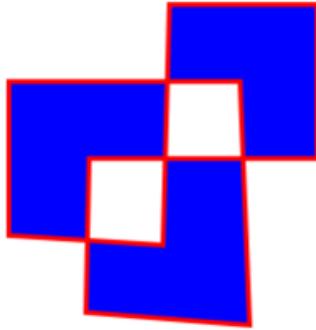


For more information about drawing a dashed polygon, see [Draw dashed shapes](#).

The following XAML example shows a polygon that uses the default fill rule:

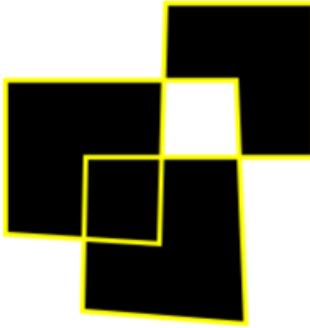
```
<Polygon Points="0 48, 0 144, 96 150, 100 0, 192 0, 192 96, 50 96, 48 192, 150 200 144 48"
    Fill="Blue"
    Stroke="Red"
    StrokeThickness="3" />
```

In this example, the fill behavior of each polygon is determined using the `EvenOdd` fill rule.



The following XAML example shows a polygon that uses the `Nonzero` fill rule:

```
<Polygon Points="0 48, 0 144, 96 150, 100 0, 192 0, 192 96, 50 96, 48 192, 150 200 144 48"
    Fill="Black"
    FillRule="Nonzero"
    Stroke="Yellow"
    StrokeThickness="3" />
```



In this example, the fill behavior of each polygon is determined using the `Nonzero` fill rule.

Related links

- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Shapes](#)
- [Xamarin.Forms Shapes: Fill rules](#)
- [Xamarin.Forms Brushes](#)

Xamarin.Forms Shapes: Polyline

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Polyline` class derives from the `Shape` class, and can be used to draw a series of connected straight lines. A polyline is similar to a polygon, except the last point in a polyline is not connected to the first point. For information on the properties that the `Polyline` class inherits from the `Shape` class, see [Xamarin.Forms Shapes](#).

`Polyline` defines the following properties:

- `Points`, of type `PointCollection`, which is a collection of `Point` structures that describe the vertex points of the polyline.
- `FillRule`, of type `FillRule`, which specifies how the intersecting areas in the polyline are combined. The default value of this property is `FillRule.EvenOdd`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `PointsCollection` type is an `ObservableCollection` of `Point` objects. The `Point` structure defines `x` and `y` properties, of type `double`, that represent an x- and y-coordinate pair in 2D space. Therefore, the `Points` property should be set to a list of x-coordinate and y-coordinate pairs that describe the polyline vertex points, delimited by a single comma and/or one or more spaces. For example, "40,10 70,80" and "40 10, 70 80" are both valid.

For more information about the `FillRule` enumeration, see [Xamarin.Forms Shapes: Fill rules](#).

Create a Polyline

To draw a polyline, create a `Polyline` object and set its `Points` property to the vertices of a shape. To give the polyline an outline, set its `Stroke` property to a `Brush`-derived object. The `StrokeThickness` property specifies the thickness of the polyline outline. For more information about `Brush` objects, see [Xamarin.Forms Brushes](#).

IMPORTANT

If you set the `Fill` property of a `Polyline` to a `Brush`-derived object, the interior space of the polyline is painted, even if the start point and end point do not intersect.

The following XAML example shows how to draw a polyline:

```
<Polyline Points="0,0 10,30, 15,0 18,60 23,30 35,30 40,0 43,60 48,30 100,30"
           Stroke="Red" />
```

In this example, a red polyline is drawn:



The following XAML example shows how to draw a dashed polyline:

```
<Polyline Points="0,0 10,30, 15,0 18,60 23,30 35,30 40,0 43,60 48,30 100,30"
    Stroke="Red"
    StrokeThickness="2"
    StrokeDashArray="1,1"
    StrokeDashOffset="6" />
```

In this example, the polyline is dashed:

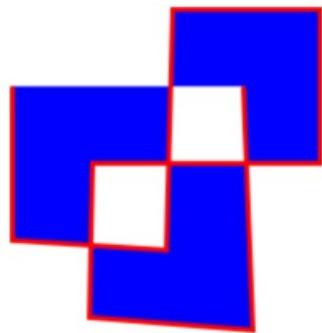


For more information about drawing a dashed polyline, see [Draw dashed shapes](#).

The following XAML example shows a polyline that uses the default fill rule:

```
<Polyline Points="0 48, 0 144, 96 150, 100 0, 192 0, 192 96, 50 96, 48 192, 150 200 144 48"
    Fill="Blue"
    Stroke="Red"
    StrokeThickness="3" />
```

In this example, the fill behavior of the polyline is determined using the `EvenOdd` fill rule.



The following XAML example shows a polyline that uses the `Nonzero` fill rule:

```
<Polyline Points="0 48, 0 144, 96 150, 100 0, 192 0, 192 96, 50 96, 48 192, 150 200 144 48"
    Fill="Black"
    FillRule="Nonzero"
    Stroke="Yellow"
    StrokeThickness="3" />
```



In this example, the fill behavior of the polyline is determined using the `Nonzero` fill rule.

Related links

- [ShapeDemos \(sample\)](#)

- [Xamarin.Forms Shapes](#)
- [Xamarin.Forms Shapes: Fill rules](#)
- [Xamarin.Forms Brushes](#)

Xamarin.Forms Shapes: Rectangle

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Rectangle` class derives from the `Shape` class, and can be used to draw rectangles and squares. For information on the properties that the `Rectangle` class inherits from the `Shape` class, see [Xamarin.Forms Shapes](#).

`Rectangle` defines the following properties:

- `RadiusX`, of type `double`, which is the x-axis radius that's used to round the corners of the rectangle. The default value of this property is 0.0.
- `RadiusY`, of type `double`, which is the y-axis radius that's used to round the corners of the rectangle. The default value of this property is 0.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `Rectangle` class sets the `Aspect` property, inherited from the `Shape` class, to `Stretch.Fill`. For more information about the `Aspect` property, see [Stretch shapes](#).

Create a Rectangle

To draw a rectangle, create a `Rectangle` object and sets its `WidthRequest` and `HeightRequest` properties. To paint the inside of the rectangle, set its `Fill` property to a `Brush`-derived object. To give the rectangle an outline, set its `Stroke` property to a `Brush`-derived object. The `StrokeThickness` property specifies the thickness of the rectangle outline. For more information about `Brush` objects, see [Xamarin.Forms Brushes](#).

To give the rectangle rounded corners, set its `RadiusX` and `RadiusY` properties. These properties set the x-axis and y-axis radii that's used to round the corners of the rectangle.

To draw a square, make the `WidthRequest` and `HeightRequest` properties of the `Rectangle` object equal.

The following XAML example shows how to draw a filled rectangle:

```
<Rectangle Fill="Red"
           WidthRequest="150"
           HeightRequest="50"
           HorizontalOptions="Start" />
```

In this example, a red filled rectangle with dimensions 150x50 (device-independent units) is drawn:



The following XAML example shows how to draw a filled rectangle, with rounded corners:

```
<Rectangle Fill="Blue"  
          Stroke="Black"  
          StrokeThickness="3"  
          RadiusX="50"  
          RadiusY="10"  
          WidthRequest="200"  
          HeightRequest="100"  
          HorizontalOptions="Start" />
```

In this example, a blue filled rectangle with rounded corners is drawn:



For information about drawing a dashed rectangle, see [Draw dashed shapes](#).

Related links

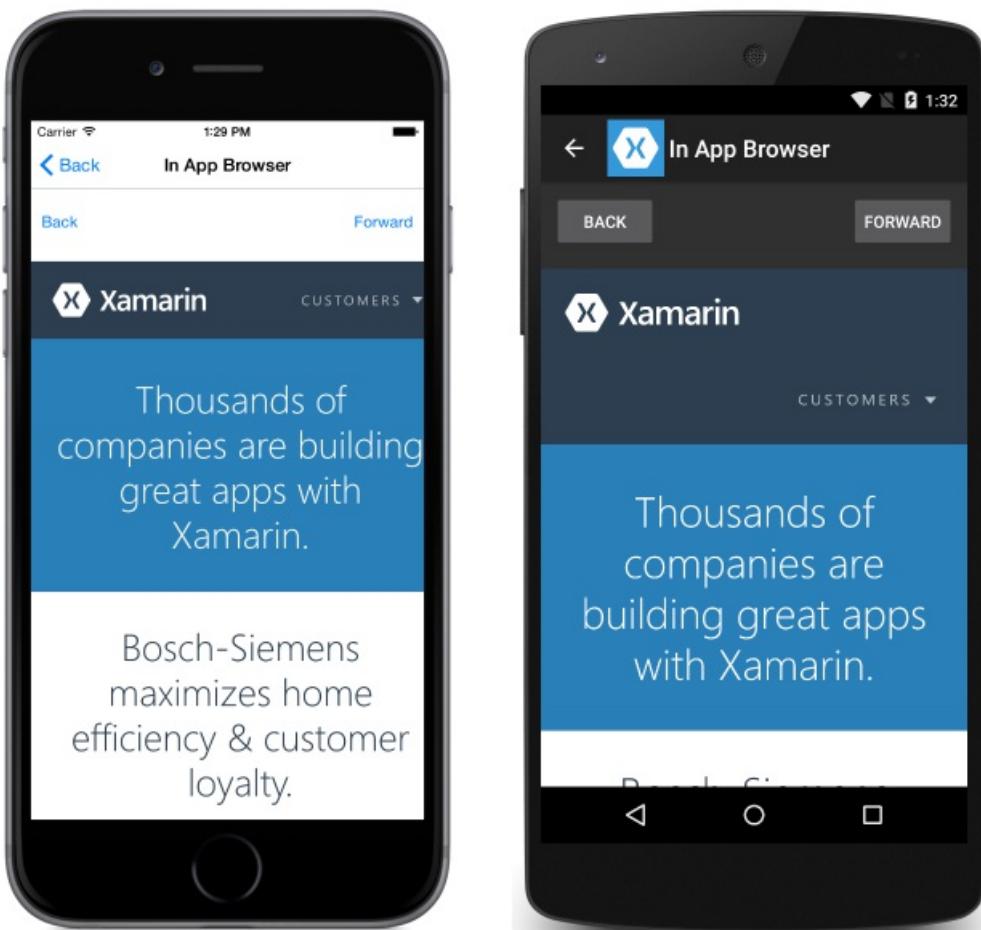
- [ShapeDemos \(sample\)](#)
- [Xamarin.Forms Shapes](#)
- [Xamarin.Forms Brushes](#)

Xamarin.Forms WebView

8/4/2022 • 14 minutes to read • [Edit Online](#)

 [Download the sample](#)

`WebView` is a view for displaying web and HTML content in 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 in the formats that are supported by the underlying platform.
- 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 = "https://dotnet.microsoft.com/apps/xamarin"
};
```

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 unsecure servers.

NOTE

If your application requires a connection to an unsecure website, you should always enter the domain as an exception using `NSExceptionDomains` 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>NSExceptionDomains</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 [verbatim string literal](#), meaning most escape characters are ignored.

NOTE

It may be necessary to set the `WidthRequest` and `HeightRequest` properties of the `WebView` to see the HTML content, depending upon the layout the `WebView` is a child of. For example, this is required in a [`StackLayout`](#).

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`:

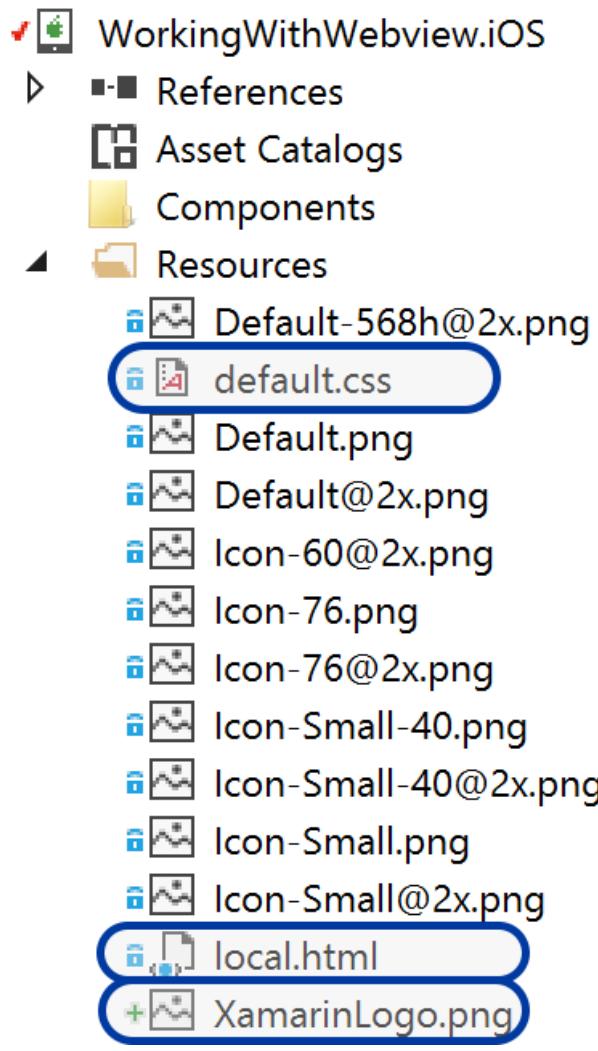
```
var source = new HtmlWebViewSource();
source.BaseUrl = DependencyService.Get<IBaseUrl>().Get();
```

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](#)



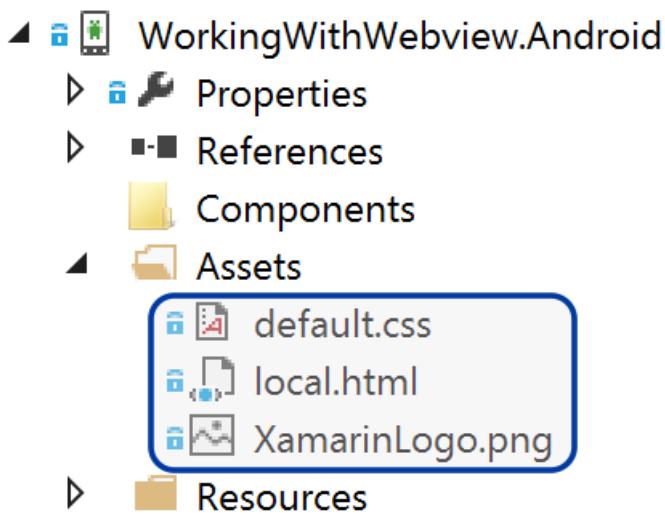
The `BaseUrl` should be set to the path of the main bundle:

```
[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();  
}
```

Universal Windows Platform

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>
```

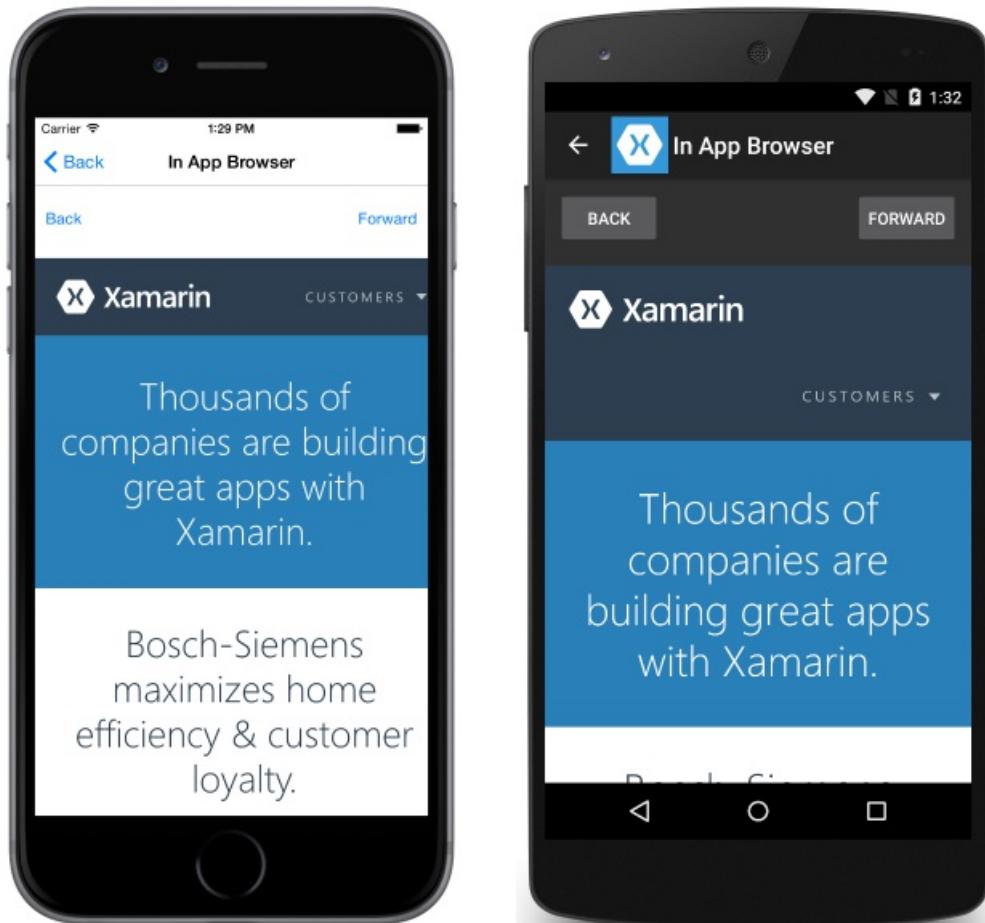
In the code-behind:

```
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.

The `WebNavigatingEventArgs` object that accompanies the `Navigating` event has four properties:

- `Cancel` – indicates whether or not to cancel the navigation.
- `NavigationEvent` – the navigation event that was raised.
- `Source` – the element that performed the navigation.
- `Url` – the navigation destination.

The `WebNavigatedEventArgs` object that accompanies the `Navigated` event has four properties:

- `NavigationEvent` – the navigation event that was raised.
- `Result` – describes the result of the navigation, using a `WebNavigationResult` enumeration member. Valid values are `Cancel`, `Failure`, `Success`, and `Timeout`.
- `Source` – the element that performed the navigation.
- `Url` – the navigation destination.

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:

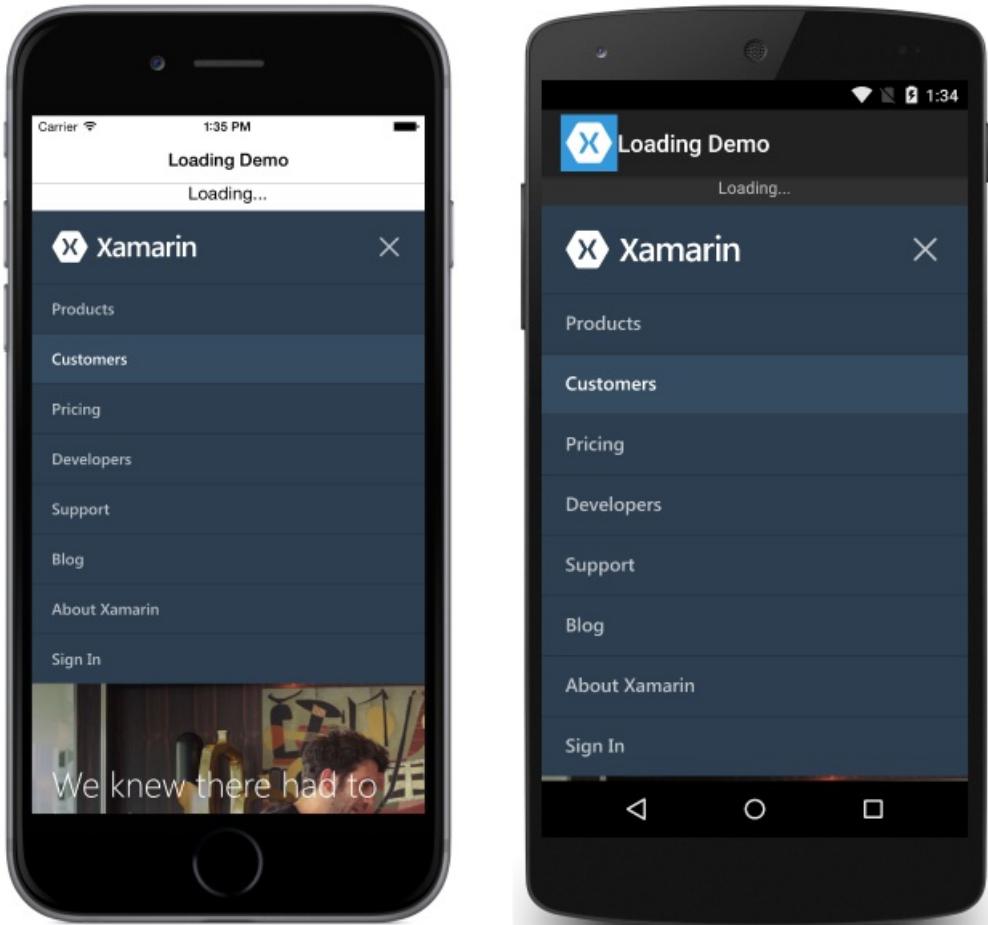
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="WebViewSample.LoadingLabelXaml"
    Title="Loading Demo">
    <StackLayout>
        <!--Loading label should not render by default.-->
        <Label x:Name="labelLoading" Text="Loading..." IsVisible="false" />
        <WebView HeightRequest="1000" WidthRequest="1000" Source="https://dotnet.microsoft.com/apps/xamarin"
            Navigated="webViewNavigated" Navigating="webViewNavigating" />
    </StackLayout>
</ContentPage>
```

The two event handlers:

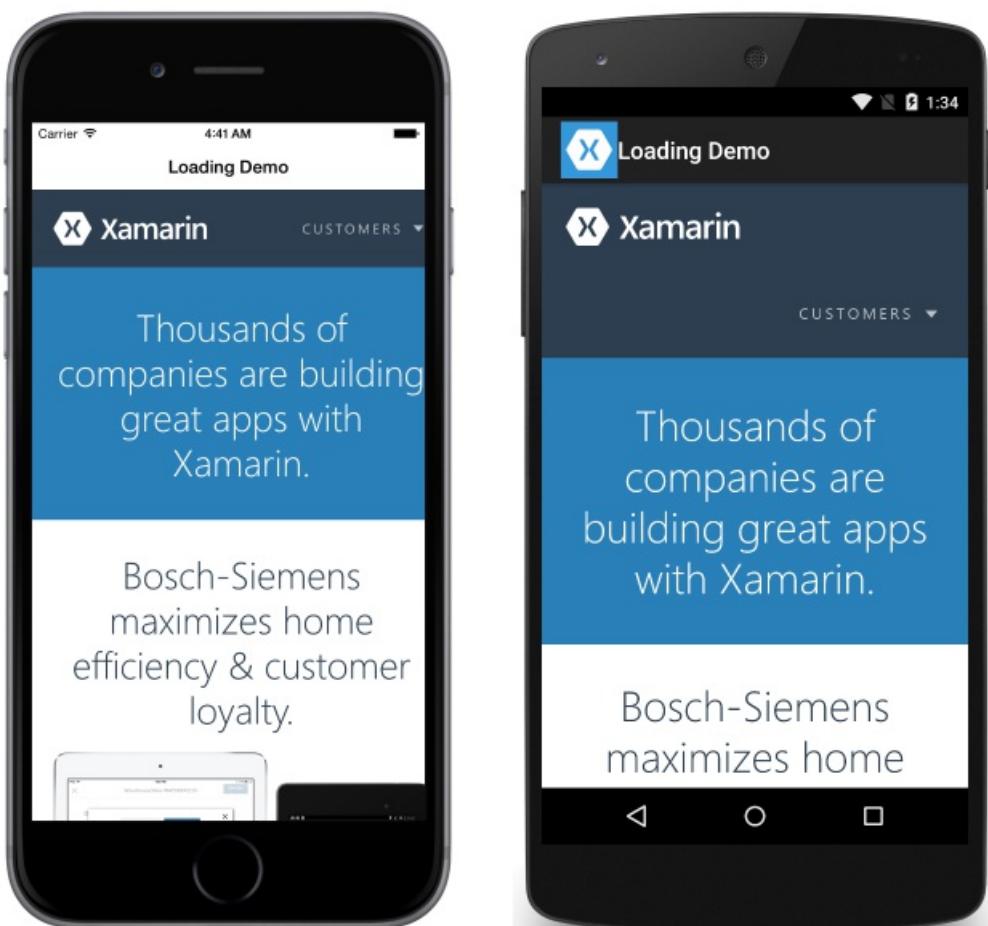
```
void webViewNavigating(object sender, WebNavigatingEventArgs e)
{
    labelLoading.IsVisible = true;
}

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

Popular web browsers adopt technologies like hardware accelerated rendering and JavaScript compilation. Prior to Xamarin.Forms 4.4, the Xamarin.Forms `WebView` was implemented on iOS by the `UIWebView` class. However, many of these technologies were unavailable in this implementation. Therefore, since Xamarin.Forms 4.4, the Xamarin.Forms `WebView` is implemented on iOS by the `WKWebView` class, which supports faster browsing.

NOTE

On iOS, the `WKWebViewRenderer` has a constructor overload that accepts a `WKWebViewConfiguration` argument. This enables the renderer to be configured on creation.

An application can return to using the iOS `UIWebView` class to implement the Xamarin.Forms `WebView`, for compatibility reasons. 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 UIWebView instead of WKWebView.
[assembly: ExportRenderer(typeof(Xamarin.Forms.WebView),
typeof(Xamarin.Forms.Platform.iOS.WebViewRenderer))]
```

NOTE

In Xamarin.Forms 5.0, the `WebViewRenderer` class has been removed. Therefore, Xamarin.Forms 5.0 doesn't contain a reference to the `UIWebView` control.

`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="https://dotnet.microsoft.com/apps/xamarin"
        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="https://dotnet.microsoft.com/apps/xamarin"
        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="https://dotnet.microsoft.com/apps/xamarin"
        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="https://dotnet.microsoft.com/apps/xamarin" Grid.Row="1" />
</Grid>
```

Invoking JavaScript

`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>
```

Cookies

Cookies can be set on a `WebView`, which are then sent with the web request to the specified URL. This is accomplished by adding `Cookie` objects to a `CookieContainer`, which is then set as the value of the `WebView.Cookies` bindable property. The following code shows an example of this:

```

using System.Net;
using Xamarin.Forms;
// ...

CookieContainer cookieContainer = new CookieContainer();
Uri uri = new Uri("https://dotnet.microsoft.com/apps/xamarin", UriKind.RelativeOrAbsolute);

Cookie cookie = new Cookie
{
    Name = "XamarinCookie",
    Expires = DateTime.Now.AddDays(1),
    Value = "My cookie",
    Domain = uri.Host,
    Path = "/"
};
cookieContainer.Add(uri, cookie);
webView.Cookies = cookieContainer;
webView.Source = new UrlWebViewSource { Url = uri.ToString() };
```

In this example, a single `cookie` is added to the `CookieContainer` object, which is then set as the value of the `WebView.Cookies` property. When the `webView` sends a web request to the specified URL, the cookie is sent with the request.

UIWebView Deprecation and App Store Rejection (ITMS-90809)

Starting in April 2020, [Apple will reject apps](#) that still use the deprecated `UIWebView` API. While Xamarin.Forms has switched to `WKWebView` as the default, there is still a reference to the older SDK in the Xamarin.Forms binaries. Current [iOS linker](#) behavior does not remove this, and as a result the deprecated `UIWebView` API will still appear to be referenced from your app when you submit to the App Store.

IMPORTANT

In Xamarin.Forms 5.0, the `WebViewRenderer` class has been removed. Therefore, Xamarin.Forms 5.0 doesn't contain a reference to the `UIWebView` control.

A preview version of the linker is available to fix this issue. To enable the preview, you will need to supply an additional argument `--optimize=experimental-xforms-product-type` to the linker.

The prerequisites for this to work are:

- **Xamarin.Forms 4.5 or higher.** Xamarin.Forms 4.6, or higher, is required if your app uses Material Visual.
- **Xamarin.iOS 13.10.0.17 or higher.** Check your Xamarin.iOS version [in Visual Studio](#). This version of Xamarin.iOS is included with Visual Studio for Mac 8.4.1 and Visual Studio 16.4.3.
- **Remove references to `UIWebView`.** Your code should not have any references to `UIWebView` or any classes that make use of `UIWebView`.

For more information about detecting and removing `UIWebView` references, see [UIWebView deprecation](#).

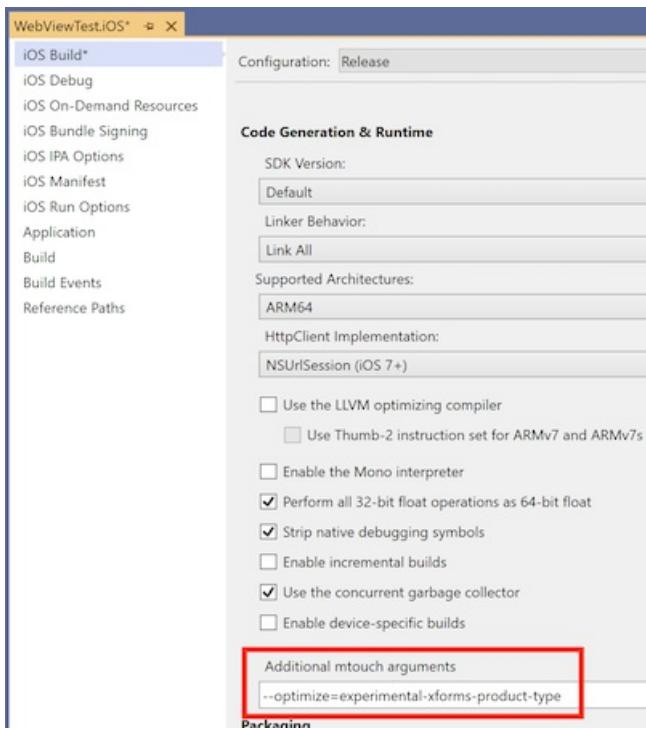
Configure the linker

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

Follow these steps for the linker to remove `UIWebView` references:

1. **Open iOS project properties** – Right-click your iOS project and choose **Properties**.
2. **Navigate to the iOS Build section** – Select the **iOS Build** section.
3. **Update the Additional mtouch arguments** – In the **Additional mtouch arguments** add this flag `--optimize=experimental-xforms-product-type` (in addition to any value that might already be in there). Note: this flag works together with the **Linker Behavior** set to **SDK Only** or **Link All**. If, for any reason, you see errors when setting the Linker Behavior to All, this is most likely a problem within the app code or a third-party library that is not linker safe. For more information on the linker, see [Linking Xamarin.iOS Apps](#).
4. **Update all build configurations** – Use the **Configuration** and **Platform** lists at the top of the window to update all build configurations. The most important configuration to update is the **Release/iPhone** configuration, since that is typically used to create builds for App Store submission.

You can see the window with the new flag in place in this screenshot:



Now when you create a new (release) build and submit it to the App Store, there should be no warnings about the deprecated API.

Related Links

- [Working with WebView \(sample\)](#)
- [WebView \(sample\)](#)
- [UIWebView deprecation](#)

Xamarin.Forms Button

8/4/2022 • 19 minutes to read • [Edit Online](#)



[Download the sample](#)

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`:

```

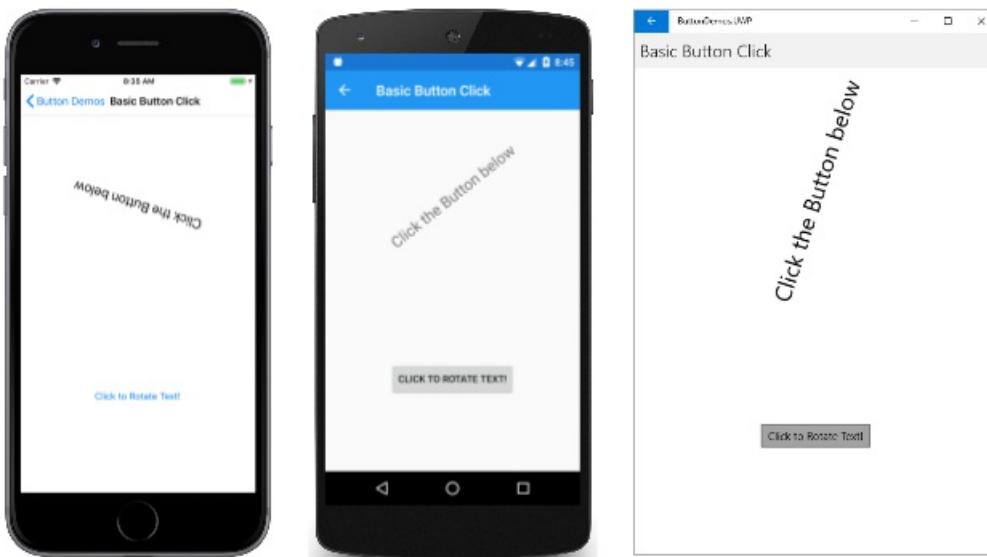
public partial class BasicButtonClickPage : ContentPage
{
    public BasicButtonClickPage ()
    {
        InitializeComponent ();
    }

    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 `OnButtonClicked` 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>

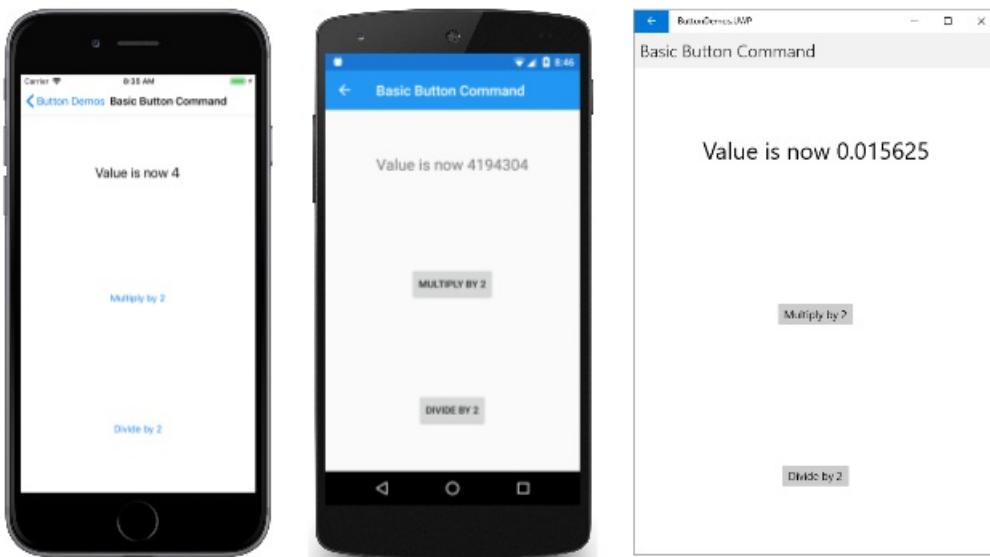
    <StackLayout>
        <Label Text="{Binding Number, StringFormat='Value is now {0}'}"
            FontSize="Large"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center" />

        <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:

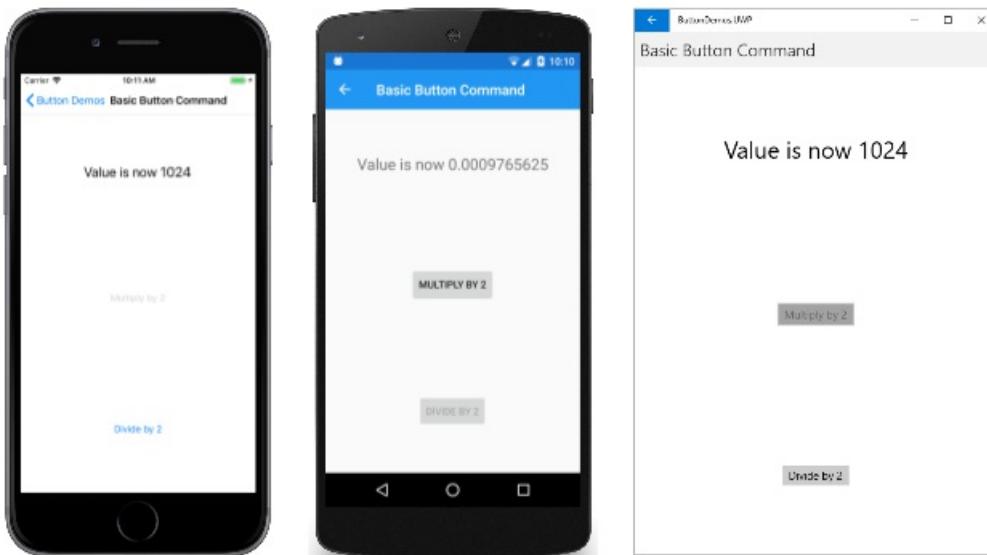
```

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;

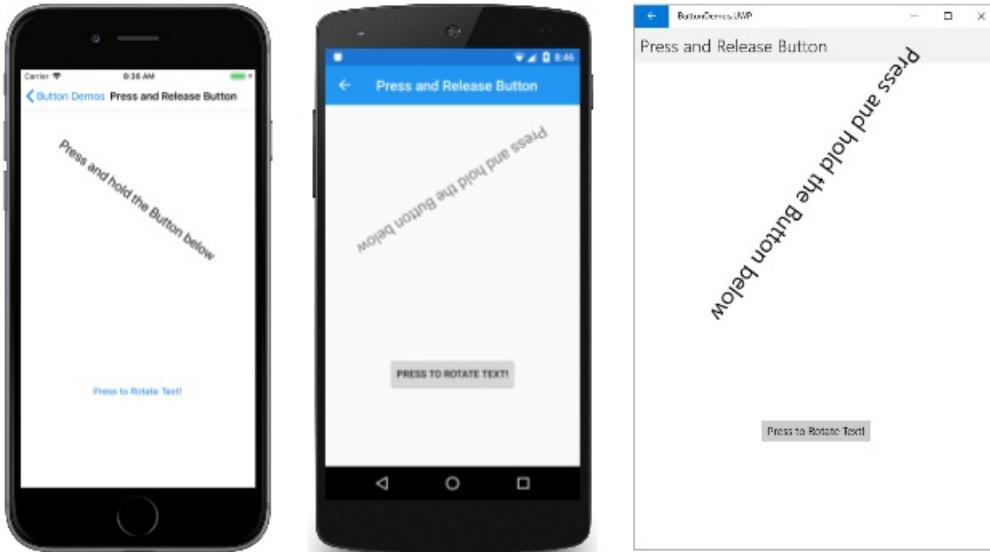
        Device.StartTimer(TimeSpan.FromMilliseconds(16), () =>
        {
            label.Rotation = 360 * (stopwatch.Elapsed.TotalSeconds % 1);

            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`
- `CharacterSpacing` is the spacing between characters of the `Button` text.
- `TextTransform` determines the casing of the `Button` text.

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}" />
    
```

```

    value={Binding CornerRadius} />

<Label Text="{Binding Source={x:Reference cornerRadiusSlider},
                    Path=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" />

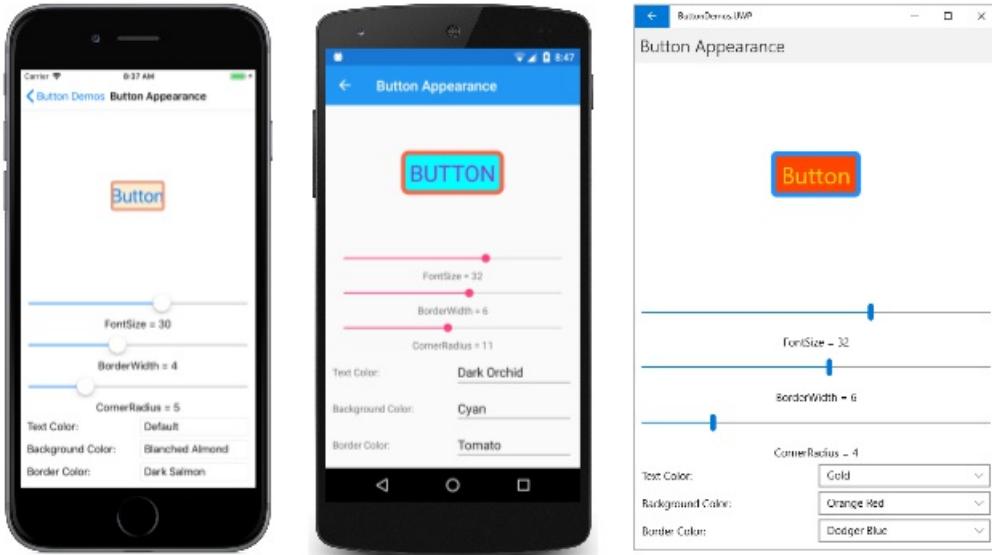
    <Label Text="Border Color:" Grid.Row="2" Grid.Column="0" />

    <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>
</Button>
```

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>

```

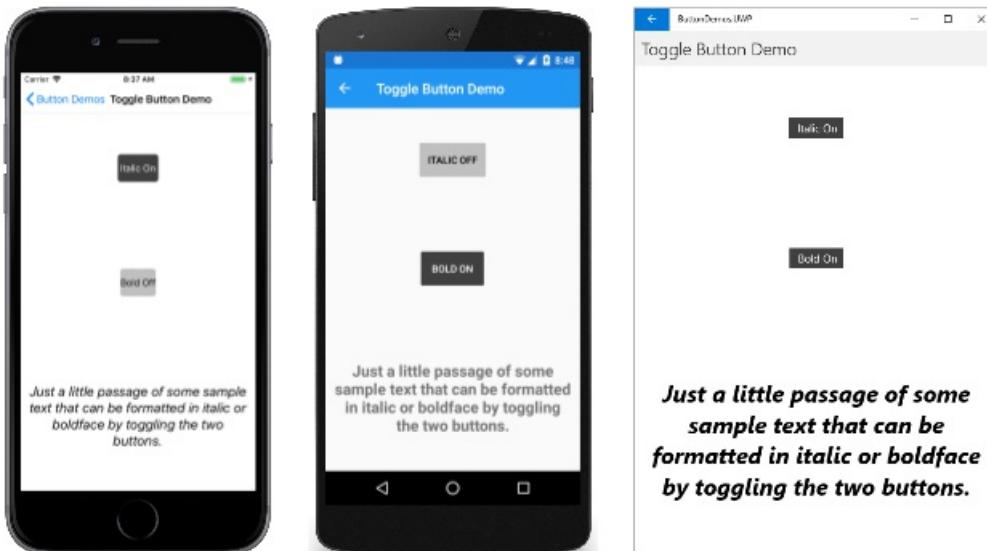
The `Toggled` event handlers are in the code-behind file. They are responsible for setting the `FontAttributes` property of the `Label` based on the state of the buttons:

```
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 `ImageSource` 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 `ImageSource` property is of type `ImageSource`, which means that the bitmaps can be loaded from a file,

embedded resource, URI, or stream.

NOTE

While a `Button` can load an animated GIF, it will only display the first frame of the GIF.

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 `ImageSource` 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",
    ImageSource = 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"
    ImageSource="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.ImageSource>
        <OnPlatform x:TypeArguments="ImageSource">
            <On Platform="iOS, Android" Value="MonkeyFace.png" />
            <On Platform="UWP" Value="Assets/MonkeyFace.png" />
        </OnPlatform>
    </Button.ImageSource>
</Button>
```

If the UWP bitmaps are stored in the root directory of the project, this markup can be considerably simplified:

```
<Button ImageSource="MonkeyFace.png" />
```

To avoid a lot of repetitious markup in the `ImageButtonDemo.xaml` file, an implicit `style` is also defined to set the `ImageSource` 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="ImageSource">
                    <OnPlatform x:TypeArguments="ImageSource">
                        <On Platform="iOS, Android" Value="MonkeyFace.png" />
                        <On Platform="UWP" Value="Assets/MonkeyFace.png" />
                    </OnPlatform>
                </Setter>
            </Style>
        </FlexLayout.Resources>

        <Button>
            <Button.ImageSource>
                <OnPlatform x:TypeArguments="ImageSource">
                    <On Platform="iOS, Android" Value="MonkeyFace.png" />
                    <On Platform="UWP" Value="Assets/MonkeyFace.png" />
                </OnPlatform>
            </Button.ImageSource>
        </Button>

        <Button Text="Default" />

        <Button Text="Left - 10"
            ContentLayout="Left, 10" />

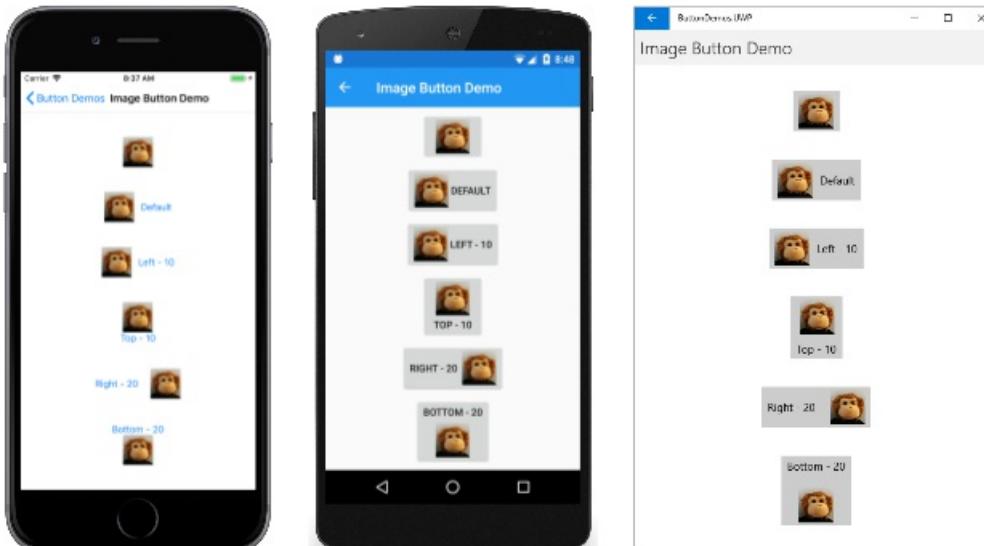
        <Button Text="Top - 10"
            ContentLayout="Top, 10" />

        <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](#)

Xamarin.Forms ImageButton

8/4/2022 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

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](#).

NOTE

While an `ImageButton` can load an animated GIF, it will only display the first frame of the GIF.

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:

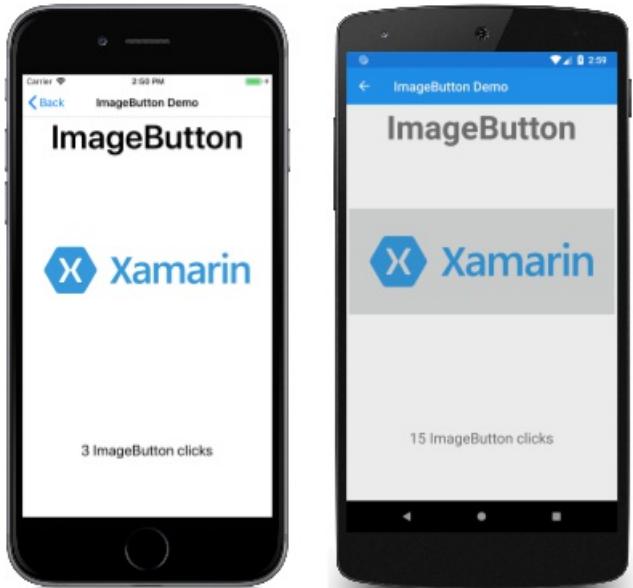
```
public partial class ImageButtonDemoPage : ContentPage
{
    int clickTotal;

    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 imageView = new ImageButton
        {
            Source = "XamarinLogo.png",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };
        imageView.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,
                imageView,
                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 `Enabled` 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](#)

Xamarin.Forms RadioButton

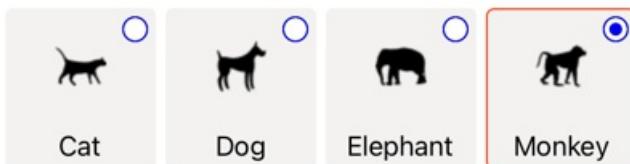
8/4/2022 • 10 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `RadioButton` is a type of button that allows users to select one option from a set. Each option is represented by one radio button, and you can only select one radio button in a group. By default, each `RadioButton` displays text:

- Cat
- Dog
- Elephant
- Monkey

However, on some platforms a `RadioButton` can display a `View`, and on all platforms the appearance of each `RadioButton` can be redefined with a `ControlTemplate`:



The `RadioButton` control defines the following properties:

- `Content`, of type `object`, which defines the `string` or `View` to be displayed by the `RadioButton`.
- `IsChecked`, of type `bool`, which defines whether the `RadioButton` is checked. This property uses a `TwoWay` binding, and has a default value of `false`.
- `GroupName`, of type `string`, which defines the name that specifies which `RadioButton` controls are mutually exclusive. This property has a default value of `null`.
- `Value`, of type `object`, which defines an optional unique value associated with the `RadioButton`.
- `BorderColor`, of type `Color`, which defines the border stroke color.
- `BorderWidth`, of type `double`, which defines the width of the `RadioButton` border.
- `CharacterSpacing`, of type `double`, which defines the spacing between characters of any displayed text.
- `CornerRadius`, of type `int`, which defines the corner radius of the `RadioButton`.
- `FontAttributes`, of type `FontAttributes`, which determines text style.
- `FontFamily`, of type `string`, which defines the font family.
- `FontSize`, of type `double`, which defines the font size.
- `TextColor`, of type `Color`, which defines the color of any displayed text.
- `TextTransform`, of type `TextTransform`, which defines the casing of any displayed text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `RadioButton` control also defines a `CheckedChanged` event that's fired when the `.IsChecked` property changes, either through user or programmatic manipulation. The `CheckedChangedEventArgs` object that accompanies the `CheckedChanged` event has a single property named `Value`, of type `bool`. When the event is fired, the value of the `CheckedChangedEventArgs.Value` property is set to the new value of the `.IsChecked`

property.

`RadioButton` grouping can be managed by the `RadioButtonGroup` class, which defines the following attached properties:

- `GroupName`, of type `string`, which defines the group name for `RadioButton` objects in a `Layout<View>`.
- `SelectedValue`, of type `object`, which represents the value of the checked `RadioButton` object within a `Layout<View>` group. This attached property uses a `TwoWay` binding by default.

For more information about the `GroupName` attached property, see [Group RadioButtons](#). For more information about the `SelectedValue` attached property, see [Respond to RadioButton state changes](#).

Create RadioButtons

The appearance of a `RadioButton` is defined by the type of data assigned to the `RadioButton.Content` property:

- When the `RadioButton.Content` property is assigned a `string`, it will be displayed on each platform, horizontally aligned next to the radio button circle.
- When the `RadioButton.Content` is assigned a `View`, it will be displayed on supported platforms (iOS, UWP), while unsupported platforms will fallback to a string representation of the `View` object (Android). In both cases, the content is displayed horizontally aligned next to the radio button circle.
- When a `ControlTemplate` is applied to a `RadioButton`, a `View` can be assigned to the `RadioButton.Content` property on all platforms. For more information, see [Redefine RadioButton appearance](#).

Display string-based content

A `RadioButton` displays text when the `Content` property is assigned a `string`:

```
<StackLayout>
    <Label Text="What's your favorite animal?" />
    <RadioButton Content="Cat" />
    <RadioButton Content="Dog" />
    <RadioButton Content="Elephant" />
    <RadioButton Content="Monkey"
        IsChecked="true" />
</StackLayout>
```

In this example, `RadioButton` objects are implicitly grouped inside the same parent container. This XAML results in the appearance shown in the following screenshots:

What's your favorite animal?

- Cat
- Dog
- Elephant
- Monkey

Display arbitrary content

On iOS and UWP, a `RadioButton` can display arbitrary content when the `Content` property is assigned a `View`:

```

<StackLayout>
    <Label Text="What's your favorite animal?" />
    <RadioButton>
        <RadioButton.Content>
            <Image Source="cat.png" />
        </RadioButton.Content>
    </RadioButton>
    <RadioButton>
        <RadioButton.Content>
            <Image Source="dog.png" />
        </RadioButton.Content>
    </RadioButton>
    <RadioButton>
        <RadioButton.Content>
            <Image Source="elephant.png" />
        </RadioButton.Content>
    </RadioButton>
    <RadioButton>
        <RadioButton.Content>
            <Image Source="monkey.png" />
        </RadioButton.Content>
    </RadioButton>
</StackLayout>

```

In this example, `RadioButton` objects are implicitly grouped inside the same parent container. This XAML results in the appearance shown in the following screenshots:

What's your favorite animal?

- 
- 
- 
- 

On Android, `RadioButton` objects will display a string-based representation of the `View` object that's been set as content:

- `Xamarin.Forms.Image`

NOTE

When a `ControlTemplate` is applied to a `RadioButton`, a `View` can be assigned to the `RadioButton.Content` property on all platforms. For more information, see [Redefine RadioButton appearance](#).

Associate values with RadioButtons

Each `RadioButton` object has a `Value` property, of type `object`, which defines an optional unique value to associate with the radio button. This enables the value of a `RadioButton` to be different to its content, and is particularly useful when `RadioButton` objects are displaying `View` objects.

The following XAML shows setting the `Content` and `Value` properties on each `RadioButton` object:

```

<StackLayout>
    <Label Text="What's your favorite animal?" />
    <RadioButton Value="Cat">
        <RadioButton.Content>
            <Image Source="cat.png" />
        </RadioButton.Content>
    </RadioButton>
    <RadioButton Value="Dog">
        <RadioButton.Content>
            <Image Source="dog.png" />
        </RadioButton.Content>
    </RadioButton>
    <RadioButton Value="Elephant">
        <RadioButton.Content>
            <Image Source="elephant.png" />
        </RadioButton.Content>
    </RadioButton>
    <RadioButton Value="Monkey">
        <RadioButton.Content>
            <Image Source="monkey.png" />
        </RadioButton.Content>
    </RadioButton>
</StackLayout>

```

In this example, each `RadioButton` has an `Image` as its content, while also defining a string-based value. This enables the value of the checked radio button to be easily identified.

Group RadioButtons

Radio buttons work in groups, and there are three approaches to grouping radio buttons:

- Place them inside the same parent container. This is known as *implicit* grouping.
- Set the `GroupName` property on each radio button in the group to the same value. This is known as *explicit* grouping.
- Set the `RadioButtonGroup.GroupName` attached property on a parent container, which in turn sets the `GroupName` property of any `RadioButton` objects in the container. This is also known as *explicit* grouping.

IMPORTANT

`RadioButton` objects don't have to belong to the same parent to be grouped. They are mutually exclusive provided that they share a group name.

Explicit grouping with the GroupName property

The following XAML example shows explicitly grouping `RadioButton` objects by setting their `GroupName` properties:

```

<Label Text="What's your favorite color?" />
<RadioButton Content="Red"
            GroupName="colors" />
<RadioButton Content="Green"
            GroupName="colors" />
<RadioButton Content="Blue"
            GroupName="colors" />
<RadioButton Content="Other"
            GroupName="colors" />

```

In this example, each `RadioButton` is mutually exclusive because it shares the same `GroupName` value.

Explicit grouping with the RadioButtonGroup.GroupName attached property

The `RadioButtonGroup` class defines a `GroupName` attached property, of type `string`, which can be set on a `Layout<View>` object. This enables any layout to be turned into a radio button group:

```
<StackLayout RadioButtonGroup.GroupName="colors">
    <Label Text="What's your favorite color?" />
    <RadioButton Content="Red" />
    <RadioButton Content="Green" />
    <RadioButton Content="Blue" />
    <RadioButton Content="Other" />
</StackLayout>
```

In this example, each `RadioButton` in the `StackLayout` will have its `GroupName` property set to `colors`, and will be mutually exclusive.

NOTE

When a `Layout<View>` object that sets the `RadioButtonGroup.GroupName` attached property contains a `RadioButton` that sets its `GroupName` property, the value of the `RadioButton.GroupName` property will take precedence.

Respond to RadioButton state changes

A radio button has two states: checked or unchecked. When a radio button is checked, its `.IsChecked` property is `true`. When a radio button is unchecked, its `.IsChecked` property is `false`. A radio button can be cleared by tapping another radio button in the same group, but it cannot be cleared by tapping it again. However, you can clear a radio button programmatically by setting its `.IsChecked` property to `false`.

Respond to an event firing

When the `.IsChecked` property changes, either through user or programmatic manipulation, the `CheckedChanged` event fires. An event handler for this event can be registered to respond to the change:

```
<RadioButton Content="Red"
            GroupName="colors"
            CheckedChanged="OnColorsRadioButtonCheckedChanged" />
```

The code-behind contains the handler for the `CheckedChanged` event:

```
void OnColorsRadioButtonCheckedChanged(object sender, CheckedChangedEventArgs e)
{
    // Perform required operation
}
```

The `sender` argument is the `RadioButton` responsible for this event. You can use this to access the `RadioButton` object, or to distinguish between multiple `RadioButton` objects sharing the same `CheckedChanged` event handler.

Respond to a property change

The `RadioButtonGroup` class defines a `SelectedValue` attached property, of type `object`, which can be set on a `Layout<View>` object. This attached property represents the value of the checked `RadioButton` within a group defined on a layout.

When the `.IsChecked` property changes, either through user or programmatic manipulation, the `RadioButtonGroup.SelectedValue` attached property also changes. Therefore, the `RadioButtonGroup.SelectedValue` attached property can be data bound to a property that stores the user's selection:

```

<StackLayout RadioButtonGroup.GroupName="{Binding GroupName}"
            RadioButtonGroup.SelectedValue="{Binding Selection}">
    <Label Text="What's your favorite animal?" />
    <RadioButton Content="Cat"
                Value="Cat" />
    <RadioButton Content="Dog"
                Value="Dog" />
    <RadioButton Content="Elephant"
                Value="Elephant" />
    <RadioButton Content="Monkey"
                Value="Monkey"/>
    <Label x:Name="animalLabel">
        <Label.FormattedText>
            <FormattedString>
                <Span Text="You have chosen:" />
                <Span Text="{Binding Selection}" />
            </FormattedString>
        </Label.FormattedText>
    </Label>
</StackLayout>

```

In this example, the value of the `RadioButtonGroup.GroupName` attached property is set by the `GroupName` property on the binding context. Similarly, the value of the `RadioButtonGroup.SelectedValue` attached property is set by the `Selection` property on the binding context. In addition, the `Selection` property is updated to the `Value` property of the checked `RadioButton`.

RadioButton visual states

`RadioButton` objects have `Checked` and `Unchecked` visual states that can be used to initiate a visual change when a `RadioButton` is checked or unchecked.

The following XAML example shows how to define a visual state for the `Checked` and `Unchecked` states:

```

<ContentPage ...>
    <ContentPage.Resources>
        <Style TargetType="RadioButton">
            <Setter Property="VisualStateManager.VisualStateGroups">
                <VisualStateGroupList>
                    <VisualStateGroup x:Name="CheckedStates">
                        <VisualState x:Name="Checked">
                            <VisualState.Setters>
                                <Setter Property="TextColor"
                                       Value="Green" />
                                <Setter Property="Opacity"
                                       Value="1" />
                            </VisualState.Setters>
                        </VisualState>
                        <VisualState x:Name="Unchecked">
                            <VisualState.Setters>
                                <Setter Property="TextColor"
                                       Value="Red" />
                                <Setter Property="Opacity"
                                       Value="0.5" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
                </VisualStateGroupList>
            </Setter>
        </Style>
    </ContentPage.Resources>
    <StackLayout>
        <Label Text="What's your favorite mode of transport?" />
        <RadioButton Content="Car" />
        <RadioButton Content="Bike" />
        <RadioButton Content="Train" />
        <RadioButton Content="Walking" />
    </StackLayout>
</ContentPage>

```

In this example, the implicit `Style` targets `RadioButton` objects. The `Checked` `VisualState` specifies that when a `RadioButton` is checked, its `TextColor` property will be set to green with an `Opacity` value of 1. The `Unchecked` `VisualState` specifies that when a `RadioButton` is in an unchecked state, its `TextColor` property will be set to red with an `Opacity` value of 0.5. Therefore, the overall effect is that when a `RadioButton` is unchecked it's red and partially transparent, and is green without transparency when it's checked:

What's your favorite mode of transport?

- Car
- Bike
- Train
- Walking

For more information about visual states, see [Xamarin.Forms Visual State Manager](#).

Redefine RadioButton appearance

By default, `RadioButton` objects use platform renderers to utilize native controls on supported platforms. However, `RadioButton` visual structure can be redefined with a `ControlTemplate`, so that `RadioButton` objects have an identical appearance on all platforms. This is possible because the `RadioButton` class inherits from the `TemplatedView` class.

The following XAML shows a `ControlTemplate` that can be used to redefine the visual structure of `RadioButton` objects:

```

<ContentPage ...>
    <ContentPage.Resources>
        <ControlTemplate x:Key="RadioButtonTemplate">
            <Frame BorderColor="#F3F2F1"
                BackgroundColor="#F3F2F1"
                HasShadow="False"
                HeightRequest="100"
                WidthRequest="100"
                HorizontalOptions="Start"
                VerticalOptions="Start"
                Padding="0">
                <VisualStateManager.VisualStateGroups>
                    <VisualStateGroupList>
                        <VisualStateGroup x:Name="CheckedStates">
                            <VisualState x:Name="Checked">
                                <VisualState.Setters>
                                    <Setter Property="BorderColor"
                                        Value="#FF3300" />
                                    <Setter TargetName="check"
                                        Property="Opacity"
                                        Value="1" />
                                </VisualState.Setters>
                            </VisualState>
                            <VisualState x:Name="Unchecked">
                                <VisualState.Setters>
                                    <Setter Property="BackgroundColor"
                                        Value="#F3F2F1" />
                                    <Setter Property="BorderColor"
                                        Value="#F3F2F1" />
                                    <Setter TargetName="check"
                                        Property="Opacity"
                                        Value="0" />
                                </VisualState.Setters>
                            </VisualState>
                        </VisualStateGroup>
                    </VisualStateGroupList>
                </VisualStateManager.VisualStateGroups>
                <Grid Margin="4"
                    WidthRequest="100">
                    <Grid WidthRequest="18"
                        HeightRequest="18"
                        HorizontalOptions="End"
                        VerticalOptions="Start">
                        <Ellipse Stroke="Blue"
                            Fill="White"
                            WidthRequest="16"
                            HeightRequest="16"
                            HorizontalOptions="Center"
                            VerticalOptions="Center" />
                        <Ellipse x:Name="check"
                            Fill="Blue"
                            WidthRequest="8"
                            HeightRequest="8"
                            HorizontalOptions="Center"
                            VerticalOptions="Center" />
                    </Grid>
                    <ContentPresenter />
                </Grid>
            </Frame>
        </ControlTemplate>

        <Style TargetType="RadioButton">
            <Setter Property="ControlTemplate"
                Value="{StaticResource RadioButtonTemplate}" />
        </Style>
    </ContentPage.Resources>
    <!-- Page content -->
</ContentPage>

```

In this example, the root element of the `ControlTemplate` is a `Frame` object that defines `Checked` and `Unchecked` visual states. The `Frame` object uses a combination of `Grid`, `Ellipse`, and `ContentPresenter` objects to define the visual structure of a `RadioButton`. The example also includes an *implicit* style that will assign the `RadioButtonTemplate` to the `ControlTemplate` property of any `RadioButton` objects on the page.

NOTE

The `ContentPresenter` object marks the location in the visual structure where `RadioButton` content will be displayed.

The following XAML shows `RadioButton` objects that consume the `ControlTemplate` via the *implicit* style:

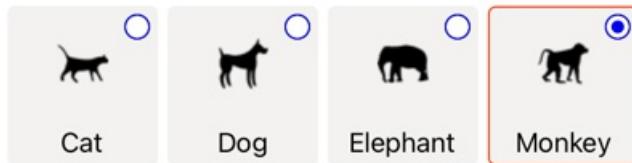
```

<StackLayout>
    <Label Text="What's your favorite animal?" />
    <StackLayout RadioButtonGroup.GroupName="animals"
        Orientation="Horizontal">
        <RadioButton Value="Cat">
            <RadioButton.Content>
                <StackLayout>
                    <Image Source="cat.png"
                        HorizontalOptions="Center"
                        VerticalOptions="CenterAndExpand" />
                    <Label Text="Cat"
                        HorizontalOptions="Center"
                        VerticalOptions="End" />
                </StackLayout>
            </RadioButton.Content>
        </RadioButton>
        <RadioButton Value="Dog">
            <RadioButton.Content>
                <StackLayout>
                    <Image Source="dog.png"
                        HorizontalOptions="Center"
                        VerticalOptions="CenterAndExpand" />
                    <Label Text="Dog"
                        HorizontalOptions="Center"
                        VerticalOptions="End" />
                </StackLayout>
            </RadioButton.Content>
        </RadioButton>
        <RadioButton Value="Elephant">
            <RadioButton.Content>
                <StackLayout>
                    <Image Source="elephant.png"
                        HorizontalOptions="Center"
                        VerticalOptions="CenterAndExpand" />
                    <Label Text="Elephant"
                        HorizontalOptions="Center"
                        VerticalOptions="End" />
                </StackLayout>
            </RadioButton.Content>
        </RadioButton>
        <RadioButton Value="Monkey">
            <RadioButton.Content>
                <StackLayout>
                    <Image Source="monkey.png"
                        HorizontalOptions="Center"
                        VerticalOptions="CenterAndExpand" />
                    <Label Text="Monkey"
                        HorizontalOptions="Center"
                        VerticalOptions="End" />
                </StackLayout>
            </RadioButton.Content>
        </RadioButton>
    </StackLayout>

```

In this example, the visual structure defined for each `RadioButton` is replaced with the visual structure defined in the `ControlTemplate`, and so at runtime the objects in the `ControlTemplate` become part of the visual tree for each `RadioButton`. In addition, the content for each `RadioButton` is substituted into the `ContentPresenter` defined in the control template. This results in the following `RadioButton` appearance:

What's your favorite animal?



For more information about control templates, see [Xamarin.Forms control templates](#).

Disable a RadioButton

Sometimes an application enters a state where a `RadioButton` being checked is not a valid operation. In such cases, the `RadioButton` can be disabled by setting its `IsEnabled` property to `false`.

Related links

- [RadioButton Demos \(sample\)](#)
- [Xamarin.Forms Button](#)
- [Xamarin.Forms Visual State Manager](#)

Xamarin.Forms RefreshView

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `RefreshView` is a container control that provides pull to refresh functionality for scrollable content. Therefore, the child of a `RefreshView` must be a scrollable control, such as `ScrollView`, `CollectionView`, or `ListView`.

`RefreshView` defines the following properties:

- `Command`, of type `ICommand`, which is executed when a refresh is triggered.
- `CommandParameter`, of type `object`, which is the parameter that's passed to the `Command`.
- `IsRefreshing`, of type `bool`, which indicates the current state of the `RefreshView`.
- `RefreshColor`, of type `Color`, the color of the progress circle that appears during the refresh.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

NOTE

On the Universal Windows Platform, the pull direction of a `RefreshView` can be set with a platform-specific. For more information, see [RefreshView Pull Direction](#).

Create a RefreshView

The following example shows how to instantiate a `RefreshView` in XAML:

```
<RefreshView IsRefreshing="{Binding IsRefreshing}"
             Command="{Binding RefreshCommand}">
    <ScrollView>
        <FlexLayout Direction="Row"
                    Wrap="Wrap"
                    AlignItems="Center"
                    AlignContent="Center"
                    BindableLayout.ItemsSource="{Binding Items}"
                    BindableLayout.ItemTemplate="{StaticResource ColorItemTemplate}" />
    </ScrollView>
</RefreshView>
```

A `RefreshView` can also be created in code:

```

RefreshView refreshView = new RefreshView();
 ICommand refreshCommand = new Command(() =>
{
    // IsRefreshing is true
    // Refresh data here
    refreshView.IsRefreshing = false;
});
refreshView.Command = refreshCommand;

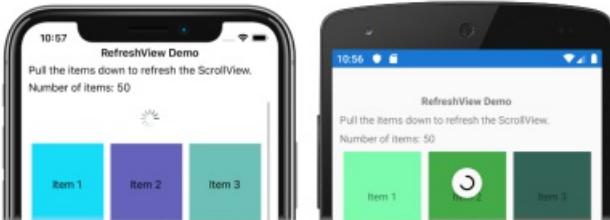
ScrollView scrollView = new ScrollView();
FlexLayout flexLayout = new FlexLayout { ... };
scrollView.Content = flexLayout;
refreshView.Content = scrollView;

```

In this example, the `RefreshView` provides pull to refresh functionality to a `ScrollView` whose child is a `FlexLayout`. The `FlexLayout` uses a bindable layout to generate its content by binding to a collection of items, and sets the appearance of each item with a `DataTemplate`. For more information about bindable layouts, see [Bindable Layouts in Xamarin.Forms](#).

The value of the `RefreshView.IsRefreshing` property indicates the current state of the `RefreshView`. When a refresh is triggered by the user, this property will automatically transition to `true`. Once the refresh completes, you should reset the property to `false`.

When the user initiates a refresh, the `ICommand` defined by the `Command` property is executed, which should refresh the items being displayed. A refresh visualization is shown while the refresh occurs, which consists of an animated progress circle:



NOTE

Manually setting the `IsRefreshing` property to `true` will trigger the refresh visualization, and will execute the `ICommand` defined by the `Command` property.

RefreshView appearance

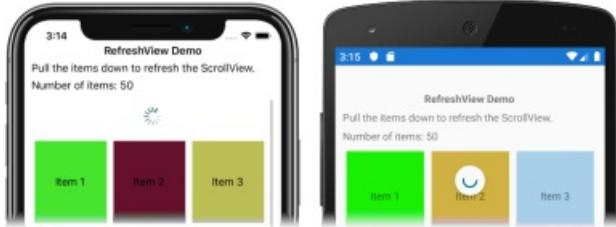
In addition to the properties that `RefreshView` inherits from the `VisualElement` class, `RefreshView` also defines the `RefreshColor` property. This property can be set to define the color of the progress circle that appears during the refresh:

```

<RefreshView RefreshColor="Teal"
            ... />

```

The following screenshot shows a `RefreshView` with the `RefreshColor` property set:



In addition, the `BackgroundColor` property can be set to a `Color` that represents the background color of the progress circle.

NOTE

On iOS, the `BackgroundColor` property sets the background color of the `UIView` that contains the progress circle.

Disable a RefreshView

An application may enter a state where pull to refresh is not a valid operation. In such cases, the `RefreshView` can be disabled by setting its `Enabled` property to `false`. This will prevent users from being able to trigger pull to refresh.

Alternatively, when defining the `Command` property, the `CanExecute` delegate of the `ICommand` can be specified to enable or disable the command.

Related links

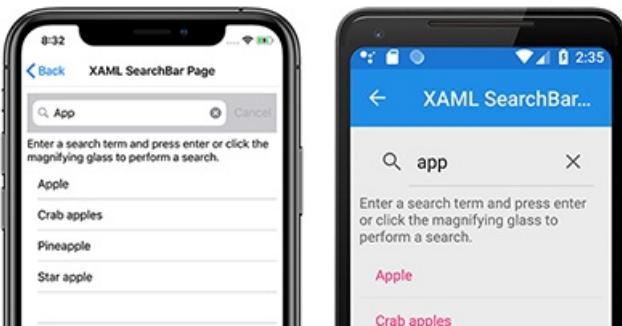
- [RefreshView \(sample\)](#)
- [Bindable Layouts in Xamarin.Forms](#)
- [RefreshView Pull Direction platform-specific](#)

Xamarin.Forms SearchBar

8/4/2022 • 4 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `SearchBar` is a user input control used to initiating a search. The `SearchBar` control supports placeholder text, query input, search execution, and cancellation. The following screenshot shows a `SearchBar` query with results displayed in a `ListView`:



The `SearchBar` class defines the following properties:

- `CancelButtonColor` is a `Color` that defines the color of the cancel button.
- `CharacterSpacing`, of type `double`, is the spacing between characters of the `SearchBar` text.
- `FontAttributes` is a `FontAttributes` enum value that determines whether the `SearchBar` font is bold, italic, or neither.
- `FontFamily` is a `string` that determines the font family used by the `SearchBar`.
- `FontSize` can be a `NamedSize` enum value or a `double` value that represents specific font sizes across platforms.
- `HorizontalTextAlignment` is a `TextAlignment` enum value that defines the horizontal alignment of the query text.
- `VerticalTextAlignment` is a `TextAlignment` enum value that defines the vertical alignment of the query text.
- `Placeholder` is a `string` that defines the placeholder text, such as "Search...".
- `PlaceholderColor` is a `Color` that defines the color of the placeholder text.
- `SearchCommand` is an `ICommand` that allows binding user actions, such as finger taps or clicks, to commands defined on a viewmodel.
- `SearchCommandParameter` is an `object` that specifies the parameter that should be passed to the `SearchCommand`.
- `Text` is a `string` containing the query text in the `SearchBar`.
- `TextColor` is a `Color` that defines the query text color.
- `TextTransform` is a `TextTransform` value that determines the casing of the `SearchBar` text.

These properties are backed by `BindableProperty` objects, which means the `SearchBar` can be customized and be the target of data bindings. Specifying font properties on the `SearchBar` is consistent with customizing text on other [Xamarin.Forms Text controls](#). For more information, see [Fonts in Xamarin.Forms](#).

Create a SearchBar

A `SearchBar` can be instantiated in XAML. Its optional `Placeholder` property can be set to define the hint text in

the query input box. The default value for the `Placeholder` is an empty string so no placeholder will appear if it isn't set. The following example shows how to instantiate a `SearchBar` in XAML with the optional `Placeholder` property set:

```
<SearchBar Placeholder="Search items..." />
```

A `SearchBar` can also be created in code:

```
SearchBar searchBar = new SearchBar{ Placeholder = "Search items..." };
```

SearchBar appearance properties

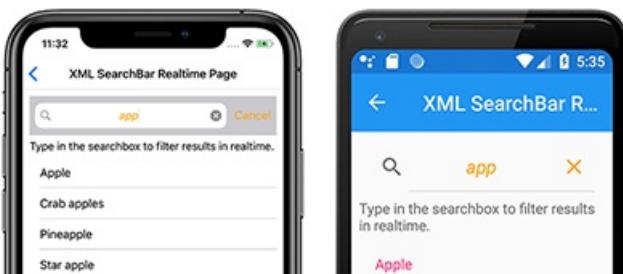
The `searchBar` control defines many properties that customize the appearance of the control. The following example shows how to instantiate a `SearchBar` in XAML with multiple properties specified:

```
<SearchBar Placeholder="Search items..."  
    CancelButtonColor="Orange"  
    PlaceholderColor="Orange"  
    TextColor="Orange"  
    TextTransform="Lowercase"  
    HorizontalTextAlignment="Center"  
    FontSize="Medium"  
    FontAttributes="Italic" />
```

These properties can also be specified when creating a `SearchBar` object in code:

```
SearchBar searchBar = new SearchBar  
{  
    Placeholder = "Search items...",  
    PlaceholderColor = Color.Orange,  
    TextColor = Color.Orange,  
    TextTransform = TextTransform.Lowercase,  
    HorizontalTextAlignment = TextAlignment.Center,  
    FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(SearchBar)),  
    FontAttributes = FontAttributes.Italic  
};
```

The following screenshot shows the resulting `SearchBar` control:



NOTE

On iOS, the `SearchBarRenderer` class contains an overridable `UpdateCancelButton` method. This method controls when the cancel button appears, and can be overridden in a custom renderer. For more information about custom renderers, see [Xamarin.Forms Custom Renderers](#).

Perform a search with event handlers

A search can be executed using the `SearchBar` control by attaching an event handler to one of the following events:

- `SearchButtonPressed` is called when the user either clicks the search button or presses the enter key.
- `TextChanged` is called anytime the text in the query box is changed.

The following example shows an event handler attached to the `TextChanged` event in XAML and uses a `ListView` to display search results:

```
<SearchBar TextChanged="OnTextChanged" />
<ListView x:Name="searchResults" >
```

An event handler can also be attached to a `SearchBar` created in code:

```
SearchBar searchBar = new SearchBar {/*...*/};
searchBar.TextChanged += OnTextChanged;
```

The `TextChanged` event handler in the code-behind file is the same, whether the `SearchBar` is created via XAML or code:

```
void OnTextChanged(object sender, EventArgs e)
{
    SearchBar searchBar = (SearchBar)sender;
    searchResults.ItemsSource = DataService.GetSearchResults(searchBar.Text);
}
```

The previous example implies the existence of a `DataService` class with a `GetSearchResults` method capable of returning items that match a query. The `SearchBar` control's `Text` property value is passed to the `GetSearchResults` method and the result is used to update the `ListView` control's `ItemsSource` property. The overall effect is that search results are displayed in the `ListView` control.

The sample application provides a `DataService` class implementation that can be used to test search functionality.

Perform a search using a viewmodel

A search can be executed without event handlers by binding the `SearchCommand` and `SearchCommandParameter` properties to `ICommand` implementations. The sample project demonstrates these implementations using the Model-View-ViewModel (MVVM) pattern. For more information about data bindings with MVVM, see [Data Bindings with MVVM](#).

The.viewmodel in the sample application contains the following code:

```

public class SearchViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void NotifyPropertyChanged([CallerMemberName] string propertyName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public ICommand PerformSearch => new Command<string>((string query) =>
    {
        SearchResults = DataService.GetSearchResults(query);
    });

    private List<string> searchResults = DataService.Fruits;
    public List<string> SearchResults
    {
        get
        {
            return searchResults;
        }
        set
        {
            searchResults = value;
            NotifyPropertyChanged();
        }
    }
}

```

NOTE

The viewmodel assumes the existence of a `DataService` class capable of performing searches. The `DataService` class, including example data, is available in the sample application.

The following XAML shows how to bind a `SearchBar` to the example viewmodel, with a `ListView` control displaying the search results:

```

<ContentPage ...>
    <ContentPage.BindingContext>
        <viewmodels:SearchViewModel />
    </ContentPage.BindingContext>
    <StackLayout ...>
        <SearchBar x:Name="searchBar"
            ...
            SearchCommand="{Binding PerformSearch}"
            SearchCommandParameter="{Binding Text, Source={x:Reference searchBar}}"/>
        <ListView x:Name="searchResults"
            ...
            ItemsSource="{Binding SearchResults}" />
    </StackLayout>
</ContentPage>

```

This example sets the `BindingContext` to be an instance of the `SearchViewModel` class. It binds the `SearchCommand` property to the `PerformSearch` `ICommand` in the.viewmodel, and binds the `SearchBar` `Text` property to the `SearchCommandParameter` property. The `ListView.ItemsSource` property is bound to the `SearchResults` property of the.viewmodel.

For more information about the `ICommand` Interface and bindings, see [Xamarin.Forms data binding](#) and the [ICommand interface](#).

Related links

- [SearchBar Demos](#)
- [Xamarin.Forms Text controls](#)
- [Fonts in Xamarin.Forms](#)
- [Xamarin.Forms data binding](#)

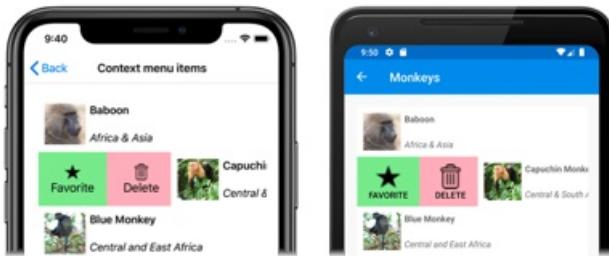
Xamarin.Forms SwipeView

8/4/2022 • 10 minutes to read • [Edit Online](#)



[Download the sample](#)

The `SwipeView` is a container control that wraps around an item of content, and provides context menu items that are revealed by a swipe gesture:



`SwipeView` defines the following properties:

- `LeftItems`, of type `SwipeItems`, which represents the swipe items that can be invoked when the control is swiped from the left side.
- `RightItems`, of type `SwipeItems`, which represents the swipe items that can be invoked when the control is swiped from the right side.
- `TopItems`, of type `SwipeItems`, which represents the swipe items that can be invoked when the control is swiped from the top down.
- `BottomItems`, of type `SwipeItems`, which represents the swipe items that can be invoked when the control is swiped from the bottom up.
- `Threshold`, of type `double`, which represents the number of device-independent units that trigger a swipe gesture to fully reveal swipe items.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

In addition, the `SwipeView` inherits the `Content` property from the `ContentView` class. The `Content` property is the content property of the `SwipeView` class, and therefore does not need to be explicitly set.

The `SwipeView` class also defines three events:

- `SwipeStarted` is fired when a swipe starts. The `SwipeStartedEventArgs` object that accompanies this event has a `SwipeDirection` property, of type `SwipeDirection`.
- `SwipeChanging` is fired as the swipe moves. The `SwipeChangingEventArgs` object that accompanies this event has a `SwipeDirection` property, of type `SwipeDirection`, and an `Offset` property of type `double`.
- `SwipeEnded` is fired when a swipe ends. The `SwipeEndedEventArgs` object that accompanies this event has a `SwipeDirection` property, of type `SwipeDirection`, and an `IsOpen` property of type `bool`.

In addition, `SwipeView` includes `open` and `close` methods, which programmatically open and close the swipe items, respectively.

NOTE

`SwipeView` has a platform-specific on iOS and Android, that controls the transition that's used when opening a `SwipeView`. For more information, see [SwipeView Swipe Transition Mode on iOS](#) and [SwipeView Swipe Transition Mode on Android](#).

Create a SwipeView

A `SwipeView` must define the content that the `SwipeView` wraps around, and the swipe items that are revealed by the swipe gesture. The swipe items are one or more `SwipeItem` objects that are placed in one of the four `SwipeView` directional collections - `LeftItems`, `RightItems`, `TopItems`, or `BottomItems`.

The following example shows how to instantiate a `SwipeView` in XAML:

```
<SwipeView>
    <SwipeView.LeftItems>
        <SwipeItems>
            <SwipeItem Text="Favorite"
                IconImageSource="favorite.png"
                BackgroundColor="LightGreen"
                Invoked="OnFavoriteSwipeItemInvoked" />
            <SwipeItem Text="Delete"
                IconImageSource="delete.png"
                BackgroundColor="LightPink"
                Invoked="OnDeleteSwipeItemInvoked" />
        </SwipeItems>
    </SwipeView.LeftItems>
    <!-- Content -->
    <Grid HeightRequest="60"
        WidthRequest="300"
        BackgroundColor="LightGray">
        <Label Text="Swipe right"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
    </Grid>
</SwipeView>
```

The equivalent C# code is:

```

// SwipeItems
SwipeItem favoriteSwipeItem = new SwipeItem
{
    Text = "Favorite",
    IconImageSource = "favorite.png",
    BackgroundColor = Color.LightGreen
};
favoriteSwipeItem.Invoked += OnFavoriteSwipeItemInvoked;

SwipeItem deleteSwipeItem = new SwipeItem
{
    Text = "Delete",
    IconImageSource = "delete.png",
    BackgroundColor = Color.LightPink
};
deleteSwipeItem.Invoked += OnDeleteSwipeItemInvoked;

List<SwipeItem> swipeItems = new List<SwipeItem>() { favoriteSwipeItem, deleteSwipeItem };

// SwipeView content
Grid grid = new Grid
{
    HeightRequest = 60,
    WidthRequest = 300,
    BackgroundColor = Color.LightGray
};
grid.Children.Add(new Label
{
    Text = "Swipe right",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center
});

SwipeView swipeView = new SwipeView
{
    LeftItems = new SwipeItems(swipeItems),
    Content = grid
};

```

In this example, the `SwipeView` content is a `Grid` that contains a `Label`:



The swipe items are used to perform actions on the `SwipeView` content, and are revealed when the control is swiped from the left side:



By default, a swipe item is executed when it is tapped by the user. However, this behavior can be changed. For more information, see [Swipe mode](#).

Once a swipe item has been executed the swipe items are hidden and the `SwipeView` content is re-displayed. However, this behavior can be changed. For more information, see [Swipe behavior](#).

NOTE

Swipe content and swipe items can be placed inline, or defined as resources.

Swipe items

The `LeftItems`, `RightItems`, `TopItems`, and `BottomItems` collections are all of type `SwipeItems`. The `SwipeItems` class defines the following properties:

- `Mode`, of type `SwipeMode`, which indicates the effect of a swipe interaction. For more information about swipe mode, see [Swipe mode](#).
- `SwipeBehaviorOnInvoked`, of type `SwipeBehaviorOnInvoked`, which indicates how a `SwipeView` behaves after a swipe item is invoked. For more information about swipe behavior, see [Swipe behavior](#).

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

Each swipe item is defined as a `SwipeItem` object that's placed into one of the four `SwipeItems` directional collections. The `SwipeItem` class derives from the `MenuItem` class, and adds the following members:

- A `BackgroundColor` property, of type `Color`, that defines the background color of the swipe item. This property is backed by a bindable property.
- An `Invoked` event, which is fired when the swipe item is executed.

IMPORTANT

The `MenuItem` class defines several properties, including `Command`, `CommandParameter`, `IconImageSource`, and `Text`. These properties can be set on a `SwipeItem` object to define its appearance, and to define an `ICommand` that executes when the swipe item is invoked. For more information, see [Xamarin.Forms MenuItem](#).

The following example shows two `SwipeItem` objects in the `LeftItems` collection of a `SwipeView`:

```
<SwipeView>
    <SwipeView.LeftItems>
        <SwipeItems>
            <SwipeItem Text="Favorite"
                IconImageSource="favorite.png"
                BackgroundColor="LightGreen"
                Invoked="OnFavoriteSwipeItemInvoked" />
            <SwipeItem Text="Delete"
                IconImageSource="delete.png"
                BackgroundColor="LightPink"
                Invoked="OnDeleteSwipeItemInvoked" />
        </SwipeItems>
    </SwipeView.LeftItems>
    <!-- Content -->
</SwipeView>
```

The appearance of each `SwipeItem` is defined by a combination of the `Text`, `IconImageSource`, and `BackgroundColor` properties:



When a `SwipeItem` is tapped, its `Invoked` event fires and is handled by its registered event handler. In addition, the `MenuItem.Clicked` event fires. Alternatively, the `Command` property can be set to an `ICommand` implementation that will be executed when the `SwipeItem` is invoked.

NOTE

When the appearance of a `SwipeItem` is defined only using the `Text` or `IconImageSource` properties, the content is always centered.

In addition to defining swipe items as `SwipeItem` objects, it's also possible to define custom swipe item views. For more information, see [Custom swipe items](#).

Swipe direction

`SwipeView` supports four different swipe directions, with the swipe direction being defined by the directional `SwipeItems` collection the `SwipeItem` objects are added to. Each swipe direction can hold its own swipe items. For example, the following example shows a `SwipeView` whose swipe items depend on the swipe direction:

```
<SwipeView>
    <SwipeView.LeftItems>
        <SwipeItems>
            <SwipeItem Text="Delete"
                IconImageSource="delete.png"
                BackgroundColor="LightPink"
                Command="{Binding DeleteCommand}" />
        </SwipeItems>
    </SwipeView.LeftItems>
    <SwipeView.RightItems>
        <SwipeItems>
            <SwipeItem Text="Favorite"
                IconImageSource="favorite.png"
                BackgroundColor="LightGreen"
                Command="{Binding FavoriteCommand}" />
            <SwipeItem Text="Share"
                IconImageSource="share.png"
                BackgroundColor="LightYellow"
                Command="{Binding ShareCommand}" />
        </SwipeItems>
    </SwipeView.RightItems>
    <!-- Content -->
</SwipeView>
```

In this example, the `Swipeview` content can be swiped right or left. Swiping to the right will show the **Delete** swipe item, while swiping to the left will show the **Favorite** and **Share** swipe items.

WARNING

Only one instance of a directional `SwipeItems` collection can be set at a time on a `SwipeView`. Therefore, you cannot have two `LeftItems` definitions on a `SwipeView`.

The `SwipeStarted`, `SwipeChanging`, and `SwipeEnded` events report the swipe direction via the `SwipeDirection` property in the event arguments. This property is of type `SwipeDirection`, which is an enumeration consisting of four members:

- `Right` indicates that a right swipe occurred.
- `Left` indicates that a left swipe occurred.

- `Up` indicates that an upwards swipe occurred.
- `Down` indicates that a downwards swipe occurred.

Swipe threshold

`SwipeView` includes a `Threshold` property, of type `double`, which represents the number of device-independent units that trigger a swipe gesture to fully reveal swipe items.

The following example shows a `SwipeView` that sets the `Threshold` property:

```
<SwipeView Threshold="200">
    <SwipeView.LeftItems>
        <SwipeItems>
            <SwipeItem Text="Favorite"
                IconImageSource="favorite.png"
                BackgroundColor="LightGreen" />
        </SwipeItems>
    </SwipeView.LeftItems>
    <!-- Content -->
</SwipeView>
```

In this example, the `SwipeView` must be swiped for 200 device-independent units before the `SwipeItem` is fully revealed.

NOTE

Currently, the `Threshold` property is only implemented on iOS and Android.

Swipe mode

The `SwipeItems` class has a `Mode` property, which indicates the effect of a swipe interaction. This property should be set to one of the `SwipeMode` enumeration members:

- `Reveal` indicates that a swipe reveals the swipe items. This is the default value of the `SwipeItems.Mode` property.
- `Execute` indicates that a swipe executes the swipe items.

In reveal mode, the user swipes a `SwipeView` to open a menu consisting of one or more swipe items, and must explicitly tap a swipe item to execute it. After the swipe item has been executed the swipe items are closed and the `SwipeView` content is re-displayed. In execute mode, the user swipes a `SwipeView` to open a menu consisting of one or more swipe items, which are then automatically executed. Following execution, the swipe items are closed and the `SwipeView` content is re-displayed.

The following example shows a `SwipeView` configured to use execute mode:

```
<SwipeView>
    <SwipeView.LeftItems>
        <SwipeItems Mode="Execute">
            <SwipeItem Text="Delete"
                IconImageSource="delete.png"
                BackgroundColor="LightPink"
                Command="{Binding DeleteCommand}" />
        </SwipeItems>
    </SwipeView.LeftItems>
    <!-- Content -->
</SwipeView>
```

In this example, the `SwipeView` content can be swiped right to reveal the swipe item, which is executed immediately. Following execution, the `SwipeView` content is re-displayed.

Swipe behavior

The `SwipeItems` class has a `SwipeBehaviorOnInvoked` property, which indicates how a `SwipeView` behaves after a swipe item is invoked. This property should be set to one of the `SwipeBehaviorOnInvoked` enumeration members:

- `Auto` indicates that in reveal mode the `SwipeView` closes after a swipe item is invoked, and in execute mode the `SwipeView` remains open after a swipe item is invoked. This is the default value of the `SwipeItems.SwipeBehaviorOnInvoked` property.
- `Close` indicates that the `SwipeView` closes after a swipe item is invoked.
- `RemainOpen` indicates that the `SwipeView` remains open after a swipe item is invoked.

The following example shows a `SwipeView` configured to remain open after a swipe item is invoked:

```
<SwipeView>
    <SwipeView.LeftItems>
        <SwipeItems SwipeBehaviorOnInvoked="RemainOpen">
            <SwipeItem Text="Favorite"
                IconImageSource="favorite.png"
                BackgroundColor="LightGreen"
                Invoked="OnFavoriteSwipeItemInvoked" />
            <SwipeItem Text="Delete"
                IconImageSource="delete.png"
                BackgroundColor="LightPink"
                Invoked="OnDeleteSwipeItemInvoked" />
        </SwipeItems>
    </SwipeView.LeftItems>
    <!-- Content -->
</SwipeView>
```

Custom swipe items

Custom swipe items can be defined with the `SwipeItemView` type. The `SwipeItemView` class derives from the `ContentView` class, and adds the following properties:

- `Command`, of type `ICommand`, which is executed when a swipe item is tapped.
- `CommandParameter`, of type `object`, which is the parameter that's passed to the `Command`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `SwipeItemView` class also defines an `Invoked` event that's fired when the item is tapped, after the `Command` is executed.

The following example shows a `SwipeItemView` object in the `LeftItems` collection of a `SwipeView`:

```

<SwipeView>
    <SwipeView.LeftItems>
        <SwipeItems>
            <SwipeItemView Command="{Binding CheckAnswerCommand}"
                           CommandParameter="{Binding Source={x:Reference resultEntry}, Path=Text}">
                <StackLayout Margin="10"
                             WidthRequest="300">
                    <Entry x:Name="resultEntry"
                           Placeholder="Enter answer"
                           HorizontalOptions="CenterAndExpand" />
                    <Label Text="Check"
                           FontAttributes="Bold"
                           HorizontalOptions="Center" />
                </StackLayout>
            </SwipeItemView>
        </SwipeItems>
    </SwipeView.LeftItems>
    <!-- Content -->
</SwipeView>

```

In this example, the `SwipeItemView` comprises a `StackLayout` containing an `Entry` and a `Label`. After the user enters input into the `Entry`, the rest of the `SwipeViewItem` can be tapped which executes the `ICommand` defined by the `SwipeItemView.Command` property.

Open and close a SwipeView programmatically

`SwipeView` includes `Open` and `Close` methods, which programmatically open and close the swipe items, respectively. By default, these methods will animate the `SwipeView` when its opened or closed.

The `open` method requires an `OpenSwipeItem` argument, to specify the direction the `SwipeView` will be opened from. The `OpenSwipeItem` enumeration has four members:

- `LeftItems`, which indicates that the `SwipeView` will be opened from the left, to reveal the swipe items in the `LeftItems` collection.
- `TopItems`, which indicates that the `SwipeView` will be opened from the top, to reveal the swipe items in the `TopItems` collection.
- `RightItems`, which indicates that the `SwipeView` will be opened from the right, to reveal the swipe items in the `RightItems` collection.
- `BottomItems`, which indicates that the `SwipeView` will be opened from the bottom, to reveal the swipe items in the `BottomItems` collection.

In addition, the `Open` method also accepts an optional `bool` argument that defines whether the `SwipeView` will be animated when it opens.

Given a `SwipeView` named `swipeView`, the following example shows how to open a `SwipeView` to reveal the swipe items in the `LeftItems` collection:

```
swipeView.Open(OpenSwipeItem.LeftItems);
```

The `swipeView` can then be closed with the `close` method:

```
swipeView.Close();
```

NOTE

The `Close` method also accepts an optional `bool` argument that defines whether the `SwipeView` will be animated when it closes.

Disable a SwipeView

An application may enter a state where swiping an item of content is not a valid operation. In such cases, the `SwipeView` can be disabled by setting its `.IsEnabled` property to `false`. This will prevent users from being able to swipe content to reveal swipe items.

In addition, when defining the `Command` property of a `SwipeItem` or `SwipeItemView`, the `CanExecute` delegate of the `ICommand` can be specified to enable or disable the swipe item.

Related links

- [SwipeView \(sample\)](#)
- [Xamarin.Forms MenuItem](#)

Xamarin.Forms CheckBox

8/4/2022 • 3 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `CheckBox` is a type of button that can either be checked or empty. When a checkbox is checked, it's considered to be on. When a checkbox is empty, it's considered to be off.

`CheckBox` defines a `bool` property named `IsChecked`, which indicates whether the `CheckBox` is checked. This property is also backed by a `BindableProperty` object, which means that it can be styled, and be the target of data bindings.

NOTE

The `IsChecked` bindable property has a default binding mode of `BindingMode.TwoWay`.

`CheckBox` defines a `CheckedChanged` event that's fired when the `IsChecked` property changes, either through user manipulation or when an application sets the `IsChecked` property. The `CheckedChangedEventArgs` object that accompanies the `CheckedChanged` event has a single property named `Value`, of type `bool`. When the event is fired, the value of the `Value` property is set to the new value of the `IsChecked` property.

Create a CheckBox

The following example shows how to instantiate a `CheckBox` in XAML:

```
<CheckBox />
```

This XAML results in the appearance shown in the following screenshots:



iOS



Android

By default, the `CheckBox` is empty. The `CheckBox` can be checked by user manipulation, or by setting the `IsChecked` property to `true`:

```
<CheckBox IsChecked="true" />
```

This XAML results in the appearance shown in the following screenshots:



iOS



Android

Alternatively, a `CheckBox` can be created in code:

```
CheckBox checkBox = new CheckBox { IsChecked = true };
```

Respond to a CheckBox changing state

When the `IsChecked` property changes, either through user manipulation or when an application sets the `IsChecked` property, the `CheckedChanged` event fires. An event handler for this event can be registered to respond to the change:

```
<CheckBox CheckedChanged="OnCheckBoxCheckedChanged" />
```

The code-behind file contains the handler for the `CheckedChanged` event:

```
void OnCheckBoxCheckedChanged(object sender, CheckedChangedEventArgs e)
{
    // Perform required operation after examining e.Value
}
```

The `sender` argument is the `CheckBox` responsible for this event. You can use this to access the `CheckBox` object, or to distinguish between multiple `CheckBox` objects sharing the same `CheckedChanged` event handler.

Alternatively, an event handler for the `CheckedChanged` event can be registered in code:

```
CheckBox checkBox = new CheckBox { ... };
checkBox.CheckedChanged += (sender, e) =>
{
    // Perform required operation after examining e.Value
};
```

Data bind a CheckBox

The `CheckedChanged` event handler can be eliminated by using data binding and triggers to respond to a `CheckBox` being checked or empty:

```
<CheckBox x:Name="checkBox" />
<Label Text="Lorem ipsum dolor sit amet, elit rutrum, enim hendrerit augue vitae praesent sed non, lorem
aenean quis praesent pede.">
    <Label.Triggers>
        <DataTrigger TargetType="Label"
            Binding="{Binding Source={x:Reference checkBox}, Path=IsChecked}"
            Value="true">
            <Setter Property="FontAttributes"
                Value="Italic, Bold" />
            <Setter Property="FontSize"
                Value="Large" />
        </DataTrigger>
    </Label.Triggers>
</Label>
```

In this example, the `Label` uses a binding expression in a data trigger to monitor the `IsChecked` property of the `CheckBox`. When this property becomes `true`, the `FontAttributes` and `FontSize` properties of the `Label` change. When the `IsChecked` property returns to `false`, the `FontAttributes` and `FontSize` properties of the `Label` are reset to their initial state.

In the following screenshots, the iOS screenshot shows the `Label` formatting when the `CheckBox` is empty, while the Android screenshot shows the `Label` formatting when the `CheckBox` is checked:



For more information about triggers, see [Xamarin.Forms Triggers](#).

Disable a Checkbox

Sometimes an application enters a state where a `CheckBox` being checked is not a valid operation. In such cases, the `CheckBox` can be disabled by setting its `IsEnabled` property to `false`.

CheckBox appearance

In addition to the properties that `CheckBox` inherits from the `View` class, `CheckBox` also defines a `Color` property that sets its color to a `Color`:

```
<CheckBox Color="Red" />
```

The following screenshots show a series of checked `CheckBox` objects, where each object has its `Color` property set to a different `Color`:



CheckBox visual states

`CheckBox` has an `IsChecked` `VisualState` that can be used to initiate a visual change to the `CheckBox` when it becomes checked.

The following XAML example shows how to define a visual state for the `IsChecked` state:

```
<CheckBox ...>
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">
                <VisualState.Setters>
                    <Setter Property="Color"
                           Value="Red" />
                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="IsChecked">
                <VisualState.Setters>
                    <Setter Property="Color"
                           Value="Green" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</CheckBox>
```

In this example, the `IsChecked` `VisualState` specifies that when the `CheckBox` is checked, its `Color` property will be set to green. The `Normal` `VisualState` specifies that when the `CheckBox` is in a normal state, its `Color` property will be set to red. Therefore, the overall effect is that the `CheckBox` is red when it's empty, and green when it's checked.

For more information about visual states, see [Xamarin.Forms Visual State Manager](#).

Related links

- [CheckBox Demos \(sample\)](#)
- [Xamarin.Forms Triggers](#)
- [Xamarin.Forms Visual State Manager](#)

Xamarin.Forms DatePicker

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

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.
- `CharacterSpacing`, of type `double`, is the spacing between characters of the `DatePicker` text.

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 `DateSelected` 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();
    }

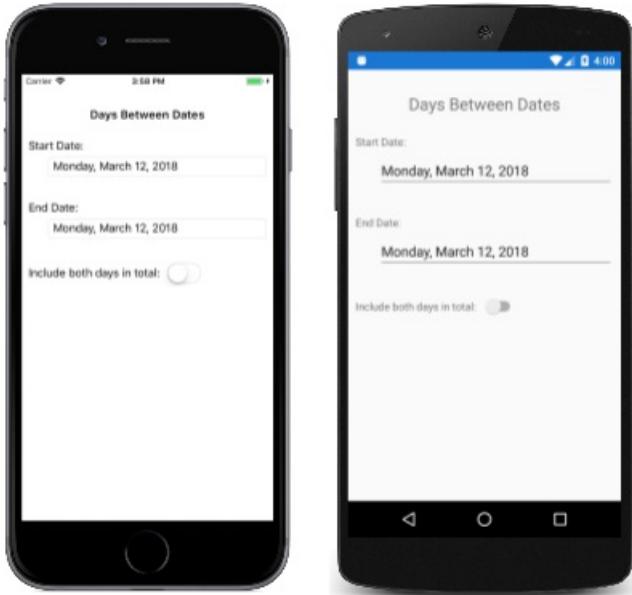
    void OnSwitchToggled(object sender, ToggledEventArgs args)
    {
        Recalculate();
    }

    void Recalculate()
    {
        TimeSpan timeSpan = endDatePicker.Date - startDatePicker.Date +
            (includeSwitch.IsToggled ? TimeSpan.FromDays(1) : TimeSpan.Zero);

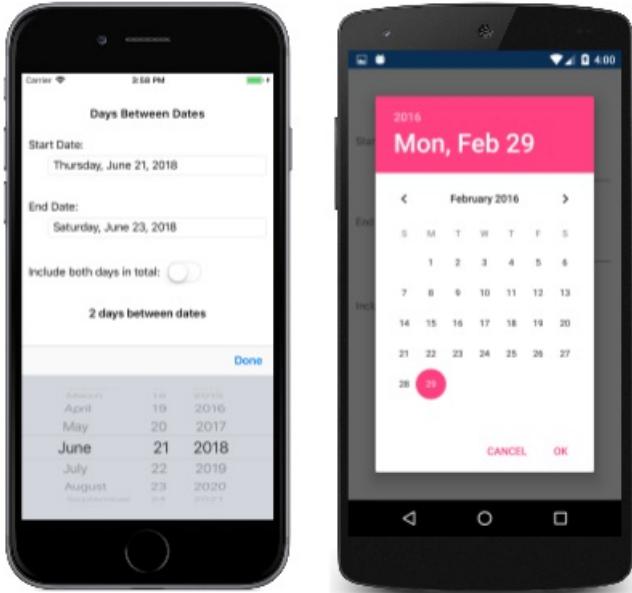
        resultLabel.Text = String.Format("{0} day{1} between dates",
                                         timeSpan.Days, timeSpan.Days == 1 ? "" : "s");
    }
}

```

When the sample is first run, both `DatePicker` views are initialized to today's date. The following screenshot shows the program running on iOS and Android:



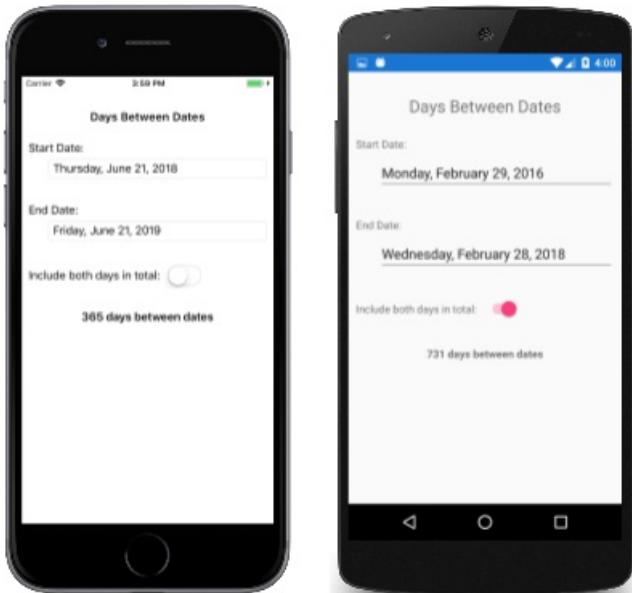
Tapping either of the `DatePicker` displays invokes the platform date picker. The platforms implement this date picker in very different ways, but each approach is familiar to users of that platform:



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](#)

Xamarin.Forms Slider

8/4/2022 • 12 minutes to read • [Edit Online](#)



[Download the sample](#)

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`.

`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.

`Slider` also defines `DragStarted` and `DragCompleted` events, that are fired at the beginning and end of the drag action. Unlike the `ValueChanged` event, the `DragStarted` and `DragCompleted` events are only fired through user manipulation of the `Slider`. When the `DragStarted` event fires, the `DragStartedCommand`, of type `ICommand`, is executed. Similarly, when the `DragCompleted` event fires, the `DragCompletedCommand`, of type `ICommand`, is executed.

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.
- `ThumbImageSource` is the image to use for the thumb, of type `ImageSource`.

NOTE

The `ThumbColor` and `ThumbImageSource` properties are mutually exclusive. If both properties are set, the `ThumbImageSource` 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
        };

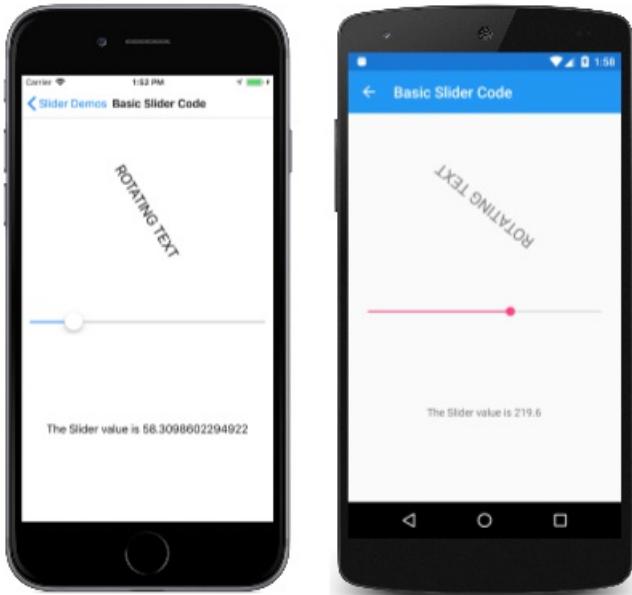
        Slider slider = new Slider
        {
            Maximum = 360
        };
        slider.ValueChanged += (sender, args) =>
        {
            rotationLabel.Rotation = slider.Value;
            displayLabel.Text = String.Format("The Slider value is {0}", args.NewValue);
        };

        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 and Android 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:

```

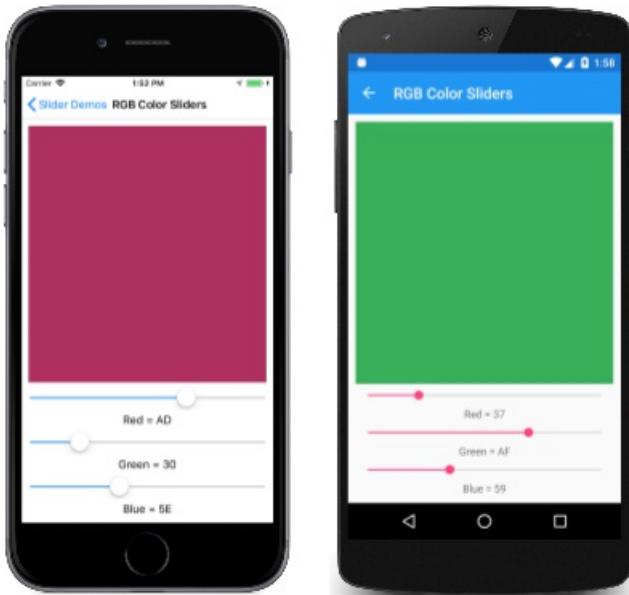
public partial class RgbColorSlidersPage : ContentPage
{
    public RgbColorSlidersPage()
    {
        InitializeComponent();
    }

    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>

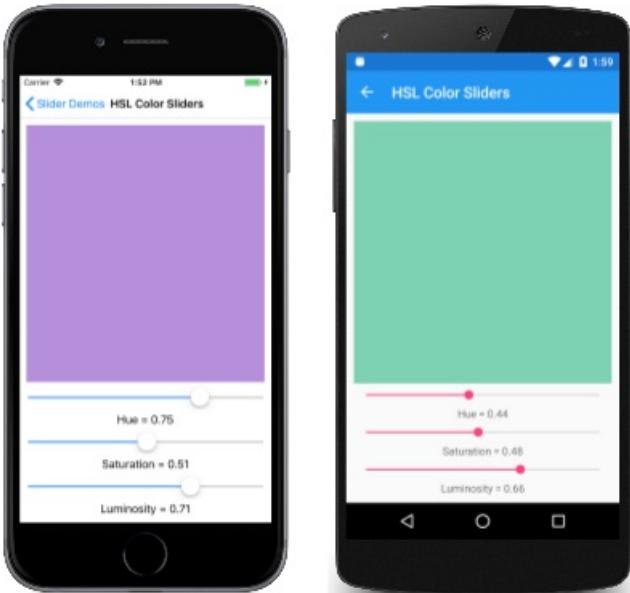
    <StackLayout Margin="10">
        <BoxView Color="{Binding Color}" VerticalOptions="FillAndExpand" />

        <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

8/4/2022 • 7 minutes to read • [Edit Online](#)



[Download the sample](#)

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
        };

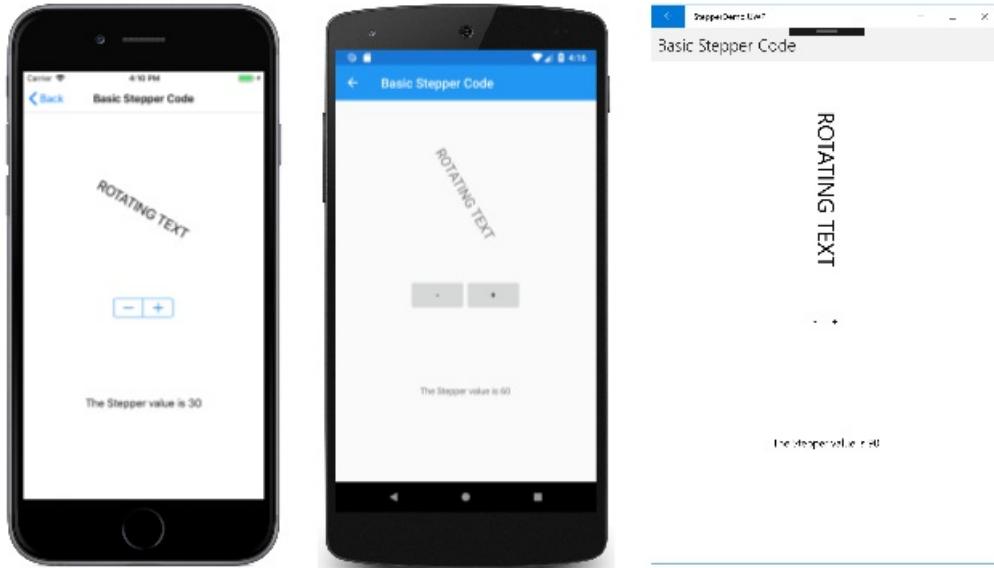
        Stepper stepper = new Stepper
        {
            Maximum = 360,
            Increment = 30,
            HorizontalOptions = LayoutOptions.Center
        };
        stepper.ValueChanged += (sender, e) =>
        {
            rotationLabel.Rotation = stepper.Value;
            displayLabel.Text = string.Format("The 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` `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](#)

Xamarin.Forms Switch

8/4/2022 • 4 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `Switch` control is a horizontal toggle button that can be manipulated by the user to toggle between on and off states, which are represented by a `boolean` value. The `Switch` class inherits from `View`.

The following screenshots show a `Switch` control in its **on** and **off** toggle states on iOS and Android:



iOS



Android



The `Switch` control defines the following properties:

- `IsToggled` is a `boolean` value that indicates whether the `Switch` is **on**.
- `OnColor` is a `Color` that affects how the `Switch` is rendered in the toggled, or **on**, state.
- `ThumbColor` is the `Color` of the switch thumb.

These properties are backed by a `BindableProperty` object, which means the `Switch` can be styled and be the target of data bindings.

The `Switch` control defines a `Toggled` event that is fired when the `IsToggled` property changes, either through user manipulation or when an application sets the `IsToggled` property. The `ToggledEventArgs` object that accompanies the `Toggled` event has a single property named `Value`, of type `bool`. When the event is fired, the value of the `Value` property reflects the new value of the `IsToggled` property.

Create a Switch

A `Switch` can be instantiated in XAML. Its `IsToggled` property can be set to toggle the `Switch`. By default, the `IsToggled` property is `false`. The following example shows how to instantiate a `Switch` in XAML with the optional `IsToggled` property set:

```
<Switch IsToggled="true"/>
```

A `Switch` can also be created in code:

```
Switch switchControl = new Switch { IsToggled = true };
```

Switch appearance

In addition to the properties that `Switch` inherits from the `View` class, `Switch` also defines `OnColor` and `ThumbColor` properties. The `OnColor` property can be set to define the `Switch` color when it is toggled to its **on** state, and the `ThumbColor` property can be set to define the `Color` of the switch thumb. The following example shows how to instantiate a `Switch` in XAML with these properties set:

```
<Switch OnColor="Orange"  
       ThumbColor="Green" />
```

The properties can also be set when creating a `Switch` in code:

```
Switch switch = new Switch { OnColor = Color.Orange, ThumbColor = Color.Green };
```

The following screenshot shows the `Switch` in its **on** and **off** toggle states, with the `OnColor` and `ThumbColor` properties set:



iOS



Android

Respond to a Switch state change

When the `IsToggled` property changes, either through user manipulation or when an application sets the `IsToggled` property, the `Toggled` event fires. An event handler for this event can be registered to respond to the change:

```
<Switch Toggled="OnToggled" />
```

The code-behind file contains the handler for the `Toggled` event:

```
void OnToggled(object sender, ToggledEventArgs e)  
{  
    // Perform an action after examining e.Value  
}
```

The `sender` argument in the event handler is the `Switch` responsible for firing this event. You can use the `sender` property to access the `Switch` object, or to distinguish between multiple `Switch` objects sharing the same `Toggled` event handler.

The `Toggled` event handler can also be assigned in code:

```
Switch switchControl = new Switch {...};  
switchControl.Toggled += (sender, e) =>  
{  
    // Perform an action after examining e.Value  
}
```

Data bind a Switch

The `Toggled` event handler can be eliminated by using data binding and triggers to respond to a `Switch` changing toggle states.

```

<Switch x:Name="styleSwitch" />
<Label Text="Lorem ipsum dolor sit amet, elit rutrum, enim hendrerit augue vitae praesent sed non, lorem
aenean quis praesent pede.">
    <Label.Triggers>
        <DataTrigger TargetType="Label"
            Binding="{Binding Source={x:Reference styleSwitch}, Path=IsToggled}"
            Value="true">
            <Setter Property="FontAttributes"
                Value="Italic, Bold" />
            <Setter Property="FontSize"
                Value="Large" />
        </DataTrigger>
    </Label.Triggers>
</Label>

```

In this example, the `Label` uses a binding expression in a `DataTrigger` to monitor the `IsToggled` property of the `Switch` named `styleSwitch`. When this property becomes `true`, the `FontAttributes` and `FontSize` properties of the `Label` are changed. When the `IsToggled` property returns to `false`, the `FontAttributes` and `FontSize` properties of the `Label` are reset to their initial state.

For information about triggers, see [Xamarin.Forms Triggers](#).

Switch visual states

`Switch` has `On` and `Off` visual states that can be used to initiate a visual change when the `IsToggled` property changes.

The following XAML example shows how to define visual states for the `On` and `Off` states:

```

<Switch IsToggled="True">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="On">
                <VisualState.Setters>
                    <Setter Property="ThumbColor"
                        Value="MediumSpringGreen" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="Off">
                <VisualState.Setters>
                    <Setter Property="ThumbColor"
                        Value="Red" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Switch>

```

In this example, the `On` `VisualState` specifies that when the `IsToggled` property is `true`, the `ThumbColor` property will be set to medium spring green. The `Off` `visualState` specifies that when the `IsToggled` property is `false`, the `ThumbColor` property will be set to red. Therefore, the overall effect is that when the `Switch` is in an off position its thumb is red, and its thumb is medium spring green when the `Switch` is in an on position:



On



Off

For more information about visual states, see [Xamarin.Forms Visual State Manager](#).

Disable a Switch

An application may enter a state where the `Switch` being toggled is not a valid operation. In such cases, the `Switch` can be disabled by setting its `.IsEnabled` property to `false`. This will prevent users from being able to manipulate the `Switch`.

Related links

- [Switch Demos](#)
- [Xamarin.Forms Triggers](#)
- [Xamarin.Forms Visual State Manager](#)

Xamarin.Forms TimePicker

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

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.
- `CharacterSpacing`, of type `double`, is the spacing between characters of the `TimePicker` text.

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();
        }
    }

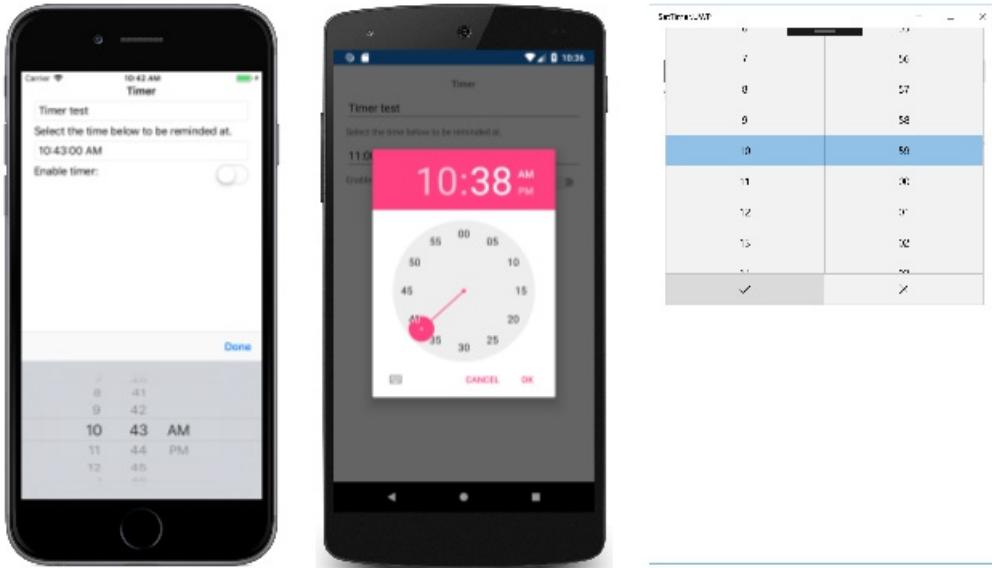
    void OnSwitchToggled(object sender, ToggledEventArgs args)
    {
        SetTriggerTime();
    }

    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.

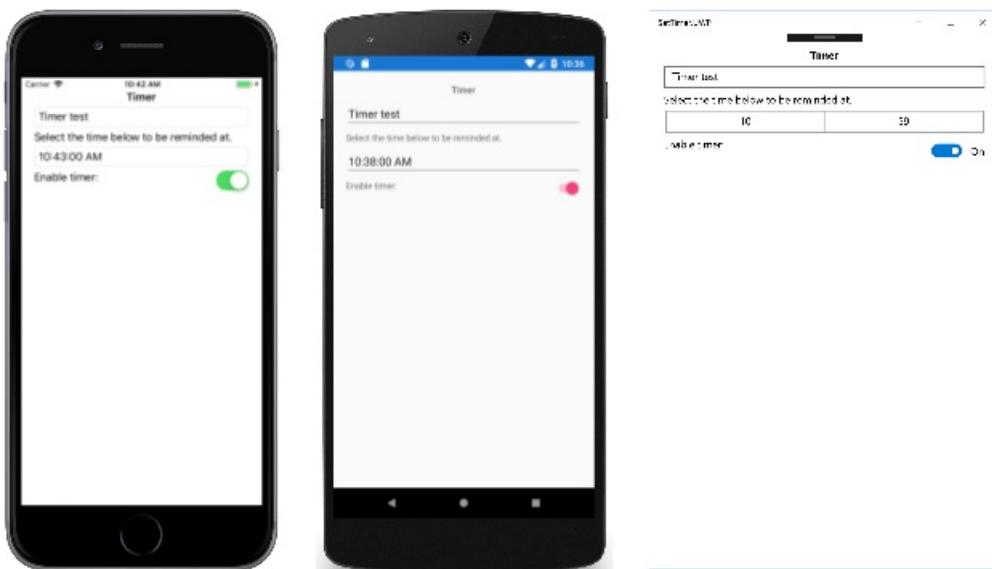
When the sample is first run, the `TimePicker` view is initialized to 11am. Tapping the `TimePicker` invokes the platform time picker. The platforms implement the time picker in very different ways, but each approach is familiar to users of that platform:



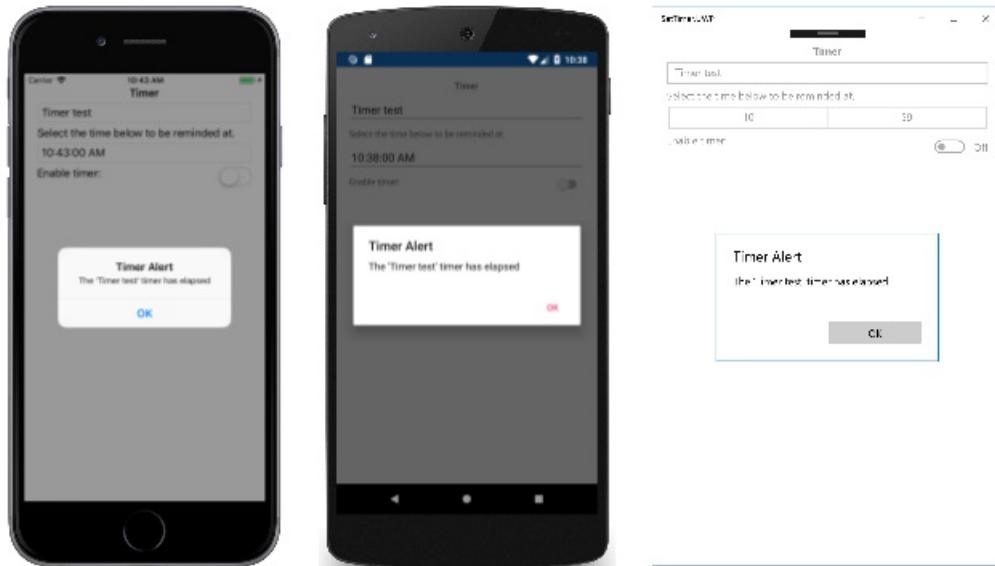
TIP

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](#)

Xamarin.Forms Editor

8/4/2022 • 8 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Editor` control is used to accept multi-line input.

Set and read 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 x:Name="editor" Text="I am an Editor" />
```

In C#:

```
var editor = new Editor { Text = "I am an Editor" };
```

To read text, access the `Text` property in C#:

```
var text = editor.Text;
```

Set 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 };
```

Prevent text entry

Users can be prevented from modifying the text in an `Editor` by setting the `IsReadOnly` property, which has a default value of `false`, to `true`:

```
<Editor Text="This is a read-only Editor"  
       IsReadOnly="true" />
```

```
var editor = new Editor { Text = "This is a read-only Editor", IsReadOnly = true }));
```

NOTE

The `IsReadOnly` property does not alter the visual appearance of an `Editor`, unlike the `.IsEnabled` property that also changes the visual appearance of the `Editor` to gray.

Transform text

An `Editor` can transform the casing of its text, stored in the `Text` property, by setting the `TextTransform` property to a value of the `TextTransform` enumeration. This enumeration has four values:

- `None` indicates that the text won't be transformed.
- `Default` indicates that the default behavior for the platform will be used. This is the default value of the `TextTransform` property.
- `Lowercase` indicates that the text will be transformed to lowercase.
- `Uppercase` indicates that the text will be transformed to uppercase.

The following example shows transforming text to uppercase:

```
<Editor Text="This text will be displayed in uppercase."
        TextTransform="Uppercase" />
```

The equivalent C# code is:

```
Editor editor = new Editor
{
    Text = "This text will be displayed in uppercase.",
    TextTransform = TextTransform.Uppercase
};
```

Limit 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.

Character spacing

Character spacing can be applied to an `Editor` by setting the `Editor.CharacterSpacing` property to a `double` value:

```
<Editor ...
        CharacterSpacing="10" />
```

The equivalent C# code is:

```
Editor editor = new Editor { CharacterSpacing = 10 };
```

The result is that characters in the text displayed by the `Editor` are spaced `CharacterSpacing` device-independent units apart.

NOTE

The `CharacterSpacing` property value is applied to the text displayed by the `Text` and `Placeholder` properties.

Auto-size 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 `EditorAutoSizeOption` 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.

Customize 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);
```

Enable and disable 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.

Enable and disable 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`:

```
<Editor ... IsTextPredictionEnabled="false" />
```

```
var editor = new Editor { ... 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.

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#:

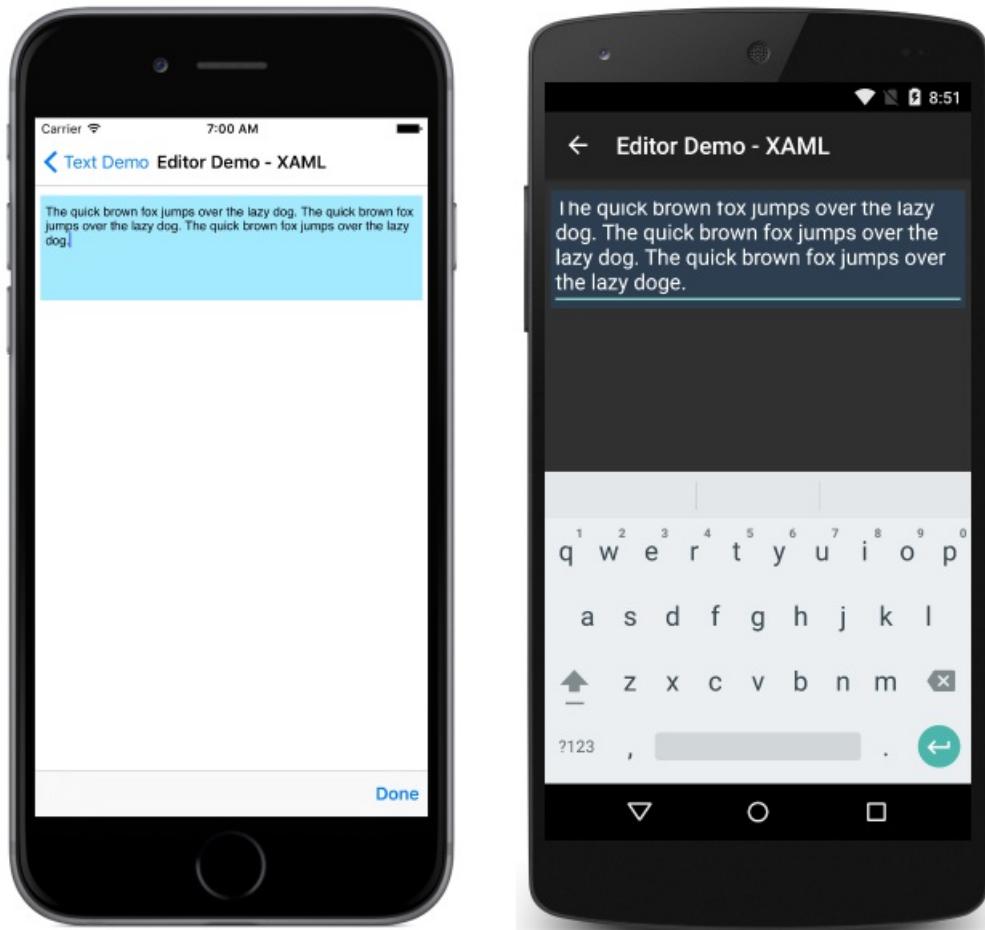
```
public partial class EditorPage : ContentPage
{
    public EditorPage ()
    {
        InitializeComponent ();
        var layout = new StackLayout { Padding = new Thickness(5,10) };
        this.Content = layout;
        //dark blue on UWP & Android, light blue on iOS
        var editor = new Editor { 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.

Events and 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.

NOTE

The `VisualElement` class, from which `Entry` inherits, also has `Focused` and `Unfocused` events.

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 (or by pressing the Tab key on UWP). 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](#)

Xamarin.Forms Entry

8/4/2022 • 10 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

Set and read 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 x:Name="entry" Text="I am an Entry" />
```

In C#:

```
var entry = new Entry { Text = "I am an Entry" };
```

To read text, access the `Text` property in C#:

```
var text = entry.Text;
```

Set 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 };
```

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.

Prevent text entry

Users can be prevented from modifying the text in an `Entry` by setting the `IsReadOnly` property, which has a default value of `false`, to `true`:

```
<Entry Text="This is a read-only Entry"  
      IsReadOnly="true" />
```

```
var entry = new Entry { Text = "This is a read-only Entry", IsReadOnly = true );
```

NOTE

The `IsReadonly` property does not alter the visual appearance of an `Entry`, unlike the `.IsEnabled` property that also changes the visual appearance of the `Entry` to gray.

Transform text

An `Entry` can transform the casing of its text, stored in the `Text` property, by setting the `TextTransform` property to a value of the `TextTransform` enumeration. This enumeration has four values:

- `None` indicates that the text won't be transformed.
- `Default` indicates that the default behavior for the platform will be used. This is the default value of the `TextTransform` property.
- `Lowercase` indicates that the text will be transformed to lowercase.
- `Uppercase` indicates that the text will be transformed to uppercase.

The following example shows transforming text to uppercase:

```
<Entry Text="This text will be displayed in uppercase."  
      TextTransform="Uppercase" />
```

The equivalent C# code is:

```
Entry entry = new Entry  
{  
    Text = "This text will be displayed in uppercase.",  
    TextTransform = TextTransform.Uppercase  
};
```

Limit 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.

Character spacing

Character spacing can be applied to an `Entry` by setting the `Entry.CharacterSpacing` property to a `double` value:

```
<Entry ...  
    CharacterSpacing="10" />
```

The equivalent C# code is:

```
Entry entry = new Entry { CharacterSpacing = 10 };
```

The result is that characters in the text displayed by the `Entry` are spaced `CharacterSpacing` device-independent units apart.

NOTE

The `CharacterSpacing` property value is applied to the text displayed by the `Text` and `Placeholder` properties.

Password fields

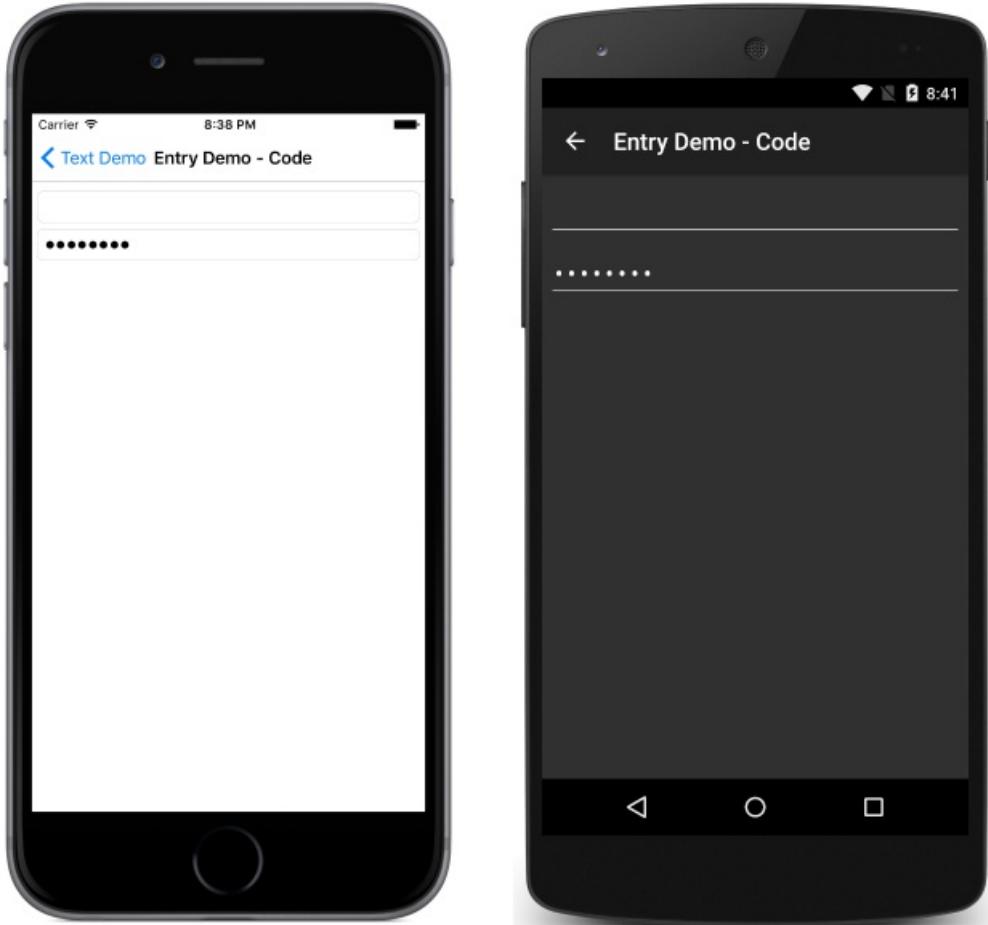
`Entry` provides the `IsPassword` property. When `IsPassword` is `true`, the contents of the field will be presented as black circles:

In XAML:

```
<Entry IsPassword="true" />
```

In C#:

```
var MyEntry = new Entry { IsPassword = true };
```



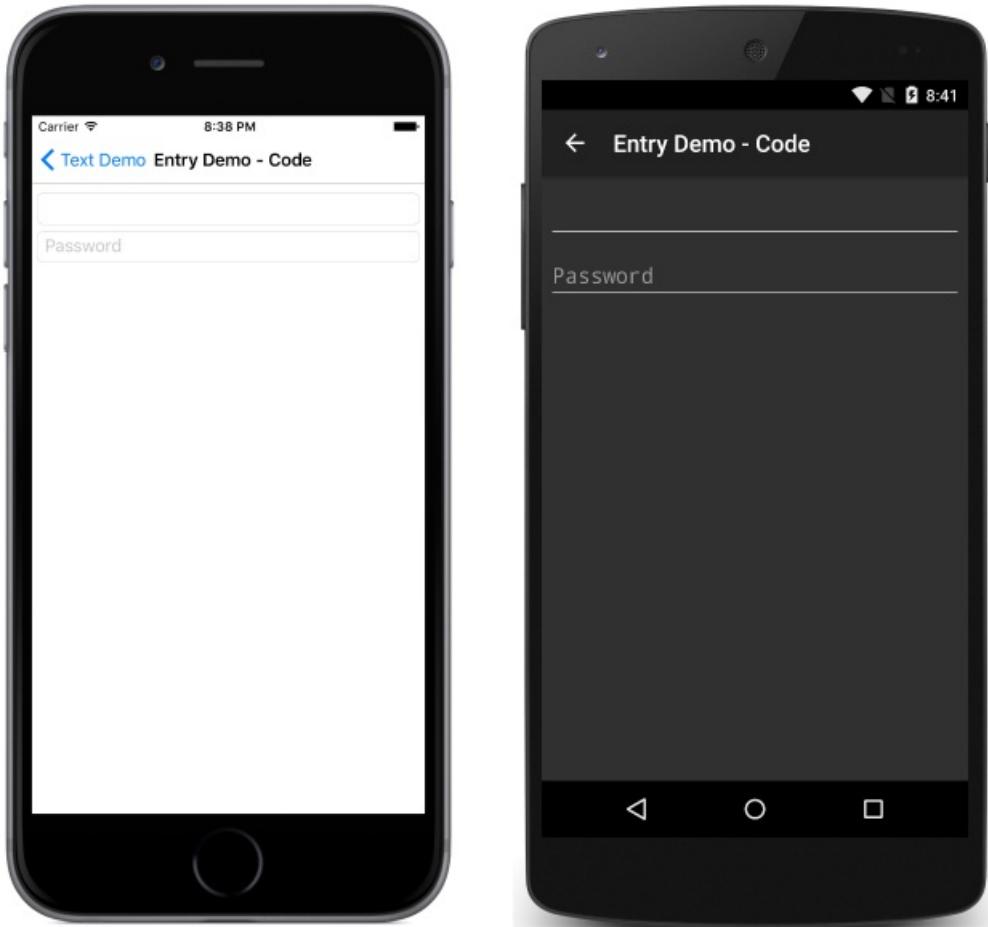
Placeholders may be used with instances of `Entry` that are configured as password fields:

In XAML:

```
<Entry IsPassword="true" Placeholder="Password" />
```

In C#:

```
var MyEntry = new Entry { IsPassword = true, Placeholder = "Password" };
```



Set 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.

Display a clear button

The `clearButtonVisibility` property can be used to control whether an `Entry` displays a clear button, which

enables the user to clear the text. This property should be set to a `ClearButtonVisibility` enumeration member:

- `Never` indicates that a clear button will never be displayed. This is the default value for the `Entry.ClearButtonVisibility` property.
- `WhileEditing` indicates that a clear button will be displayed in the `Entry`, while it has focus and text.

The following example shows setting the property in XAML:

```
<Entry Text="Xamarin.Forms"  
      ClearButtonVisibility="WhileEditing" />
```

The equivalent C# code is:

```
var entry = new Entry { Text = "Xamarin.Forms", ClearButtonVisibility = ClearButtonVisibility.WhileEditing };
```

The following screenshots show an `Entry` with the clear button enabled:



Customize 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);
```

Customize 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](#).

Enable and disable 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.

Enable and disable 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.

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.

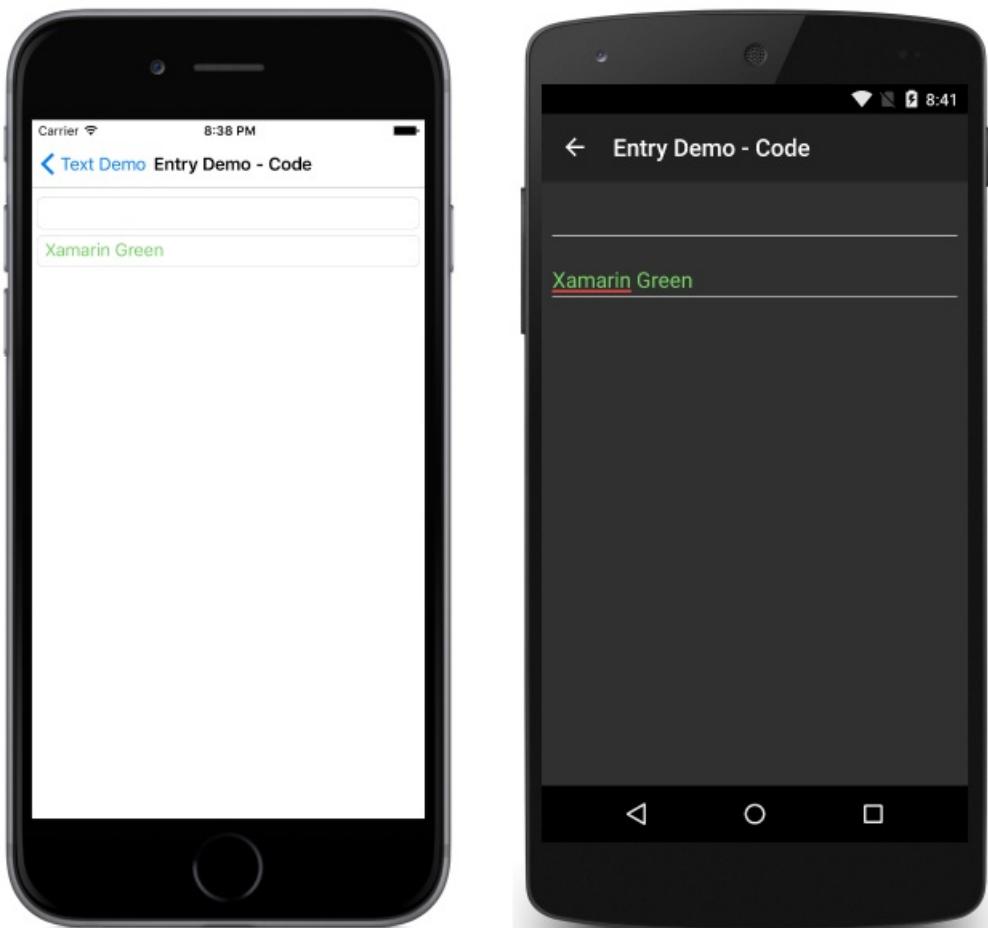
Use the following code to set the text color of an entry:

In XAML:

```
<Entry TextColor="Green" />
```

In C#:

```
var entry = new Entry();
entry.TextColor = Color.Green;
```



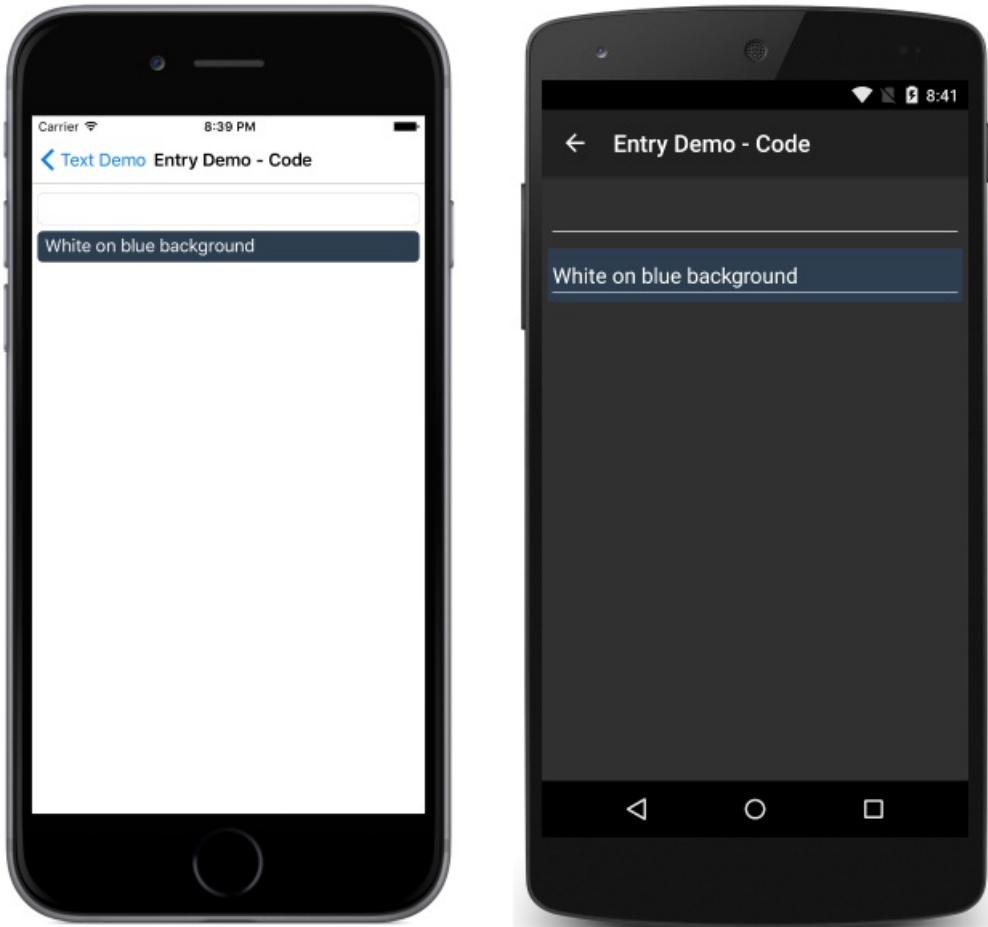
Note that the placeholder is not affected by the specified `TextColor`.

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.

NOTE

The `VisualElement` class, from which `Entry` inherits, also has `Focused` and `Unfocused` events.

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 (or by pressing the Tab key on UWP). 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 ActivityIndicator

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `ActivityIndicator` control displays an animation to show that the application is engaged in a lengthy activity. Unlike the `ProgressBar`, the `ActivityIndicator` gives no indication of progress. The `ActivityIndicator` inherits from `View`.

The following screenshots show an `ActivityIndicator` control on iOS and Android:



iOS



Android

The `ActivityIndicator` control defines the following properties:

- `Color` is a `Color` value that defines the display color of the `ActivityIndicator`.
- `IsRunning` is a `bool` value that indicates whether the `ActivityIndicator` should be visible and animating, or hidden. When the value is `false` the `ActivityIndicator` isn't visible.

These properties are backed by `BindableProperty` objects, which means that the `ActivityIndicator` can be styled and be the target of data bindings.

Create an ActivityIndicator

The `ActivityIndicator` class can be instantiated in XAML. Its `IsRunning` property determines if the control is visible and animating. The `IsRunning` property defaults to `false`. The following example shows how to instantiate an `ActivityIndicator` in XAML with the optional `IsRunning` property set:

```
<ActivityIndicator IsRunning="true" />
```

An `ActivityIndicator` can also be created in code:

```
ActivityIndicator activityIndicator = new ActivityIndicator { IsRunning = true };
```

ActivityIndicator appearance properties

The `color` property defines the `ActivityIndicator` color. The following example shows how to instantiate an `ActivityIndicator` in XAML with the `Color` property set:

```
<ActivityIndicator Color="Orange" />
```

The `color` property can also be set when creating an `ActivityIndicator` in code:

```
ActivityIndicator activityIndicator = new ActivityIndicator { Color = Color.Orange };
```

The following screenshots show the `ActivityIndicator` with the `color` property set to `Color.Orange` on iOS and Android:



iOS



Android

Related links

- [ActivityIndicator Demos](#)
- [ProgressBar](#)

Xamarin.Forms ProgressBar

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `ProgressBar` control visually represents progress as a horizontal bar that is filled to a percentage represented by a `float` value. The `ProgressBar` class inherits from `View`.

The following screenshots show a `ProgressBar` on iOS and Android:

iOS



Android



The `ProgressBar` control defines two properties:

- `Progress` is a `float` value that represents the current progress as a value from 0 to 1. `Progress` values less than 0 will be clamped to 0, values greater than 1 will be clamped to 1.
- `ProgressColor` is a `Color` that affects the interior bar color representing the current progress.

These properties are backed by `BindableProperty` objects, which means that the `ProgressBar` can be styled and be the target of data bindings.

The `ProgressBar` control also defines a `ProgressTo` method that animates the bar from its current value to a specified value. For more information, see [Animate a ProgressBar](#).

NOTE

The `ProgressBar` does not accept user manipulation so it is skipped when using the Tab key to select controls.

Create a ProgressBar

A `ProgressBar` can be instantiated in XAML. Its `Progress` property determines the fill percentage of the inner, colored bar. The default `Progress` property value is 0. The following example shows how to instantiate a `ProgressBar` in XAML with the optional `Progress` property set:

```
<ProgressBar Progress="0.5" />
```

A `ProgressBar` can also be created in code:

```
ProgressBar progressBar = new ProgressBar { Progress = 0.5f };
```

WARNING

Do not use unconstrained horizontal layout options such as `Center`, `Start`, or `End` with `ProgressBar`. On UWP, the `ProgressBar` collapses to a bar of zero width. Keep the default `HorizontalOptions` value of `Fill` and don't use a width of `Auto` when putting a `ProgressBar` in a `Grid` layout.

ProgressBar appearance properties

The `ProgressColor` property defines the inner bar color when the `Progress` property is greater than zero. The following example shows how to instantiate a `ProgressBar` in XAML with the `ProgressColor` property set:

```
<ProgressBar ProgressColor="Orange" />
```

The `ProgressColor` property can also be set when creating a `ProgressBar` in code:

```
ProgressBar progressBar = new ProgressBar { ProgressColor = Color.Orange };
```

The following screenshots show the `ProgressBar` with the `ProgressColor` property set to `Color.Orange` on iOS and Android:

iOS



Android



Animate a ProgressBar

The `ProgressTo` method animates the `ProgressBar` from its current `Progress` value to a provided value over time. The method accepts a `float` progress value, a `uint` duration in milliseconds, an `Easing` enum value and returns a `Task<bool>`. The following code demonstrates how to animate a `ProgressBar`:

```
// animate to 75% progress over 500 milliseconds with linear easing
await progressBar.ProgressTo(0.75, 500, Easing.Linear);
```

For more information about the `Easing` enumeration, see [Easing functions in Xamarin.Forms](#).

Related links

- [ProgressBar Demos](#)

Xamarin.Forms CarouselView

8/4/2022 • 2 minutes to read • [Edit Online](#)

Introduction

The `CarouselView` is a view for presenting data in a scrollable layout, where users can swipe to move through a collection of items.

Data

A `CarouselView` is populated with data by setting its `ItemsSource` property to any collection that implements `IEnumerable`. The appearance of each item can be defined by setting the `ItemTemplate` property to a `DataTemplate`.

Layout

By default, a `CarouselView` will display its items in a horizontal list. However, it also has access to the same layouts as `CollectionView`, including a vertical orientation.

Interaction

The currently displayed item in a `CarouselView` can be accessed through the `CurrentItem` and `Position` properties.

Empty views

In `CarouselView`, an empty view can be specified that provides feedback to the user when no data is available for display. The empty view can be a string, a view, or multiple views.

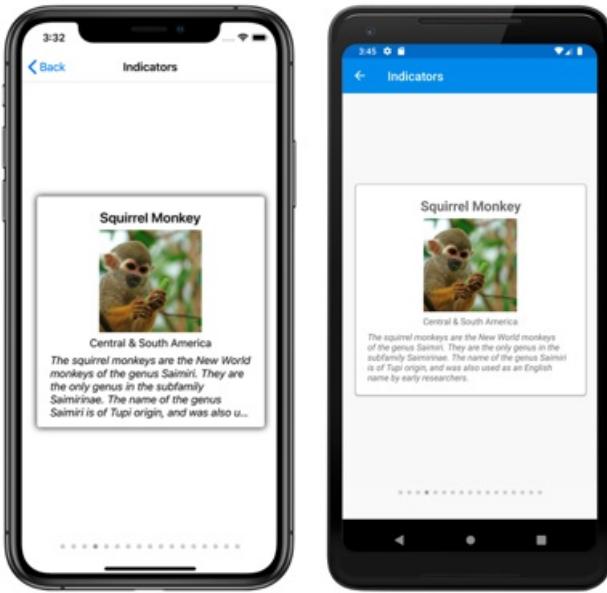
Scrolling

When a user swipes to initiate a scroll, the end position of the scroll can be controlled so that items are fully displayed. In addition, `CarouselView` defines two `ScrollTo` methods, that programmatically scroll items into view. One of the overloads scrolls the item at the specified index into view, while the other scrolls the specified item into view.

Xamarin.Forms CarouselView Introduction

8/4/2022 • 2 minutes to read • [Edit Online](#)

`CarouselView` is a view for presenting data in a scrollable layout, where users can swipe to move through a collection of items. By default, `CarouselView` will display its items in a horizontal orientation. A single item will be displayed on screen, with swipe gestures resulting in forwards and backwards navigation through the collection of items. In addition, indicators can be displayed that represent each item in the `CarouselView`:



By default, `CarouselView` provides looped access to its collection of items. Therefore, swiping backwards from the first item in the collection will display the last item in the collection. Similarly, swiping forwards from the last item in the collection will return to the first item in the collection.

`CarouselView` shares much of its implementation with `CollectionView`. However, the two controls have different use cases. `CollectionView` is typically used to present lists of data of any length, whereas `CarouselView` is typically used to highlight information in a list of limited length.

Xamarin.Forms CarouselView Data

8/4/2022 • 9 minutes to read • [Edit Online](#)



[Download the sample](#)

`CarouselView` includes the following properties that define the data to be displayed, and its appearance:

- `ItemsSource`, of type `IEnumerable`, specifies the collection of items to be displayed, and has a default value of `null`.
- `ItemTemplate`, of type `DataTemplate`, specifies the template to apply to each item in the collection of items to be displayed.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

NOTE

`CarouselView` defines a `ItemsUpdatingScrollMode` property that represents the scrolling behavior of the `CarouselView` when new items are added to it. For more information about this property, see [Control scroll position when new items are added](#).

`CarouselView` supports incremental data virtualization as the user scrolls. For more information, see [Load data incrementally](#).

Populate a CarouselView with data

A `CarouselView` is populated with data by setting its `ItemsSource` property to any collection that implements `IEnumerable`. By default, `CarouselView` displays items horizontally.

IMPORTANT

If the `CarouselView` is required to refresh as items are added, removed, or changed in the underlying collection, the underlying collection should be an `IEnumerable` collection that sends property change notifications, such as `ObservableCollection`.

`CarouselView` can be populated with data by using data binding to bind its `ItemsSource` property to an `IEnumerable` collection. In XAML, this is achieved with the `Binding` markup extension:

```
<CarouselView ItemsSource="{Binding Monkeys}" />
```

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView();
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
```

In this example, the `ItemsSource` property data binds to the `Monkeys` property of the connected.viewmodel.

NOTE

Compiled bindings can be enabled to improve data binding performance in Xamarin.Forms applications. For more information, see [Compiled Bindings](#).

For information on how to change the `CarouselView` orientation, see [Xamarin.Forms CarouselView Layout](#). For information on how to define the appearance of each item in the `CarouselView`, see [Define item appearance](#). For more information about data binding, see [Xamarin.Forms Data Binding](#).

Define item appearance

The appearance of each item in the `CarouselView` can be defined by setting the `CarouselView.ItemTemplate` property to a `DataTemplate`:

```
<CarouselView ItemsSource="{Binding Monkeys}">
    <CarouselView.ItemTemplate>
        <DataTemplate>
            <StackLayout>
                <Frame HasShadow="True"
                    BorderColor="DarkGray"
                    CornerRadius="5"
                    Margin="20"
                    HeightRequest="300"
                    HorizontalOptions="Center"
                    VerticalOptions="CenterAndExpand">
                    <StackLayout>
                        <Label Text="{Binding Name}"
                            FontAttributes="Bold"
                            FontSize="Large"
                            HorizontalOptions="Center"
                            VerticalOptions="Center" />
                        <Image Source="{Binding ImageUrl}"
                            Aspect="AspectFill"
                            HeightRequest="150"
                            WidthRequest="150"
                            HorizontalOptions="Center" />
                        <Label Text="{Binding Location}"
                            HorizontalOptions="Center" />
                        <Label Text="{Binding Details}"
                            FontAttributes="Italic"
                            HorizontalOptions="Center"
                            MaxLines="5"
                            LineBreakMode="TailTruncation" />
                    
```

The equivalent C# code is:

```

CarouselView carouselView = new CarouselView();
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");

carouselView.ItemTemplate = new DataTemplate(() =>
{
    Label nameLabel = new Label { ... };
    nameLabel.SetBinding(Label.TextProperty, "Name");

    Image image = new Image { ... };
    image.SetBinding(Image.SourceProperty, "ImageUrl");

    Label locationLabel = new Label { ... };
    locationLabel.SetBinding(Label.TextProperty, "Location");

    Label detailsLabel = new Label { ... };
    detailsLabel.SetBinding(Label.TextProperty, "Details");

    StackLayout stackLayout = new StackLayout
    {
        Children = { nameLabel, image, locationLabel, detailsLabel }
    };

    Frame frame = new Frame { ... };
    StackLayout rootStackLayout = new StackLayout
    {
        Children = { frame }
    };

    return rootStackLayout;
});

```

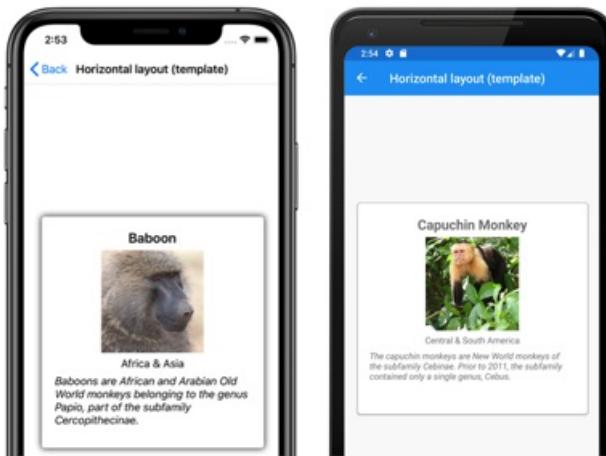
The elements specified in the `DataTemplate` define the appearance of each item in the `CarouselView`. In the example, layout within the `DataTemplate` is managed by a `StackLayout`, and the data is displayed with an `Image` object, and three `Label` objects, that all bind to properties of the `Monkey` class:

```

public class Monkey
{
    public string Name { get; set; }
    public string Location { get; set; }
    public string Details { get; set; }
    public string ImageUrl { get; set; }
}

```

The following screenshots show the result of templating each item:



For more information about data templates, see [Xamarin.Forms Data Templates](#).

Choose item appearance at runtime

The appearance of each item in the `CarouselView` can be chosen at runtime, based on the item value, by setting the `CarouselView.ItemTemplate` property to a `DataTemplateSelector` object:

```
<ContentPage ...>
    xmlns:controls="clr-namespace:CarouselViewDemos.Controls"
    x:Class="CarouselViewDemos.Views.HorizontalLayoutDataTemplateSelectorPage">
<ContentPage.Resources>
    <DataTemplate x:Key="AmericanMonkeyTemplate">
        ...
    </DataTemplate>

    <DataTemplate x:Key="OtherMonkeyTemplate">
        ...
    </DataTemplate>

    <controls:MonkeyDataTemplateSelector x:Key="MonkeySelector"
        AmericanMonkey="{StaticResource AmericanMonkeyTemplate}"
        OtherMonkey="{StaticResource OtherMonkeyTemplate}" />
</ContentPage.Resources>

<CarouselView ItemsSource="{Binding Monkeys}"
    ItemTemplate="{StaticResource MonkeySelector}" />
</ContentPage>
```

The equivalent C# code is:

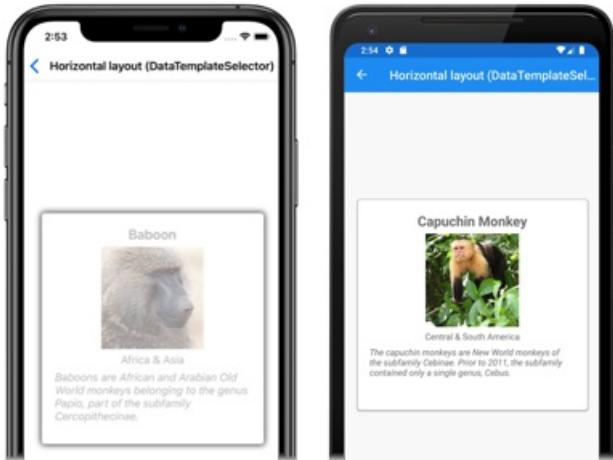
```
CarouselView carouselView = new CarouselView
{
    ItemTemplate = new MonkeyDataTemplateSelector { ... }
};
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
```

The `ItemTemplate` property is set to a `MonkeyDataTemplateSelector` object. The following example shows the `MonkeyDataTemplateSelector` class:

```
public class MonkeyDataTemplateSelector : DataTemplateSelector
{
    public DataTemplate AmericanMonkey { get; set; }
    public DataTemplate OtherMonkey { get; set; }

    protected override DataTemplate OnSelectTemplate(object item, BindableObject container)
    {
        return ((Monkey)item).Location.Contains("America") ? AmericanMonkey : OtherMonkey;
    }
}
```

The `MonkeyDataTemplateSelector` class defines `AmericanMonkey` and `OtherMonkey` `DataTemplate` properties that are set to different data templates. The `OnSelectTemplate` override returns the `AmericanMonkey` template when the monkey name contains "America". When the monkey name doesn't contain "America", the `OnSelectTemplate` override returns the `OtherMonkey` template, which displays its data grayed out:



For more information about data template selectors, see [Create a Xamarin.Forms DataTemplateSelector](#).

IMPORTANT

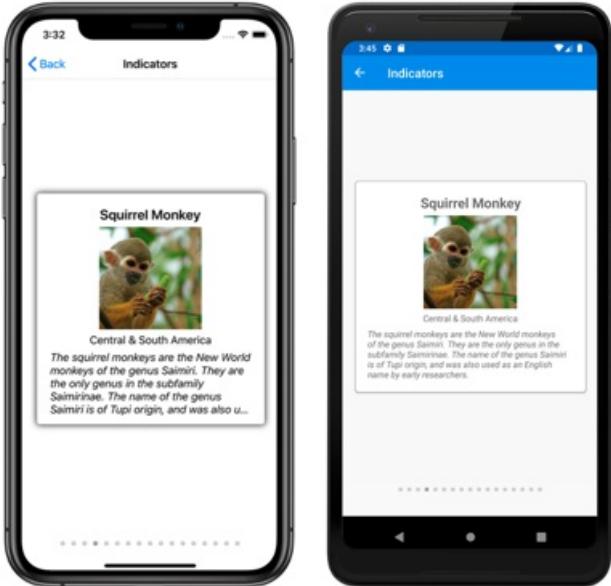
When using `CarouselView`, never set the root element of your `DataTemplate` objects to a `ViewCell`. This will result in an exception being thrown because `CarouselView` has no concept of cells.

Display indicators

Indicators, that represent the number of items and current position in a `CarouselView`, can be displayed next to the `CarouselView`. This can be accomplished with the `IndicatorView` control:

```
<StackLayout>
    <CarouselView ItemsSource="{Binding Monkeys}"
                  IndicatorView="indicatorView">
        <CarouselView.ItemTemplate>
            <!-- DataTemplate that defines item appearance -->
        </CarouselView.ItemTemplate>
    </CarouselView>
    <IndicatorView x:Name="indicatorView"
                  IndicatorColor="LightGray"
                  SelectedIndicatorColor="DarkGray"
                  HorizontalOptions="Center" />
</StackLayout>
```

In this example, the `IndicatorView` is rendered beneath the `CarouselView`, with an indicator for each item in the `CarouselView`. The `IndicatorView` is populated with data by setting the `CarouselView.IndicatorView` property to the `IndicatorView` object. Each indicator is a light gray circle, while the indicator that represents the current item in the `CarouselView` is dark gray:



IMPORTANT

Setting the `CarouselView.IndicatorView` property results in the `IndicatorView.Position` property binding to the `CarouselView.Position` property, and the `IndicatorView.ItemsSource` property binding to the `CarouselView.ItemsSource` property.

For more information about indicators, see [Xamarin.Forms IndicatorView](#).

Context menus

`CarouselView` supports context menus for items of data through the `SwipeView`, which reveals the context menu with a swipe gesture. The `SwipeView` is a container control that wraps around an item of content, and provides context menu items for that item of content. Therefore, context menus are implemented for a `CarouselView` by creating a `SwipeView` that defines the content that the `SwipeView` wraps around, and the context menu items that are revealed by the swipe gesture. This is achieved by adding a `SwipeView` to the `DataTemplate` that defines the appearance of each item of data in the `CarouselView`:

```

<CarouselView x:Name="carouselView"
    ItemsSource="{Binding Monkeys}">
    <CarouselView.ItemTemplate>
        <DataTemplate>
            <StackLayout>
                <Frame HasShadow="True"
                    BorderColor="DarkGray"
                    CornerRadius="5"
                    Margin="20"
                    HeightRequest="300"
                    HorizontalOptions="Center"
                    VerticalOptions="CenterAndExpand">
                    <SwipeView>
                        <SwipeView.TopItems>
                            <SwipeItems>
                                <SwipeItem Text="Favorite"
                                    IconImageSource="favorite.png"
                                    BackgroundColor="LightGreen"
                                    Command="{Binding Source={x:Reference carouselView},
Path=BindingContext.FavoriteCommand}"
                                    CommandParameter="{Binding}" />
                            </SwipeItems>
                        </SwipeView.TopItems>
                        <SwipeView.BottomItems>
                            <SwipeItems>
                                <SwipeItem Text="Delete"
                                    IconImageSource="delete.png"
                                    BackgroundColor="LightPink"
                                    Command="{Binding Source={x:Reference carouselView},
Path=BindingContext.DeleteCommand}"
                                    CommandParameter="{Binding}" />
                            </SwipeItems>
                        </SwipeView.BottomItems>
                    <StackLayout>
                        <!-- Define item appearance -->
                    </StackLayout>
                </SwipeView>
            </Frame>
        </StackLayout>
    </DataTemplate>
</CarouselView.ItemTemplate>
</CarouselView>

```

The equivalent C# code is:

```

CarouselView carouselView = new CarouselView();
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");

carouselView.ItemTemplate = new DataTemplate(() =>
{
    StackLayout stackLayout = new StackLayout();
    Frame frame = new Frame { ... };

    SwipeView swipeView = new SwipeView();
    SwipeItem favoriteSwipeItem = new SwipeItem
    {
        Text = "Favorite",
        IconImageSource = "favorite.png",
        BackgroundColor = Color.LightGreen
    };
    favoriteSwipeItem.SetBinding(MenuItem.CommandProperty, new Binding("BindingContext.FavoriteCommand", source: carouselView));
    favoriteSwipeItem.SetBinding(MenuItem.CommandParameterProperty, ".");

    SwipeItem deleteSwipeItem = new SwipeItem
    {
        Text = "Delete",
        IconImageSource = "delete.png",
        BackgroundColor = Color.LightPink
    };
    deleteSwipeItem.SetBinding(MenuItem.CommandProperty, new Binding("BindingContext.DeleteCommand", source: carouselView));
    deleteSwipeItem.SetBinding(MenuItem.CommandParameterProperty, ".");

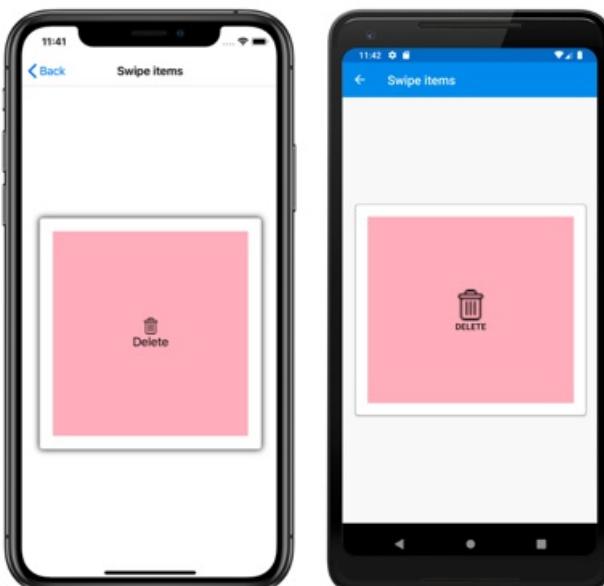
    swipeView.TopItems = new SwipeItems { favoriteSwipeItem };
    swipeView.BottomItems = new SwipeItems { deleteSwipeItem };

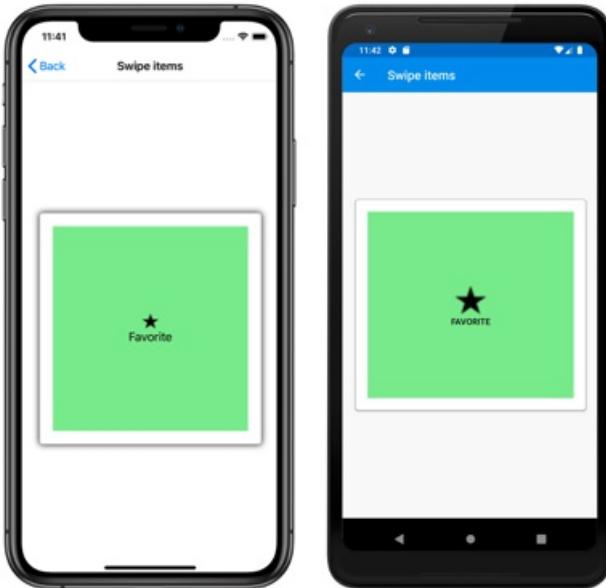
    StackLayout swipeViewStackLayout = new StackLayout { ... };
    swipeView.Content = swipeViewStackLayout;
    frame.Content = swipeView;
    stackLayout.Children.Add(frame);

    return stackLayout;
});

```

In this example, the `SwipeView` content is a `StackLayout` that defines the appearance of each item that's surrounded by a `Frame` in the `CarouselView`. The swipe items are used to perform actions on the `SwipeView` content, and are revealed when the control is swiped from the top and from the bottom:





`SwipeView` supports four different swipe directions, with the swipe direction being defined by the directional `SwipeItems` collection the `SwipeItems` objects are added to. By default, a swipe item is executed when it's tapped by the user. In addition, once a swipe item has been executed the swipe items are hidden and the `SwipeView` content is re-displayed. However, these behaviors can be changed.

For more information about the `SwipeView` control, see [Xamarin.Forms SwipeView](#).

Pull to refresh

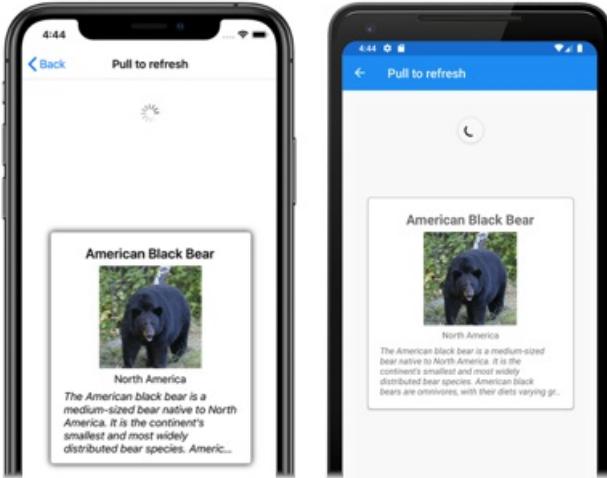
`CarouselView` supports pull to refresh functionality through the `RefreshView`, which enables the data being displayed to be refreshed by pulling down on the items. The `RefreshView` is a container control that provides pull to refresh functionality to its child, provided that the child supports scrollable content. Therefore, pull to refresh is implemented for a `CarouselView` by setting it as the child of a `RefreshView`:

```
<RefreshView IsRefreshing="{Binding IsRefreshing}"  
            Command="{Binding RefreshCommand}">  
    <CarouselView ItemsSource="{Binding Animals}">  
        ...  
    </CarouselView>  
</RefreshView>
```

The equivalent C# code is:

```
RefreshView refreshView = new RefreshView();  
ICommand refreshCommand = new Command(() =>  
{  
    // IsRefreshing is true  
    // Refresh data here  
    refreshView.IsRefreshing = false;  
});  
refreshView.Command = refreshCommand;  
  
CarouselView carouselView = new CarouselView();  
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Animals");  
refreshView.Content = carouselView;  
// ...
```

When the user initiates a refresh, the `ICommand` defined by the `Command` property is executed, which should refresh the items being displayed. A refresh visualization is shown while the refresh occurs, which consists of an animated progress circle:



The value of the `RefreshView.IsRefreshing` property indicates the current state of the `RefreshView`. When a refresh is triggered by the user, this property will automatically transition to `true`. Once the refresh completes, you should reset the property to `false`.

For more information about `RefreshView`, see [Xamarin.Forms RefreshView](#).

Load data incrementally

`CarouselView` supports incremental data virtualization as the user scrolls. This enables scenarios such as asynchronously loading a page of data from a web service, as the user scrolls. In addition, the point at which more data is loaded is configurable so that users don't see blank space, or are stopped from scrolling.

`CarouselView` defines the following properties to control incremental loading of data:

- `RemainingItemsThreshold`, of type `int`, the threshold of items not yet visible in the list at which the `RemainingItemsThresholdReached` event will be fired.
- `RemainingItemsThresholdReachedCommand`, of type `ICommand`, which is executed when the `RemainingItemsThreshold` is reached.
- `RemainingItemsThresholdReachedCommandParameter`, of type `object`, which is the parameter that's passed to the `RemainingItemsThresholdReachedCommand`.

`CarouselView` also defines a `RemainingItemsThresholdReached` event that is fired when the `CarouselView` is scrolled far enough that `RemainingItemsThreshold` items have not been displayed. This event can be handled to load more items. In addition, when the `RemainingItemsThresholdReached` event is fired, the `RemainingItemsThresholdReachedCommand` is executed, enabling incremental data loading to take place in a viewmodel.

The default value of the `RemainingItemsThreshold` property is -1, which indicates that the `RemainingItemsThresholdReached` event will never be fired. When the property value is 0, the `RemainingItemsThresholdReached` event will be fired when the final item in the `ItemsSource` is displayed. For values greater than 0, the `RemainingItemsThresholdReached` event will be fired when the `ItemsSource` contains that number of items not yet scrolled to.

NOTE

`CarouselView` validates the `RemainingItemsThreshold` property so that its value is always greater than or equal to -1.

The following XAML example shows a `CarouselView` that loads data incrementally:

```
<CarouselView ItemsSource="{Binding Animals}"
    RemainingItemsThreshold="2"
    RemainingItemsThresholdReached="OnCarouselViewRemainingItemsThresholdReached"
    RemainingItemsThresholdReachedCommand="{Binding LoadMoreDataCommand}">
...
</CarouselView>
```

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView
{
    RemainingItemsThreshold = 2
};
carouselView.RemainingItemsThresholdReached += OnCollectionViewRemainingItemsThresholdReached;
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Animals");
```

In this code example, the `RemainingItemsThresholdReached` event fires when there are 2 items not yet scrolled to, and in response executes the `OnCollectionViewRemainingItemsThresholdReached` event handler:

```
void OnCollectionViewRemainingItemsThresholdReached(object sender, EventArgs e)
{
    // Retrieve more data here and add it to the CollectionView's ItemsSource collection.
}
```

NOTE

Data can also be loaded incrementally by binding the `RemainingItemsThresholdReachedCommand` to an `ICommand` implementation in the viewmodel.

Related links

- [CarouselView \(sample\)](#)
- [Xamarin.Forms IndicatorView](#)
- [Xamarin.Forms RefreshView](#)
- [Xamarin.Forms Data Binding](#)
- [Xamarin.Forms Data Templates](#)
- [Create a Xamarin.Forms DataTemplateSelector](#)

Xamarin.Forms CarouselView Layout

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

`CarouselView` defines the following properties that control layout:

- `ItemsLayout`, of type `LinearItemsLayout`, specifies the layout to be used.
- `PeekAreaInsets`, of type `Thickness`, specifies how much to make adjacent items partially visible by.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

By default, a `CarouselView` will display its items in a horizontal orientation. A single item will be displayed on screen, with swipe gestures resulting in forwards and backwards navigation through the collection of items. However, a vertical orientation is also possible. This is because the `ItemsLayout` property is of type `LinearItemsLayout`, which inherits from the `ItemsLayout` class. The `ItemsLayout` class defines the following properties:

- `Orientation`, of type `ItemsLayoutOrientation`, specifies the direction in which the `CarouselView` expands as items are added.
- `SnapPointsAlignment`, of type `SnapPointsAlignment`, specifies how snap points are aligned with items.
- `SnapPointsType`, of type `SnapPointsType`, specifies the behavior of snap points when scrolling.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings. For more information about snap points, see [Snap points](#) in the [Xamarin.Forms CollectionView Scrolling](#) guide.

The `ItemsLayoutOrientation` enumeration defines the following members:

- `Vertical` indicates that the `CarouselView` will expand vertically as items are added.
- `Horizontal` indicates that the `CarouselView` will expand horizontally as items are added.

The `LinearItemsLayout` class inherits from the `ItemsLayout` class, and defines an `ItemSpacing` property, of type `double`, that represents the empty space around each item. The default value of this property is 0, and its value must always be greater than or equal to 0. The `LinearItemsLayout` class also defines static `Vertical` and `Horizontal` members. These members can be used to create vertical or horizontal lists, respectively.

Alternatively, a `LinearItemsLayout` object can be created, specifying an `ItemsLayoutOrientation` enumeration member as an argument.

NOTE

`CarouselView` uses the native layout engines to perform layout.

Horizontal layout

By default, `CarouselView` will display its items horizontally. Therefore, it's not necessary to set the `ItemsLayout` property to use this layout:

```

<CarouselView ItemsSource="{Binding Monkeys}">
    <CarouselView.ItemTemplate>
        <DataTemplate>
            <StackLayout>
                <Frame HasShadow="True"
                    BorderColor="DarkGray"
                    CornerRadius="5"
                    Margin="20"
                    HeightRequest="300"
                    HorizontalOptions="Center"
                    VerticalOptions="CenterAndExpand">
                    <StackLayout>
                        <Label Text="{Binding Name}"
                            FontAttributes="Bold"
                            FontSize="Large"
                            HorizontalOptions="Center"
                            VerticalOptions="Center" />
                        <Image Source="{Binding ImageUrl}"
                            Aspect="Aspectfill"
                            HeightRequest="150"
                            WidthRequest="150"
                            HorizontalOptions="Center" />
                        <Label Text="{Binding Location}"
                            HorizontalOptions="Center" />
                        <Label Text="{Binding Details}"
                            FontAttributes="Italic"
                            HorizontalOptions="Center"
                            MaxLines="5"
                            LineBreakMode="TailTruncation" />
                    </StackLayout>
                </Frame>
            </StackLayout>
        </DataTemplate>
    </CarouselView.ItemTemplate>
</CarouselView>

```

Alternatively, this layout can also be accomplished by setting the `ItemsLayout` property to a `LinearItemsLayout` object, specifying the `Horizontal` `ItemsLayoutOrientation` enumeration member as the `Orientation` property value:

```

<CarouselView ItemsSource="{Binding Monkeys}">
    <CarouselView.ItemsLayout>
        <LinearItemsLayout Orientation="Horizontal" />
    </CarouselView.ItemsLayout>
    ...
</CarouselView>

```

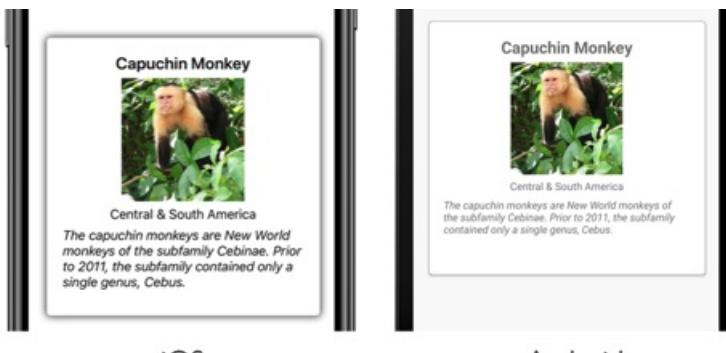
The equivalent C# code is:

```

CarouselView carouselView = new CarouselView
{
    ...
    ItemsLayout = LinearItemsLayout.Horizontal
};

```

This results in a layout that grows horizontally as new items are added:



iOS

Android

Vertical layout

`CarouselView` can display its items vertically by setting the `ItemsLayout` property to a `LinearItemsLayout` object, specifying the `Vertical` `ItemsLayoutOrientation` enumeration member as the `Orientation` property value:

```
<CarouselView ItemsSource="{Binding Monkeys}">
    <CarouselView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical" />
    </CarouselView.ItemsLayout>
    <CarouselView.ItemTemplate>
        <DataTemplate>
            <StackLayout>
                <Frame HasShadow="True"
                    BorderColor="DarkGray"
                    CornerRadius="5"
                    Margin="20"
                    HeightRequest="300"
                    HorizontalOptions="Center"
                    VerticalOptions="CenterAndExpand">
                    <StackLayout>
                        <Label Text="{Binding Name}"
                            FontAttributes="Bold"
                            FontSize="Large"
                            HorizontalOptions="Center"
                            VerticalOptions="Center" />
                        <Image Source="{Binding ImageUrl}"
                            Aspect="AspectFill"
                            HeightRequest="150"
                            WidthRequest="150"
                            HorizontalOptions="Center" />
                        <Label Text="{Binding Location}"
                            HorizontalOptions="Center" />
                        <Label Text="{Binding Details}"
                            FontAttributes="Italic"
                            HorizontalOptions="Center"
                            MaxLines="5"
                            LineBreakMode="TailTruncation" />
                    </StackLayout>
                </Frame>
            </StackLayout>
        </DataTemplate>
    </CarouselView.ItemTemplate>
</CarouselView>
```

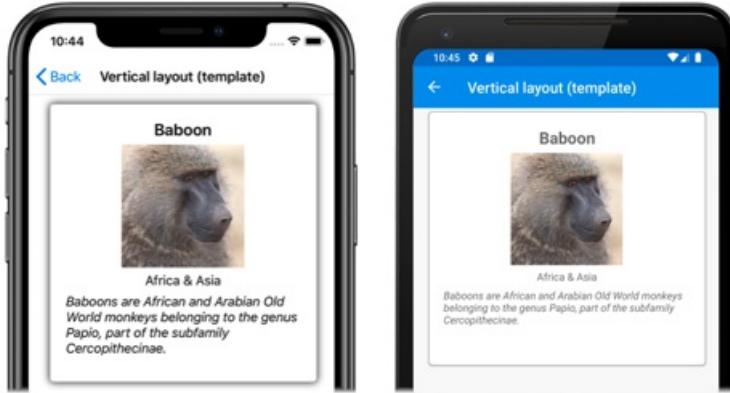
The equivalent C# code is:

```

CarouselView carouselView = new CarouselView
{
    ...
    ItemsLayout = LinearItemsLayout.Vertical
};

```

This results in a layout that grows vertically as new items are added:



Partially visible adjacent items

By default, `CarouselView` displays full items at once. However, this behavior can be changed by setting the `PeekAreaInsets` property to a `Thickness` value that specifies how much to make adjacent items partially visible by. This can be useful to indicate to users that there are additional items to view. The following XAML shows an example of setting this property:

```

<CarouselView ItemsSource="{Binding Monkeys}"
              PeekAreaInsets="100">
    ...
</CarouselView>

```

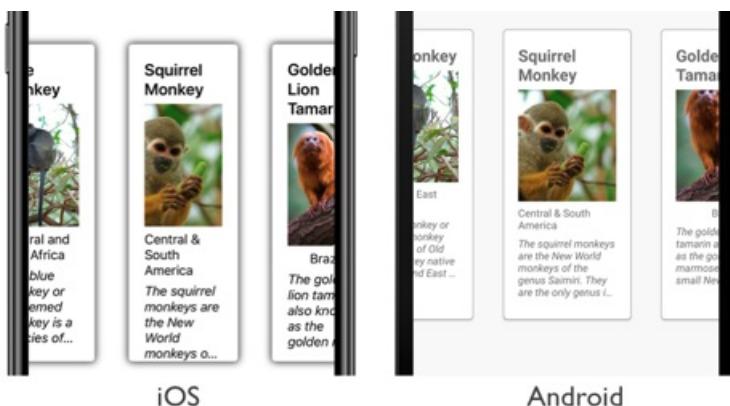
The equivalent C# code is:

```

CarouselView carouselView = new CarouselView
{
    ...
    PeekAreaInsets = new Thickness(100)
};

```

The result is that adjacent items are partially exposed on screen:



Item spacing

By default, there is no space between each item in a `CarouselView`. This behavior can be changed by setting the `ItemSpacing` property on the items layout used by the `CarouselView`.

When a `CarouselView` sets its `ItemsLayout` property to a `LinearItemsLayout` object, the `LinearItemsLayout.ItemSpacing` property can be set to a `double` value that represents the space between items:

```
<CarouselView ItemsSource="{Binding Monkeys}">
    <CarouselView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical"
            ItemSpacing="20" />
    </CarouselView.ItemsLayout>
    ...
</CarouselView>
```

NOTE

The `LinearItemsLayout.ItemSpacing` property has a validation callback set, which ensures that the value of the property is always greater than or equal to 0.

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView
{
    ...
    ItemsLayout = new LinearItemsLayout(ItemsLayoutOrientation.Vertical)
    {
        ItemSpacing = 20
    }
};
```

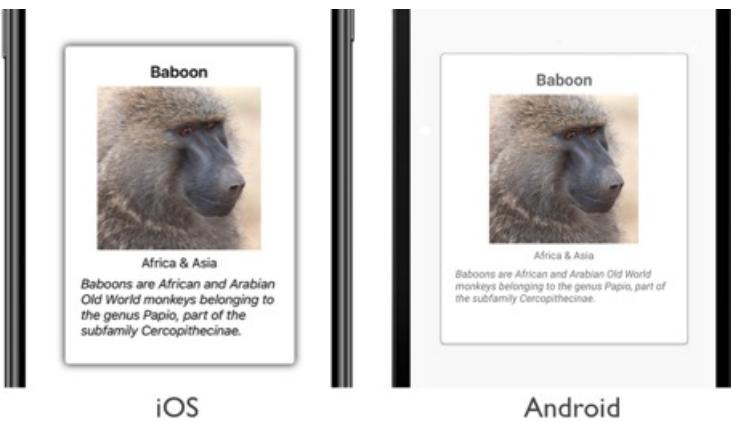
This code results in a vertical layout, that has a spacing of 20 between items.

Dynamic resizing of items

Items in a `CarouselView` can be dynamically resized at runtime by changing layout related properties of elements within the `DataTemplate`. For example, the following code example changes the `HeightRequest` and `WidthRequest` properties of an `Image` object, and the `HeightRequest` property of its parent `Frame`:

```
void OnImageTapped(object sender, EventArgs e)
{
    Image image = sender as Image;
    image.HeightRequest = image.WidthRequest = image.HeightRequest.Equals(150) ? 200 : 150;
    Frame frame = ((Frame)image.Parent.Parent);
    frame.HeightRequest = frame.HeightRequest.Equals(300) ? 350 : 300;
}
```

The `OnImageTapped` event handler is executed in response to an `Image` object being tapped, and changes the dimensions of the image (and its parent `Frame`), so that it's more easily viewed:



Right-to-left layout

`CarouselView` can layout its content in a right-to-left flow direction by setting its `FlowDirection` property to `RightToLeft`. However, the `FlowDirection` property should ideally be set on a page or root layout, which causes all the elements within the page, or root layout, to respond to the flow direction:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="CarouselViewDemos.Views.HorizontalTemplateLayoutRTLPage"
    Title="Horizontal layout (RTL FlowDirection)"
    FlowDirection="RightToLeft">
    <CarouselView ItemsSource="{Binding Monkeys}">
        ...
    </CarouselView>
</ContentPage>
```

The default `FlowDirection` for an element with a parent is `MatchParent`. Therefore, the `CarouselView` inherits the `FlowDirection` property value from the `ContentPage`.

For more information about flow direction, see [Right-to-left localization](#).

Related links

- [CarouselView \(sample\)](#)
- [Right-to-left localization](#)
- [Xamarin.Forms CarouselView Scrolling](#)

Xamarin.Forms CarouselView Interaction

8/4/2022 • 8 minutes to read • [Edit Online](#)



[Download the sample](#)

`CarouselView` defines the following properties that control user interaction:

- `CurrentItem`, of type `object`, the current item being displayed. This property has a default binding mode of `TwoWay`, and has a `null` value when there isn't any data to display.
- `CurrentItemChangedCommand`, of type `ICommand`, which is executed when the current item changes.
- `CurrentItemChangedCommandParameter`, of type `object`, which is the parameter that's passed to the `CurrentItemChangedCommand`.
- `IsBounceEnabled`, of type `bool`, which specifies whether the `CarouselView` will bounce at a content boundary. The default value is `true`.
- `IsSwipeEnabled`, of type `bool`, which determines whether a swipe gesture will change the displayed item. The default value is `true`.
- `Loop`, of type `bool`, which determines whether the `CarouselView` provides looped access to its collection of items. The default value is `true`.
- `Position`, of type `int`, the index of the current item in the underlying collection. This property has a default binding mode of `TwoWay`, and has a 0 value when there isn't any data to display.
- `PositionChangedCommand`, of type `ICommand`, which is executed when the position changes.
- `PositionChangedCommandParameter`, of type `object`, which is the parameter that's passed to the `PositionChangedCommand`.
- `VisibleViews`, of type `ObservableCollection<View>`, which is a read-only property that contains the objects for the items that are currently visible.

All of these properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

`CarouselView` defines a `CurrentItemChanged` event that's fired when the `CurrentItem` property changes, either due to user scrolling, or when an application sets the property. The `CurrentItemChangedEventArgs` object that accompanies the `CurrentItemChanged` event has two properties, both of type `object`:

- `PreviousItem` – the previous item, after the property change.
- `CurrentItem` – the current item, after the property change.

`CarouselView` also defines a `PositionChanged` event that's fired when the `Position` property changes, either due to user scrolling, or when an application sets the property. The `PositionChangedEventArgs` object that accompanies the `PositionChanged` event has two properties, both of type `int`:

- `PreviousPosition` – the previous position, after the property change.
- `CurrentPosition` – the current position, after the property change.

Respond to the current item changing

When the currently displayed item changes, the `CurrentItem` property will be set to the value of the item. When this property changes, the `CurrentItemChangedCommand` is executed with the value of the `CurrentItemChangedCommandParameter` being passed to the `ICommand`. The `Position` property is then updated, and the `CurrentItemChanged` event fires.

IMPORTANT

The `Position` property changes when the `CurrentItem` property changes. This will result in the `PositionChangedCommand` being executed, and the `PositionChanged` event firing.

Event

The following XAML example shows a `CarouselView` that uses an event handler to respond to the current item changing:

```
<CarouselView ItemsSource="{Binding Monkeys}"
    CurrentItemChanged="OnCurrentItemChanged">
    ...
</CarouselView>
```

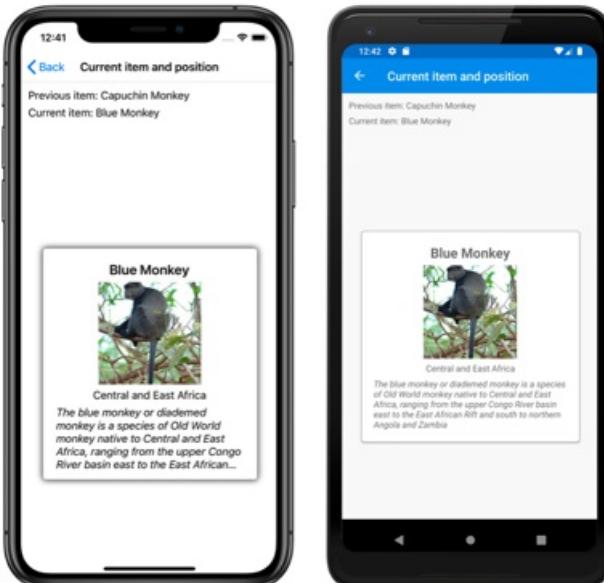
The equivalent C# code is:

```
CarouselView carouselView = new CarouselView();
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
carouselView.CurrentItemChanged += OnCurrentItemChanged;
```

In this example, the `OnCurrentItemChanged` event handler is executed when the `CurrentItemChanged` event fires:

```
void OnCurrentItemChanged(object sender, CurrentItemChangedEventArgs e)
{
    Monkey previousItem = e.PreviousItem as Monkey;
    Monkey currentItem = e.CurrentItem as Monkey;
}
```

In this example, the `OnCurrentItemChanged` event handler exposes the previous and current items:



Command

The following XAML example shows a `CarouselView` that uses a command to respond to the current item changing:

```

<CarouselView ItemsSource="{Binding Monkeys}"
    CurrentItemChangedCommand="{Binding ItemChangedCommand}"
    CurrentItemChangedCommandParameter="{Binding Source={RelativeSource Self}, Path=CurrentItem}">
...
</CarouselView>

```

The equivalent C# code is:

```

CarouselView carouselView = new CarouselView();
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
carouselView.SetBinding(CarouselView.CurrentItemChangedCommandProperty, "ItemChangedCommand");
carouselView.SetBinding(CarouselView.CurrentItemChangedCommandParameterProperty, new Binding("CurrentItem",
source: RelativeBindingSource.Self));

```

In this example, the `CurrentItemChangedCommand` property binds to the `ItemChangedCommand` property, passing the `CurrentItem` property value to it as an argument. The `ItemChangedCommand` can then respond to the current item changing, as required:

```

public ICommand ItemChangedCommand => new Command<Monkey>((item) =>
{
    PreviousMonkey = CurrentMonkey;
    CurrentMonkey = item;
});

```

In this example, the `ItemChangedCommand` updates objects that store the previous and current items.

Respond to the position changing

When the currently displayed item changes, the `Position` property will be set to the index of the current item in the underlying collection. When this property changes, the `PositionChangedCommand` is executed with the value of the `PositionChangedCommandParameter` being passed to the `ICommand`. The `PositionChanged` event then fires. If the `Position` property has been programmatically changed, the `CarouselView` will be scrolled to the item that corresponds to the `Position` value.

NOTE

Setting the `Position` property to 0 will result in the first item in the underlying collection being displayed.

Event

The following XAML example shows a `CarouselView` that uses an event handler to respond to the `Position` property changing:

```

<CarouselView ItemsSource="{Binding Monkeys}"
    PositionChanged="OnPositionChanged">
...
</CarouselView>

```

The equivalent C# code is:

```

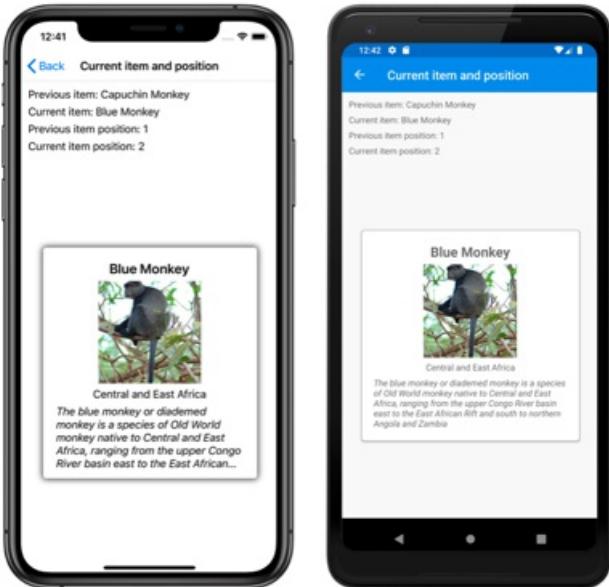
CarouselView carouselView = new CarouselView();
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
carouselView.PositionChanged += OnPositionChanged;

```

In this example, the `OnPositionChanged` event handler is executed when the `PositionChanged` event fires:

```
void OnPositionChanged(object sender, PositionChangedEventArgs e)
{
    int previousItemPosition = e.PreviousPosition;
    int currentItemPosition = e.CurrentPosition;
}
```

In this example, the `OnCurrentItemChanged` event handler exposes the previous and current positions:



Command

The following XAML example shows a `CarouselView` that uses a command to respond to the `Position` property changing:

```
<CarouselView ItemsSource="{Binding Monkeys}"
    PositionChangedCommand="{Binding PositionChangedCommand}"
    PositionChangedCommandParameter="{Binding Source={RelativeSource Self}, Path=Position}"
    ...
/>
```

The equivalent C# code is:

```
CarouselView carouseView = new CarouselView();
carouseView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
carouseView.SetBinding(CarouselView.PositionChangedCommandProperty, "PositionChangedCommand");
carouseView.SetBinding(CarouselView.PositionChangedCommandParameterProperty, new Binding("Position",
source: RelativeBindingSource.Self));
```

In this example, the `PositionChangedCommand` property binds to the `PositionChangedCommand` property, passing the `Position` property value to it as an argument. The `PositionChangedCommand` can then respond to the position changing, as required:

```
public ICommand PositionChangedCommand => new Command<int>((position) =>
{
    PreviousPosition = CurrentPosition;
    CurrentPosition = position;
});
```

In this example, the `PositionChangedCommand` updates objects that store the previous and current positions.

Preset the current item

The current item in a `CarouselView` can be programmatically set by setting the `CurrentItem` property to the item. The following XAML example shows a `CarouselView` that pre-chooses the current item:

```
<CarouselView ItemsSource="{Binding Monkeys}"
    CurrentItem="{Binding CurrentItem}">
    ...
</CarouselView>
```

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView();
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
carouselView.SetBinding(CarouselView.CurrentItemProperty, "CurrentItem");
```

NOTE

The `CurrentItem` property has a default binding mode of `TwoWay`.

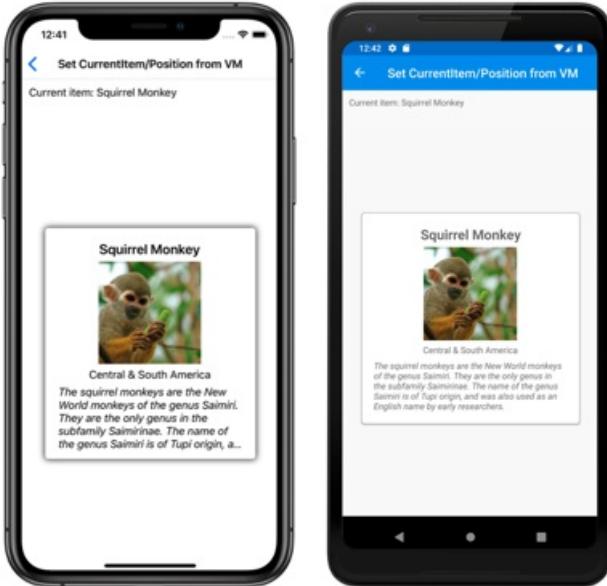
The `carouselView.CurrentItem` property data binds to the `CurrentItem` property of the connected view model, which is of type `Monkey`. By default, a `TwoWay` binding is used so that if the user changes the current item, the value of the `CurrentItem` property will be set to the current `Monkey` object. The `CurrentItem` property is defined in the `MonkeysViewModel` class:

```
public class MonkeysViewModel : INotifyPropertyChanged
{
    // ...
    public ObservableCollection<Monkey> Monkeys { get; private set; }

    public Monkey CurrentItem { get; set; }

    public MonkeysViewModel()
    {
        // ...
        CurrentItem = Monkeys.Skip(3).FirstOrDefault();
        OnPropertyChanged("CurrentItem");
    }
}
```

In this example, the `CurrentItem` property is set to the fourth item in the `Monkeys` collection:



Preset the position

The displayed item `CarouselView` can be programmatically set by setting the `Position` property to the index of the item in the underlying collection. The following XAML example shows a `CarouselView` that sets the displayed item:

```
<CarouselView ItemsSource="{Binding Monkeys}"
              Position="{Binding Position}">
    ...
</CarouselView>
```

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView();
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
carouselView.SetBinding(CarouselView.PositionProperty, "Position");
```

NOTE

The `Position` property has a default binding mode of `TwoWay`.

The `CarouselView.Position` property data binds to the `Position` property of the connected view model, which is of type `int`. By default, a `TwoWay` binding is used so that if the user scrolls through the `CarouselView`, the value of the `Position` property will be set to the index of the displayed item. The `Position` property is defined in the `MonkeysViewModel` class:

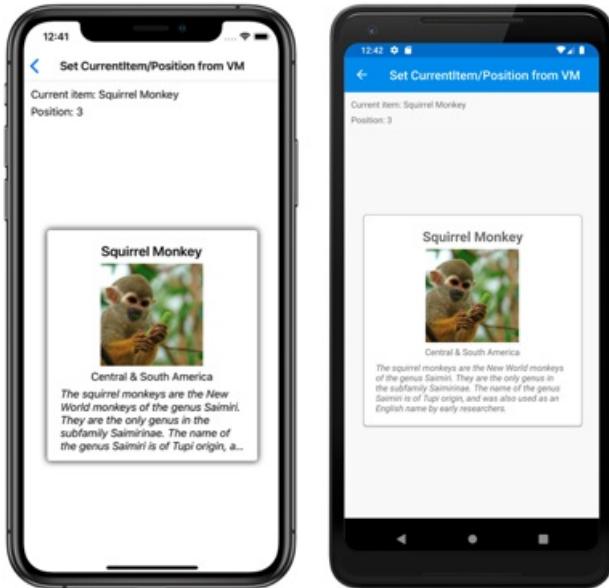
```

public class MonkeysViewModel : INotifyPropertyChanged
{
    // ...
    public int Position { get; set; }

    public MonkeysViewModel()
    {
        // ...
        Position = 3;
        OnPropertyChanged("Position");
    }
}

```

In this example, the `Position` property is set to the fourth item in the `Monkeys` collection:



Define visual states

`CarouselView` defines four visual states:

- `CurrentItem` represents the visual state for the currently displayed item.
- `PreviousItem` represents the visual state for the previously displayed item.
- `NextItem` represents the visual state for the next item.
- `DefaultItem` represents the visual state for the remainder of the items.

These visual states can be used to initiate visual changes to the items displayed by the `CarouselView`.

The following XAML example shows how to define the `CurrentItem`, `PreviousItem`, `NextItem`, and `DefaultItem` visual states:

```

<CarouselView ItemsSource="{Binding Monkeys}"
    PeekAreaInsets="100">
    <CarouselView.ItemTemplate>
        <DataTemplate>
            <StackLayout>
                <VisualStateManager.VisualStateGroups>
                    <VisualStateGroup x:Name="CommonStates">
                        <VisualState x:Name="CurrentItem">
                            <VisualState.Setters>
                                <Setter Property="Scale"
                                    Value="1.1" />
                            </VisualState.Setters>
                        </VisualState>
                        <VisualState x:Name="PreviousItem">
                            <VisualState.Setters>
                                <Setter Property="Opacity"
                                    Value="0.5" />
                            </VisualState.Setters>
                        </VisualState>
                        <VisualState x:Name="NextItem">
                            <VisualState.Setters>
                                <Setter Property="Opacity"
                                    Value="0.5" />
                            </VisualState.Setters>
                        </VisualState>
                        <VisualState x:Name="DefaultItem">
                            <VisualState.Setters>
                                <Setter Property="Opacity"
                                    Value="0.25" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
                </VisualStateManager.VisualStateGroups>

                <!-- Item template content -->
                <Frame HasShadow="true">
                    ...
                </Frame>
            </StackLayout>
        </DataTemplate>
    </CarouselView.ItemTemplate>
</CarouselView>

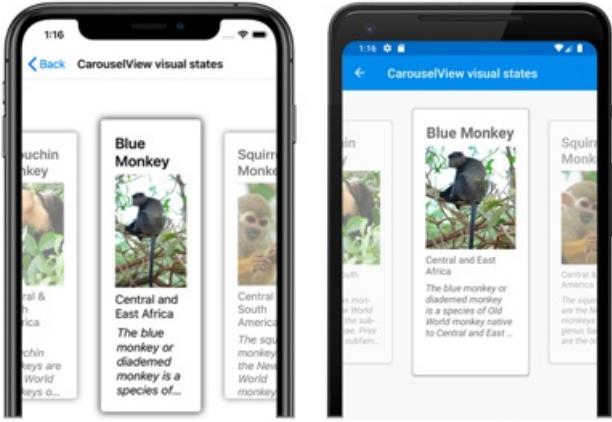
```

In this example, the `CurrentItem` visual state specifies that the current item displayed by the `CarouselView` will have its `Scale` property changed from its default value of 1 to 1.1. The `PreviousItem` and `NextItem` visual states specify that the items surrounding the current item will be displayed with an `Opacity` value of 0.5. The `DefaultItem` visual state specifies that the remainder of the items displayed by the `CarouselView` will be displayed with an `Opacity` value of 0.25.

NOTE

Alternatively, the visual states can be defined in a `Style` that has a `TargetType` property value that's the type of the root element of the `DataTemplate`, which is set as the `ItemTemplate` property value.

The following screenshots show the `CurrentItem`, `PreviousItem`, and `NextItem` visual states:



For more information about visual states, see [Xamarin.Forms Visual State Manager](#).

Clear the current item

The `CurrentItem` property can be cleared by setting it, or the object it binds to, to `null`.

Disable bounce

By default, `CarouselView` bounces items at content boundaries. This can be disabled by setting the `IsBounceEnabled` property to `false`.

Disable loop

By default, `CarouselView` provides looped access to its collection of items. Therefore, swiping backwards from the first item in the collection will display the last item in the collection. Similarly, swiping forwards from the last item in the collection will return to the first item in the collection. This behavior can be disabled by setting the `Loop` property to `false`.

Disable swipe interaction

By default, `CarouselView` allows users to move through items using a swipe gesture. This swipe interaction can be disabled by setting the `IsSwipeEnabled` property to `false`.

Related links

- [CarouselView \(sample\)](#)
- [Xamarin.Forms Visual State Manager](#)

Xamarin.Forms CarouselView EmptyView

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

`CarouselView` defines the following properties that can be used to provide user feedback when there's no data to display:

- `EmptyView`, of type `object`, the string, binding, or view that will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The default value is `null`.
- `EmptyViewTemplate`, of type `DataTemplate`, the template to use to format the specified `EmptyView`. The default value is `null`.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

The main usage scenarios for setting the `EmptyView` property are displaying user feedback when a filtering operation on a `CarouselView` yields no data, and displaying user feedback while data is being retrieved from a web service.

NOTE

The `EmptyView` property can be set to a view that includes interactive content if required.

For more information about data templates, see [Xamarin.Forms Data Templates](#).

Display a string when data is unavailable

The `EmptyView` property can be set to a string, which will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The following XAML shows an example of this scenario:

```
<CarouselView ItemsSource="{Binding EmptyMonkeys}"
              EmptyView="No items to display." />
```

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView
{
    EmptyView = "No items to display."
};
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "EmptyMonkeys");
```

The result is that, because the data bound collection is `null`, the string set as the `EmptyView` property value is displayed.

Display views when data is unavailable

The `EmptyView` property can be set to a view, which will be displayed when the `ItemsSource` property is `null`,

or when the collection specified by the `ItemsSource` property is `null` or empty. This can be a single view, or a view that contains multiple child views. The following XAML example shows the `EmptyView` property set to a view that contains multiple child views:

```
<StackLayout Margin="20">
    <SearchBar SearchCommand="{Binding FilterCommand}"
        SearchCommandParameter="{Binding Source={RelativeSource Self}, Path=Text}"
        Placeholder="Filter" />
    <CarouselView ItemsSource="{Binding Monkeys}">
        <CarouselView.EmptyView>
            <ContentView>
                <StackLayout HorizontalOptions="CenterAndExpand"
                    VerticalOptions="CenterAndExpand">
                    <Label Text="No results matched your filter."
                        Margin="10,25,10,10"
                        FontAttributes="Bold"
                        FontSize="18"
                        HorizontalOptions="Fill"
                        HorizontalTextAlignment="Center" />
                    <Label Text="Try a broader filter?"
                        FontAttributes="Italic"
                        FontSize="12"
                        HorizontalOptions="Fill"
                        HorizontalTextAlignment="Center" />
                </StackLayout>
            </ContentView>
        </CarouselView.EmptyView>
        <CarouselView.ItemTemplate>
            ...
        </CarouselView.ItemTemplate>
    </CarouselView>
</StackLayout>
```

In this example, what looks like a redundant `ContentView` has been added as the root element of the `EmptyView`. This is because internally, the `EmptyView` is added to a native container that doesn't provide any context for Xamarin.Forms layout. Therefore, to position the views that comprise your `EmptyView`, you must add a root layout, whose child is a layout that can position itself within the root layout.

The equivalent C# code is:

```
SearchBar searchBar = new SearchBar { ... };
CarouselView carouselView = new CarouselView
{
    EmptyView = new ContentView
    {
        Content = new StackLayout
        {
            Children =
            {
                new Label { Text = "No results matched your filter.", ... },
                new Label { Text = "Try a broader filter?", ... }
            }
        }
    }
};
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
```

When the `SearchBar` executes the `FilterCommand`, the collection displayed by the `CarouselView` is filtered for the search term stored in the `SearchBar.Text` property. If the filtering operation yields no data, the `StackLayout` set as the `EmptyView` property value is displayed.

Display a templated custom type when data is unavailable

The `EmptyView` property can be set to a custom type, whose template is displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The `EmptyViewTemplate` property can be set to a `DataTemplate` that defines the appearance of the `EmptyView`. The following XAML shows an example of this scenario:

```
<StackLayout Margin="20">
    <SearchBar x:Name="searchBar"
        SearchCommand="{Binding FilterCommand}"
        SearchCommandParameter="{Binding Source={RelativeSource Self}, Path=Text}"
        Placeholder="Filter" />
    <CarouselView ItemsSource="{Binding Monkeys}">
        <CarouselView.EmptyView>
            <controls:FilterData Filter="{Binding Source={x:Reference searchBar}, Path=Text}" />
        </CarouselView.EmptyView>
        <CarouselView.EmptyViewTemplate>
            <DataTemplate>
                <Label Text="{Binding Filter, StringFormat='Your filter term of {0} did not match any records.'}">
                    Margin="10,25,10,10"
                    FontAttributes="Bold"
                    FontSize="18"
                    HorizontalOptions="Fill"
                    HorizontalTextAlignment="Center" />
            </DataTemplate>
        </CarouselView.EmptyViewTemplate>
        <CarouselView.ItemTemplate>
            ...
        </CarouselView.ItemTemplate>
    </CarouselView>
</StackLayout>
```

The equivalent C# code is:

```
SearchBar searchBar = new SearchBar { ... };
CarouselView carouselView = new CarouselView
{
    EmptyView = new FilterData { Filter = searchBar.Text },
    EmptyViewTemplate = new DataTemplate(() =>
    {
        return new Label { ... };
    })
};
```

The `FilterData` type defines a `Filter` property, and a corresponding `BindableProperty`:

```
public class FilterData : BindableObject
{
    public static readonly BindableProperty FilterProperty = BindableProperty.Create(nameof(Filter),
        typeof(string), typeof(FilterData), null);

    public string Filter
    {
        get { return (string)GetValue(FilterProperty); }
        set { SetValue(FilterProperty, value); }
    }
}
```

The `EmptyView` property is set to a `FilterData` object, and the `Filter` property data binds to the `SearchBar.Text` property. When the `SearchBar` executes the `FilterCommand`, the collection displayed by the

`CarouselView` is filtered for the search term stored in the `Filter` property. If the filtering operation yields no data, the `Label` defined in the `DataTemplate`, that's set as the `EmptyViewTemplate` property value, is displayed.

NOTE

When displaying a templated custom type when data is unavailable, the `EmptyViewTemplate` property can be set to a view that contains multiple child views.

Choose an EmptyView at runtime

Views that will be displayed as an `EmptyView` when data is unavailable, can be defined as `ContentView` objects in a `ResourceDictionary`. The `EmptyView` property can then be set to a specific `ContentView`, based on some business logic, at runtime. The following XAML example shows an example of this scenario:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewmodels="clr-namespace:CarouselViewDemos.ViewModels"
    x:Class="CarouselViewDemos.Views.EmptyViewSwapPage"
    Title="EmptyView (swap)">
    <ContentPage.BindingContext>
        <viewmodels:MonkeysViewModel />
    </ContentPage.BindingContext>
    <ContentPage.Resources>
        <ContentView x:Key="BasicEmptyView">
            <StackLayout>
                <Label Text="No items to display."
                    Margin="10,25,10,10"
                    FontAttributes="Bold"
                    FontSize="18"
                    HorizontalOptions="Fill"
                    HorizontalTextAlignment="Center" />
            </StackLayout>
        </ContentView>
        <ContentView x:Key="AdvancedEmptyView">
            <StackLayout>
                <Label Text="No results matched your filter."
                    Margin="10,25,10,10"
                    FontAttributes="Bold"
                    FontSize="18"
                    HorizontalOptions="Fill"
                    HorizontalTextAlignment="Center" />
                <Label Text="Try a broader filter?"
                    FontAttributes="Italic"
                    FontSize="12"
                    HorizontalOptions="Fill"
                    HorizontalTextAlignment="Center" />
            </StackLayout>
        </ContentView>
    </ContentPage.Resources>
    <StackLayout Margin="20">
        <SearchBar SearchCommand="{Binding FilterCommand}"
            SearchCommandParameter="{Binding Source={RelativeSource Self}, Path=Text}"
            Placeholder="Filter" />
        <StackLayout Orientation="Horizontal">
            <Label Text="Toggle EmptyViews" />
            <Switch Toggled="OnEmptyViewSwitchToggled" />
        </StackLayout>
        <CarouselView x:Name="carouselView"
            ItemsSource="{Binding Monkeys}">
            <CarouselView.ItemTemplate>
                ...
            </CarouselView.ItemTemplate>
        </CarouselView>
    </StackLayout>
</ContentPage>

```

This XAML defines two `ContentView` objects in the page-level `ResourceDictionary`, with the `Switch` object controlling which `ContentView` object will be set as the `EmptyView` property value. When the `Switch` is toggled, the `OnEmptyViewSwitchToggled` event handler executes the `ToggleEmptyView` method:

```

void ToggleEmptyView(bool isToggled)
{
    carouselView.EmptyView = isToggled ? Resources["BasicEmptyView"] : Resources["AdvancedEmptyView"];
}

```

The `ToggleEmptyView` method sets the `EmptyView` property of the `carouselView` object to one of the two `ContentView` objects stored in the `ResourceDictionary`, based on the value of the `Switch.IsToggled` property.

When the `SearchBar` executes the `FilterCommand`, the collection displayed by the `CarouselView` is filtered for the search term stored in the `SearchBar.Text` property. If the filtering operation yields no data, the `ContentView` object set as the `EmptyView` property is displayed.

For more information about resource dictionaries, see [Xamarin.Forms Resource Dictionaries](#).

Choose an EmptyViewTemplate at runtime

The appearance of the `EmptyView` can be chosen at runtime, based on its value, by setting the `CarouselView.EmptyViewTemplate` property to a `DataTemplateSelector` object:

```
<ContentPage ...>
    <x:Bind xmlns:controls="clr-namespace:CarouselViewDemos.Controls">
        <ContentPage.Resources>
            <DataTemplate x:Key="AdvancedTemplate">
                ...
            </DataTemplate>

            <DataTemplate x:Key="BasicTemplate">
                ...
            </DataTemplate>

            <controls:SearchTermDataTemplateSelector x:Key="SearchSelector"
                DefaultTemplate="{StaticResource AdvancedTemplate}"
                OtherTemplate="{StaticResource BasicTemplate}" />
        </ContentPage.Resources>

        <StackLayout Margin="20">
            <SearchBar x:Name="searchBar"
                SearchCommand="{Binding FilterCommand}"
                SearchCommandParameter="{Binding Source={RelativeSource Self}, Path=Text}"
                Placeholder="Filter" />
            <CarouselView ItemsSource="{Binding Monkeys}">
                <CarouselView.EmptyView>
                    <Binding Source={x:Reference searchBar}, Path=Text>
                <CarouselView.EmptyView>
                <CarouselView.EmptyViewTemplate>
                    <StaticResource SearchSelector>
                <CarouselView.EmptyViewTemplate>
            <CarouselView.ItemTemplate>
                ...
            </CarouselView.ItemTemplate>
        </CarouselView>
    </StackLayout>
</ContentPage>
```

The equivalent C# code is:

```
SearchBar searchBar = new SearchBar { ... };
CarouselView carouselView = new CarouselView()
{
    EmptyView = searchBar.Text,
    EmptyViewTemplate = new SearchTermDataTemplateSelector { ... }
};
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
```

The `EmptyView` property is set to the `SearchBar.Text` property, and the `EmptyViewTemplate` property is set to a `SearchTermDataTemplateSelector` object.

When the `SearchBar` executes the `FilterCommand`, the collection displayed by the `CarouselView` is filtered for the search term stored in the `SearchBar.Text` property. If the filtering operation yields no data, the `DataTemplate` chosen by the `SearchTermDataTemplateSelector` object is set as the `EmptyViewTemplate` property and displayed.

The following example shows the `SearchTermDataTemplateSelector` class:

```
public class SearchTermDataTemplateSelector : DataTemplateSelector
{
    public DataTemplate DefaultTemplate { get; set; }
    public DataTemplate OtherTemplate { get; set; }

    protected override DataTemplate OnSelectTemplate(object item, BindableObject container)
    {
        string query = (string)item;
        return query.ToLower().Equals("xamarin") ? OtherTemplate : DefaultTemplate;
    }
}
```

The `SearchTermTemplateSelector` class defines `DefaultTemplate` and `OtherTemplate` `DataTemplate` properties that are set to different data templates. The `OnSelectTemplate` override returns `DefaultTemplate`, which displays a message to the user, when the search query isn't equal to "xamarin". When the search query is equal to "xamarin", the `OnSelectTemplate` override returns `OtherTemplate`, which displays a basic message to the user.

For more information about data template selectors, see [Create a Xamarin.Forms DataTemplateSelector](#).

Related links

- [CarouselView \(sample\)](#)
- [Xamarin.Forms Data Templates](#)
- [Xamarin.Forms Resource Dictionaries](#)
- [Create a Xamarin.Forms DataTemplateSelector](#)

Xamarin.Forms CarouselView Scrolling

8/4/2022 • 8 minutes to read • [Edit Online](#)



[Download the sample](#)

`CarouselView` defines the following scroll related properties:

- `HorizontalScrollBarVisibility`, of type `ScrollBarVisibility`, which specifies when the horizontal scroll bar is visible.
- `IsDragging`, of type `bool`, which indicates whether the `CarouselView` is scrolling. This is a read only property, whose default value is `false`.
- `IsScrollAnimated`, of type `bool`, which specifies whether an animation will occur when scrolling the `CarouselView`. The default value is `true`.
- `ItemsUpdatingScrollMode`, of type `ItemsUpdatingScrollMode`, which represents the scrolling behavior of the `CarouselView` when new items are added to it.
- `VerticalScrollBarVisibility`, of type `ScrollBarVisibility`, which specifies when the vertical scroll bar is visible.

All of these properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings.

`CarouselView` also defines two `ScrollTo` methods, that scroll items into view. One of the overloads scrolls the item at the specified index into view, while the other scrolls the specified item into view. Both overloads have additional arguments that can be specified to indicate the exact position of the item after the scroll has completed, and whether to animate the scroll.

`CarouselView` defines a `ScrollToRequested` event that is fired when one of the `ScrollTo` methods is invoked. The `ScrollToRequestedEventArgs` object that accompanies the `ScrollToRequested` event has many properties, including `IsAnimated`, `Index`, `Item`, and `ScrollToPosition`. These properties are set from the arguments specified in the `ScrollTo` method calls.

In addition, `CarouselView` defines a `Scrolled` event that is fired to indicate that scrolling occurred. The `ItemsViewScrolledEventArgs` object that accompanies the `Scrolled` event has many properties. For more information, see [Detect scrolling](#).

When a user swipes to initiate a scroll, the end position of the scroll can be controlled so that items are fully displayed. This feature is known as snapping, because items snap to position when scrolling stops. For more information, see [Snap points](#).

`CarouselView` can also load data incrementally as the user scrolls. For more information, see [Load data incrementally](#).

Detect scrolling

The `IsDragging` property can be examined to determine whether the `CarouselView` is currently scrolling through items.

In addition, `CarouselView` defines a `Scrolled` event which is fired to indicate that scrolling occurred. This event should be consumed when data about the scroll is required.

The following XAML example shows a `CarouselView` that sets an event handler for the `Scrolled` event:

```
<CarouselView Scrolled="OnCollectionViewScrolled">
    ...
</CarouselView>
```

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView();
carouselView.Scrolled += OnCarouselViewScrolled;
```

In this code example, the `OnCarouselViewScrolled` event handler is executed when the `Scrolled` event fires:

```
void OnCarouselViewScrolled(object sender, ItemsViewScrolledEventArgs e)
{
    Debug.WriteLine("HorizontalDelta: " + e.HorizontalDelta);
    Debug.WriteLine("VerticalDelta: " + e.VerticalDelta);
    Debug.WriteLine("HorizontalOffset: " + e.HorizontalOffset);
    Debug.WriteLine("VerticalOffset: " + e.VerticalOffset);
    Debug.WriteLine("FirstVisibleItemIndex: " + e.FirstVisibleItemIndex);
    Debug.WriteLine("CenterItemIndex: " + e.CenterItemIndex);
    Debug.WriteLine("LastVisibleItemIndex: " + e.LastVisibleItemIndex);
}
```

In this example, the `OnCarouselViewScrolled` event handler outputs the values of the `ItemsViewScrolledEventArgs` object that accompanies the event.

IMPORTANT

The `Scrolled` event is fired for user initiated scrolls, and for programmatic scrolls.

Scroll an item at an index into view

The first `ScrollTo` method overload scrolls the item at the specified index into view. Given a `CarouselView` object named `carouselView`, the following example shows how to scroll the item at index 6 into view:

```
carouselView.ScrollTo(6);
```

NOTE

The `ScrollToRequested` event is fired when the `ScrollTo` method is invoked.

Scroll an item into view

The second `ScrollTo` method overload scrolls the specified item into view. Given a `CarouselView` object named `carouselView`, the following example shows how to scroll the Proboscis Monkey item into view:

```
MonkeysViewModel viewModel = BindingContext as MonkeysViewModel;
Monkey monkey = viewModel.Monkeys.FirstOrDefault(m => m.Name == "Proboscis Monkey");
carouselView.ScrollTo(monkey);
```

NOTE

The `ScrollToRequested` event is fired when the `ScrollTo` method is invoked.

Disable scroll animation

A scrolling animation is displayed when moving between items in a `CarouselView`. This animation occurs both for user initiated scrolls, and for programmatic scrolls. Setting the `IsScrollAnimated` property to `false` will disable the animation for both scrolling categories.

Alternatively, the `animate` argument of the `ScrollTo` method can be set to `false` to disable the scrolling animation on programmatic scrolls:

```
carouselView.ScrollTo(monkey, animate: false);
```

Control scroll position

When scrolling an item into view, the exact position of the item after the scroll has completed can be specified with the `position` argument of the `ScrollTo` methods. This argument accepts a `ScrollToPosition` enumeration member.

MakeVisible

The `ScrollToPosition.MakeVisible` member indicates that the item should be scrolled until it's visible in the view:

```
carouselView.ScrollTo(monkey, position: ScrollToPosition.MakeVisible);
```

This example code results in the minimal scrolling required to scroll the item into view.

NOTE

The `ScrollToPosition.MakeVisible` member is used by default, if the `position` argument is not specified when calling the `ScrollTo` method.

Start

The `ScrollToPosition.Start` member indicates that the item should be scrolled to the start of the view:

```
carouselView.ScrollTo(monkey, position: ScrollToPosition.Start);
```

This example code results in the item being scrolled to the start of the view.

Center

The `ScrollToPosition.Center` member indicates that the item should be scrolled to the center of the view:

```
carouselViewView.ScrollTo(monkey, position: ScrollToPosition.Center);
```

This example code results in the item being scrolled to the center of the view.

End

The `ScrollToPosition.End` member indicates that the item should be scrolled to the end of the view:

```
carouselViewView.ScrollTo(monkey, position: ScrollToPosition.End);
```

This example code results in the item being scrolled to the end of the view.

Control scroll position when new items are added

`CarouselView` defines a `ItemsUpdatingScrollMode` property, which is backed by a bindable property. This property gets or sets a `ItemsUpdatingScrollMode` enumeration value that represents the scrolling behavior of the `CarouselView` when new items are added to it. The `ItemsUpdatingScrollMode` enumeration defines the following members:

- `KeepItemsInView` keeps the first item in the list displayed when new items are added.
- `KeepScrollOffset` ensures that the current scroll position is maintained when new items are added.
- `KeepLastItemInView` adjusts the scroll offset to keep the last item in the list displayed when new items are added.

The default value of the `ItemsUpdatingScrollMode` property is `KeepItemsInView`. Therefore, when new items are added to a `CarouselView` the first item in the list will remain displayed. To ensure that the last item in the list is displayed when new items are added, set the `ItemsUpdatingScrollMode` property to `KeepLastItemInView`:

```
<CarouselView ItemsUpdatingScrollMode="KeepLastItemInView">
    ...
</CarouselView>
```

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView
{
    ItemsUpdatingScrollMode = ItemsUpdatingScrollMode.KeepLastItemInView
};
```

Scroll bar visibility

`CarouselView` defines `HorizontalScrollBarVisibility` and `VerticalScrollBarVisibility` properties, which are backed by bindable properties. These properties get or set a `ScrollBarVisibility` enumeration value that represents when the horizontal, or vertical, scroll bar is visible. The `ScrollBarVisibility` enumeration defines the following members:

- `Default` indicates the default scroll bar behavior for the platform, and is the default value for the `HorizontalScrollBarVisibility` and `VerticalScrollBarVisibility` properties.
- `Always` indicates that scroll bars will be visible, even when the content fits in the view.
- `Never` indicates that scroll bars will not be visible, even if the content doesn't fit in the view.

Snap points

When a user swipes to initiate a scroll, the end position of the scroll can be controlled so that items are fully displayed. This feature is known as snapping, because items snap to position when scrolling stops, and is controlled by the following properties from the `ItemsLayout` class:

- `SnapPointsType`, of type `SnapPointsType`, specifies the behavior of snap points when scrolling.
- `SnapPointsAlignment`, of type `SnapPointsAlignment`, specifies how snap points are aligned with items.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

NOTE

When snapping occurs, it will occur in the direction that produces the least amount of motion.

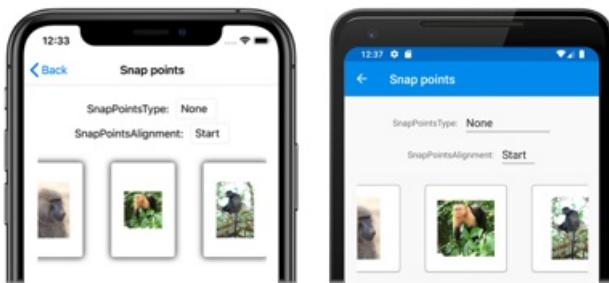
Snap points type

The `SnapPointsType` enumeration defines the following members:

- `None` indicates that scrolling does not snap to items.
- `Mandatory` indicates that content always snaps to the closest snap point to where scrolling would naturally stop, along the direction of inertia.
- `MandatorySingle` indicates the same behavior as `Mandatory`, but only scrolls one item at a time.

By default on a `CarouselView`, the `SnapPointsType` property is set to `SnapPointsType.MandatorySingle`, which ensures that scrolling only scrolls one item at a time.

The following screenshots show a `CarouselView` with snapping turned off:



Snap points alignment

The `SnapPointsAlignment` enumeration defines `Start`, `Center`, and `End` members.

IMPORTANT

The value of the `SnapPointsAlignment` property is only respected when the `SnapPointsType` property is set to `Mandatory`, or `MandatorySingle`.

Start

The `SnapPointsAlignment.Start` member indicates that snap points are aligned with the leading edge of items.

The following XAML example shows how to set this enumeration member:

```
<CarouselView ItemsSource="{Binding Monkeys}"
    PeekAreaInsets="100">
    <CarouselView.ItemsLayout>
        <LinearItemsLayout Orientation="Horizontal"
            SnapPointsType="MandatorySingle"
            SnapPointsAlignment="Start" />
    </CarouselView.ItemsLayout>
    ...
</CarouselView>
```

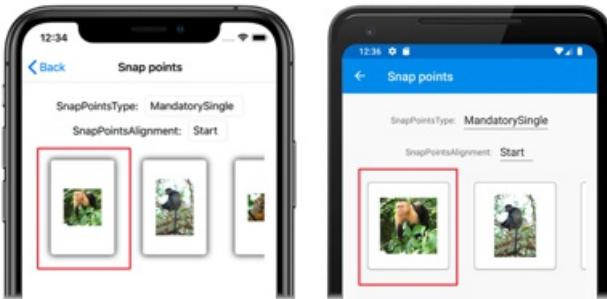
The equivalent C# code is:

```

CarouselView carouselView = new CarouselView
{
    ItemsLayout = new LinearItemsLayout(ItemsLayoutOrientation.Horizontal)
    {
        SnapPointsType = SnapPointsType.MandatorySingle,
        SnapPointsAlignment = SnapPointsAlignment.Start
    },
    // ...
};

```

When a user swipes to initiate a scroll in a horizontally scrolling `CarouselView`, the left item will be aligned with the left of the view:



Center

The `SnapPointsAlignment.Center` member indicates that snap points are aligned with the center of items.

By default on a `CarouselView`, the `SnapPointsAlignment` property is set to `Center`. However, for completeness, the following XAML example shows how to set this enumeration member:

```

<CarouselView ItemsSource="{Binding Monkeys}"
              PeekAreaInsets="100">
    <CarouselView.ItemsLayout>
        <LinearItemsLayout Orientation="Horizontal"
                           SnapPointsType="MandatorySingle"
                           SnapPointsAlignment="Center" />
    </CarouselView.ItemsLayout>
    ...
</CarouselView>

```

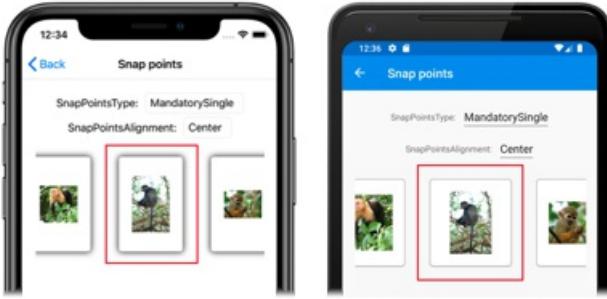
The equivalent C# code is:

```

CarouselView carouselView = new CarouselView
{
    ItemsLayout = new LinearItemsLayout(ItemsLayoutOrientation.Horizontal)
    {
        SnapPointsType = SnapPointsType.MandatorySingle,
        SnapPointsAlignment = SnapPointsAlignment.Center
    },
    // ...
};

```

When a user swipes to initiate a scroll in a horizontally scrolling `CarouselView`, the center item will be aligned with the center of the view:



End

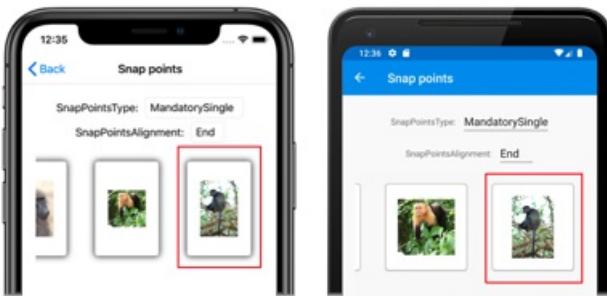
The `SnapPointsAlignment.End` member indicates that snap points are aligned with the trailing edge of items. The following XAML example shows how to set this enumeration member:

```
<CarouselView ItemsSource="{Binding Monkeys}"
    PeekAreaInsets="100">
    <CarouselView.ItemsLayout>
        <LinearItemsLayout Orientation="Horizontal"
            SnapPointsType="MandatorySingle"
            SnapPointsAlignment="End" />
    </CarouselView.ItemsLayout>
    ...
</CarouselView>
```

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView
{
    ItemsLayout = new LinearItemsLayout(ItemsLayoutOrientation.Horizontal)
    {
        SnapPointsType = SnapPointsType.MandatorySingle,
        SnapPointsAlignment = SnapPointsAlignment.End
    },
    // ...
};
```

When a user swipes to initiate a scroll in a horizontally scrolling `CarouselView`, the right item will be aligned with the right of the view.



Related links

- [CarouselView \(sample\)](#)

Xamarin.Forms CollectionView

8/4/2022 • 2 minutes to read • [Edit Online](#)

Introduction

The `CollectionView` is a flexible and performant view for presenting lists of data using different layout specifications.

Data

A `CollectionView` is populated with data by setting its `ItemsSource` property to any collection that implements `IEnumerable`. The appearance of each item in the list can be defined by setting the `ItemTemplate` property to a `DataTemplate`.

Layout

By default, a `CollectionView` will display its items in a vertical list. However, vertical and horizontal lists and grids can be specified.

Selection

By default, `CollectionView` selection is disabled. However, single and multiple selection can be enabled.

Empty views

In `CollectionView`, an empty view can be specified that provides feedback to the user when no data is available for display. The empty view can be a string, a view, or multiple views.

Scrolling

When a user swipes to initiate a scroll, the end position of the scroll can be controlled so that items are fully displayed. In addition, `CollectionView` defines two `ScrollTo` methods, that programmatically scroll items into view. One of the overloads scrolls the item at the specified index into view, while the other scrolls the specified item into view.

Grouping

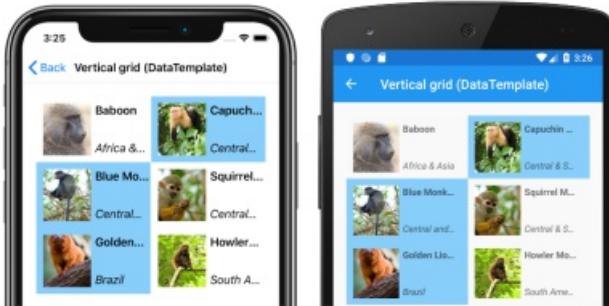
`CollectionView` can display correctly grouped data by setting its `IsGrouped` property to `true`.

Xamarin.Forms CollectionView Introduction

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

`CollectionView` is a view for presenting lists of data using different layout specifications. It aims to provide a more flexible, and performant alternative to `ListView`. For example, the following screenshots show a `CollectionView` that uses a two column vertical grid, and which allows multiple selection:



`CollectionView` should be used for presenting lists of data that require scrolling or selection. A bindable layout can be used when the data to be displayed doesn't require scrolling or selection. For more information, see [Bindable Layouts in Xamarin.Forms](#).

`CollectionView` is available from Xamarin.Forms 4.3.

IMPORTANT

`CollectionView` is available on iOS and Android, but is only [partially available](#) on the Universal Windows Platform.

CollectionView and ListView differences

While the `CollectionView` and `ListView` APIs are similar, there are some notable differences:

- `CollectionView` has a flexible layout model, which allows data to be presented vertically or horizontally, in a list or a grid.
- `CollectionView` supports single and multiple selection.
- `CollectionView` has no concept of cells. Instead, a data template is used to define the appearance of each item of data in the list.
- `CollectionView` automatically utilizes the virtualization provided by the underlying native controls.
- `CollectionView` reduces the API surface of `ListView`. Many properties and events from `ListView` are not present in `CollectionView`.
- `CollectionView` does not include built-in separators.
- `CollectionView` will throw an exception if its `ItemsSource` is updated off the UI thread.

Move from ListView to CollectionView

`ListView` implementations in existing Xamarin.Forms applications can be migrated to `CollectionView` implementations with the help of the following table:

CONCEPT	LISTVIEW API	COLLECTIONVIEW
Data	<code>ItemsSource</code>	A <code>CollectionView</code> is populated with data by setting its <code>ItemsSource</code> property. For more information, see Populate a CollectionView with data .
Item appearance	<code>ItemTemplate</code>	The appearance of each item in a <code>CollectionView</code> can be defined by setting the <code>ItemTemplate</code> property to a <code>DataTemplate</code> . For more information, see Define item appearance .
Cells	<code>TextCell</code> , <code>ImageCell</code> , <code>ViewCell</code>	<code>CollectionView</code> has no concept of cells, and therefore no concept of disclosure indicators. Instead, a data template is used to define the appearance of each item of data in the list.
Row separators	<code>SeparatorColor</code> , <code>SeparatorVisibility</code>	<code>CollectionView</code> does not include built-in separators. These can be provided, if desired, in the item template.
Selection	<code>SelectionMode</code> , <code>SelectedItem</code>	<code>CollectionView</code> supports single and multiple selection. For more information, see Xamarin.Forms CollectionView Selection .
Row height	<code>HasUnevenRows</code> , <code>RowHeight</code>	In a <code>CollectionView</code> , the row height of each item is determined by the <code>ItemSizingStrategy</code> property. For more information, see Item sizing .
Caching	<code>CachingStrategy</code>	<code>CollectionView</code> automatically uses the virtualization provided by the underlying native controls.
Headers and footers	<code>Header</code> , <code>HeaderElement</code> , <code>HeaderTemplate</code> , <code>Footer</code> , <code>FooterElement</code> , <code>FooterTemplate</code>	<code>CollectionView</code> can present a header and footer that scroll with the items in the list, via the <code>Header</code> , <code>Footer</code> , <code>HeaderTemplate</code> , and <code>FooterTemplate</code> properties. For more information, see Headers and footers .
Grouping	<code>GroupDisplayBinding</code> , <code>GroupHeaderTemplate</code> , <code>GroupShortNameBinding</code> , <code>IsGroupingEnabled</code>	<code>CollectionView</code> displays correctly grouped data by setting its <code>IsGrouped</code> property to <code>true</code> . Group headers and group footers can be customized by setting the <code>GroupHeaderTemplate</code> and <code>GroupFooterTemplate</code> properties to <code>DataTemplate</code> objects. For more information, see Xamarin.Forms CollectionView Grouping .

CONCEPT	LISTVIEW API	COLLECTIONVIEW
Pull to refresh	<code>IsPullToRefreshEnabled</code> , <code>IsRefreshing</code> , <code>RefreshAllowed</code> , <code>RefreshCommand</code> , <code>RefreshControlColor</code> , <code>BeginRefresh()</code> , <code>EndRefresh()</code>	Pull to refresh functionality is supported by setting a <code>CollectionView</code> as the child of a <code>RefreshView</code> . For more information, see Pull to refresh .
Context menu items	<code>ContextActions</code>	Context menu items are supported by setting a <code>SwipeView</code> as the root view in the <code>DataTemplate</code> that defines the appearance of each item of data in the <code>CollectionView</code> . For more information, see Context menus .
Scrolling	<code>ScrollTo()</code>	<code>CollectionView</code> defines <code>ScrollTo</code> methods, that scroll items into view. For more information, see Scrolling .

Related links

- [CollectionView \(sample\)](#)
- [Bindable Layouts in Xamarin.Forms](#)

Xamarin.Forms CollectionView Data

8/4/2022 • 8 minutes to read • [Edit Online](#)



[Download the sample](#)

`CollectionView` includes the following properties that define the data to be displayed, and its appearance:

- `ItemsSource`, of type `IEnumerable`, specifies the collection of items to be displayed, and has a default value of `null`.
- `ItemTemplate`, of type `DataTemplate`, specifies the template to apply to each item in the collection of items to be displayed.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

NOTE

`CollectionView` defines a `ItemsUpdatingScrollMode` property that represents the scrolling behavior of the `CollectionView` when new items are added to it. For more information about this property, see [Control scroll position when new items are added](#).

`CollectionView` supports incremental data virtualization as the user scrolls. For more information, see [Load data incrementally](#).

Populate a CollectionView with data

A `CollectionView` is populated with data by setting its `ItemsSource` property to any collection that implements `IEnumerable`. By default, `CollectionView` displays items in a vertical list.

IMPORTANT

If the `CollectionView` is required to refresh as items are added, removed, or changed in the underlying collection, the underlying collection should be an `IEnumerable` collection that sends property change notifications, such as `ObservableCollection`.

`CollectionView` can be populated with data by using data binding to bind its `ItemsSource` property to an `IEnumerable` collection. In XAML, this is achieved with the `Binding` markup extension:

```
<CollectionView ItemsSource="{Binding Monkeys}" />
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView();
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
```

In this example, the `ItemsSource` property data binds to the `Monkeys` property of the connected.viewmodel.

NOTE

Compiled bindings can be enabled to improve data binding performance in Xamarin.Forms applications. For more information, see [Compiled Bindings](#).

For information on how to change the `CollectionView` layout, see [Xamarin.Forms CollectionView Layout](#). For information on how to define the appearance of each item in the `CollectionView`, see [Define item appearance](#). For more information about data binding, see [Xamarin.Forms Data Binding](#).

WARNING

`CollectionView` will throw an exception if its `ItemsSource` is updated off the UI thread.

Define item appearance

The appearance of each item in the `CollectionView` can be defined by setting the `CollectionView.ItemTemplate` property to a `DataTemplate`:

```
<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Grid Padding="10">
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="Auto" />
                </Grid.ColumnDefinitions>
                <Image Grid.RowSpan="2"
                      Source="{Binding ImageUrl}"
                      Aspect="AspectFill"
                      HeightRequest="60"
                      WidthRequest="60" />
                <Label Grid.Column="1"
                      Text="{Binding Name}"
                      FontAttributes="Bold" />
                <Label Grid.Row="1"
                      Grid.Column="1"
                      Text="{Binding Location}"
                      FontAttributes="Italic"
                      VerticalOptions="End" />
            </Grid>
        </DataTemplate>
    </CollectionView.ItemTemplate>
    ...
</CollectionView>
```

The equivalent C# code is:

```

CollectionView collectionView = new CollectionView();
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");

collectionView.ItemTemplate = new DataTemplate(() =>
{
    Grid grid = new Grid { Padding = 10 };
    grid.RowDefinitions.Add(new RowDefinition { Height = GridLength.Auto });
    grid.RowDefinitions.Add(new RowDefinition { Height = GridLength.Auto });
    grid.ColumnDefinitions.Add(new ColumnDefinition { Width = GridLength.Auto });
    grid.ColumnDefinitions.Add(new ColumnDefinition { Width = GridLength.Auto });

    Image image = new Image { Aspect = Aspect.AspectFill, HeightRequest = 60, WidthRequest = 60 };
    image.SetBinding(Image.SourceProperty, "ImageUrl");

    Label nameLabel = new Label { FontAttributes = FontAttributes.Bold };
    nameLabel.SetBinding(Label.TextProperty, "Name");

    Label locationLabel = new Label { FontAttributes = FontAttributes.Italic, VerticalOptions =
LayoutOptions.End };
    locationLabel.SetBinding(Label.TextProperty, "Location");

    Grid.SetRowSpan(image, 2);

    grid.Children.Add(image);
    grid.Children.Add(nameLabel, 1, 0);
    grid.Children.Add(locationLabel, 1, 1);

    return grid;
});

```

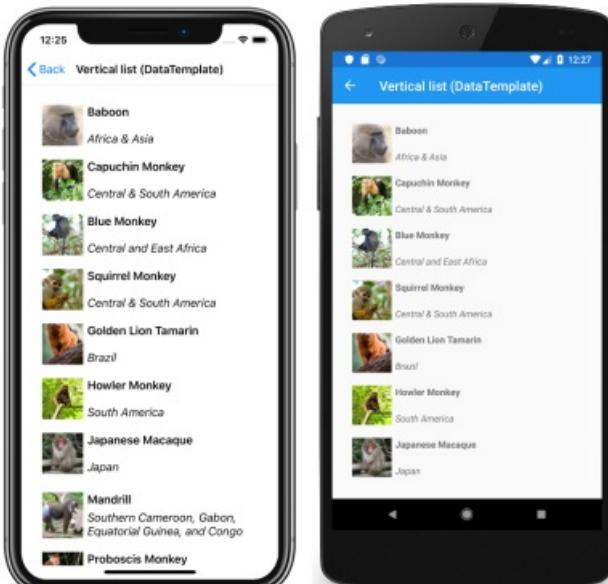
The elements specified in the `DataTemplate` define the appearance of each item in the list. In the example, layout within the `DataTemplate` is managed by a `Grid`. The `Grid` contains an `Image` object, and two `Label` objects, that all bind to properties of the `Monkey` class:

```

public class Monkey
{
    public string Name { get; set; }
    public string Location { get; set; }
    public string Details { get; set; }
    public string ImageUrl { get; set; }
}

```

The following screenshots show the result of templating each item in the list:



For more information about data templates, see [Xamarin.Forms Data Templates](#).

Choose item appearance at runtime

The appearance of each item in the `CollectionView` can be chosen at runtime, based on the item value, by setting the `CollectionView.ItemTemplate` property to a `DataTemplateSelector` object:

```
<ContentPage ...>
    xmlns:controls="clr-namespace:CollectionViewDemos.Controls">
    <ContentPage.Resources>
        <DataTemplate x:Key="AmericanMonkeyTemplate">
            ...
        </DataTemplate>

        <DataTemplate x:Key="OtherMonkeyTemplate">
            ...
        </DataTemplate>

        <controls:MonkeyDataTemplateSelector x:Key="MonkeySelector"
            AmericanMonkey="{StaticResource AmericanMonkeyTemplate}"
            OtherMonkey="{StaticResource OtherMonkeyTemplate}" />
    </ContentPage.Resources>

    <CollectionView ItemsSource="{Binding Monkeys}"
        ItemTemplate="{StaticResource MonkeySelector}" />
</ContentPage>
```

The equivalent C# code is:

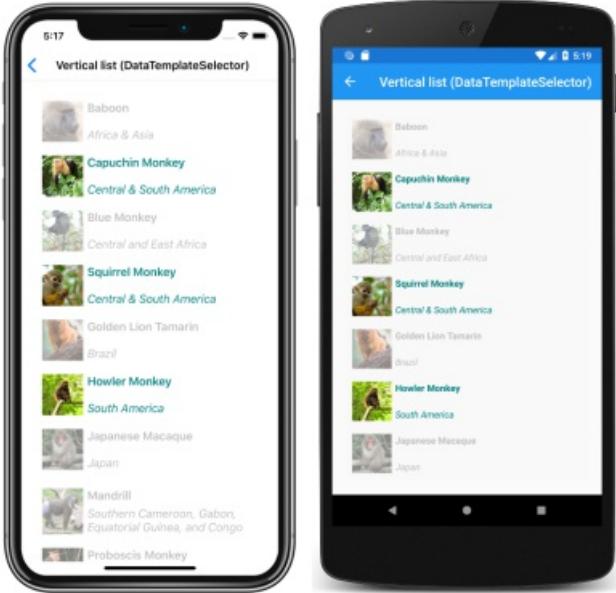
```
CollectionView collectionView = new CollectionView
{
    ItemTemplate = new MonkeyDataTemplateSelector { ... }
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
```

The `ItemTemplate` property is set to a `MonkeyDataTemplateSelector` object. The following example shows the `MonkeyDataTemplateSelector` class:

```
public class MonkeyDataTemplateSelector : DataTemplateSelector
{
    public DataTemplate AmericanMonkey { get; set; }
    public DataTemplate OtherMonkey { get; set; }

    protected override DataTemplate OnSelectTemplate(object item, BindableObject container)
    {
        return ((Monkey)item).Location.Contains("America") ? AmericanMonkey : OtherMonkey;
    }
}
```

The `MonkeyDataTemplateSelector` class defines `AmericanMonkey` and `OtherMonkey` `DataTemplate` properties that are set to different data templates. The `OnSelectTemplate` override returns the `AmericanMonkey` template, which displays the monkey name and location in teal, when the monkey name contains "America". When the monkey name doesn't contain "America", the `OnSelectTemplate` override returns the `OtherMonkey` template, which displays the monkey name and location in silver:



For more information about data template selectors, see [Create a Xamarin.Forms DataTemplateSelector](#).

IMPORTANT

When using `CollectionView`, never set the root element of your `DataTemplate` objects to a `ViewCell`. This will result in an exception being thrown because `CollectionView` has no concept of cells.

Context menus

`CollectionView` supports context menus for items of data through the `SwipeView`, which reveals the context menu with a swipe gesture. The `SwipeView` is a container control that wraps around an item of content, and provides context menu items for that item of content. Therefore, context menus are implemented for a `CollectionView` by creating a `SwipeView` that defines the content that the `SwipeView` wraps around, and the context menu items that are revealed by the swipe gesture. This is achieved by setting the `SwipeView` as the root view in the `DataTemplate` that defines the appearance of each item of data in the `CollectionView`:

```
<CollectionView x:Name="collectionView"
    ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <SwipeView>
                <SwipeView.LeftItems>
                    <SwipeItems>
                        <SwipeItem Text="Favorite"
                            IconImageSource="favorite.png"
                            BackgroundColor="LightGreen"
                            Command="{Binding Source={x:Reference collectionView},
Path=BindingContext.FavoriteCommand}"
                            CommandParameter="{Binding}" />
                        <SwipeItem Text="Delete"
                            IconImageSource="delete.png"
                            BackgroundColor="LightPink"
                            Command="{Binding Source={x:Reference collectionView},
Path=BindingContext.DeleteCommand}"
                            CommandParameter="{Binding}" />
                    </SwipeItems>
                </SwipeView.LeftItems>
                <Grid BackgroundColor="White"
                    Padding="10">
                    <!-- Define item appearance -->
                </Grid>
            </SwipeView>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

The equivalent C# code is:

```

CollectionView collectionView = new CollectionView();
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");

collectionView.ItemTemplate = new DataTemplate(() =>
{
    // Define item appearance
    Grid grid = new Grid { Padding = 10, BackgroundColor = Color.White };
    // ...

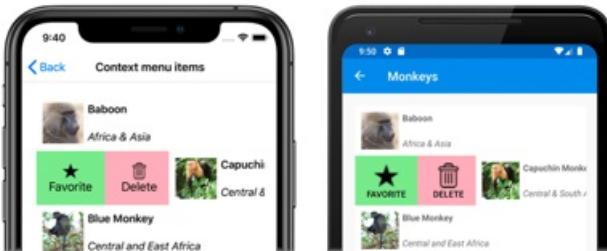
    SwipeView swipeView = new SwipeView();
    SwipeItem favoriteSwipeItem = new SwipeItem
    {
        Text = "Favorite",
        IconImageSource = "favorite.png",
        BackgroundColor = Color.LightGreen
    };
    favoriteSwipeItem.SetBinding(MenuItem.CommandProperty, new Binding("BindingContext.FavoriteCommand", source: collectionView));
    favoriteSwipeItem.SetBinding(MenuItem.CommandParameterProperty, ".");

    SwipeItem deleteSwipeItem = new SwipeItem
    {
        Text = "Delete",
        IconImageSource = "delete.png",
        BackgroundColor = Color.LightPink
    };
    deleteSwipeItem.SetBinding(MenuItem.CommandProperty, new Binding("BindingContext.DeleteCommand", source: collectionView));
    deleteSwipeItem.SetBinding(MenuItem.CommandParameterProperty, ".");

    swipeView.LeftItems = new SwipeItems { favoriteSwipeItem, deleteSwipeItem };
    swipeView.Content = grid;
    return swipeView;
});

```

In this example, the `SwipeView` content is a `Grid` that defines the appearance of each item in the `CollectionView`. The swipe items are used to perform actions on the `SwipeView` content, and are revealed when the control is swiped from the left side:



`SwipeView` supports four different swipe directions, with the swipe direction being defined by the directional `SwipeItems` collection the `SwipeItems` objects are added to. By default, a swipe item is executed when it's tapped by the user. In addition, once a swipe item has been executed the swipe items are hidden and the `SwipeView` content is re-displayed. However, these behaviors can be changed.

For more information about the `SwipeView` control, see [Xamarin.Forms SwipeView](#).

Pull to refresh

`CollectionView` supports pull to refresh functionality through the `RefreshView`, which enables the data being displayed to be refreshed by pulling down on the list of items. The `RefreshView` is a container control that provides pull to refresh functionality to its child, provided that the child supports scrollable content. Therefore, pull to refresh is implemented for a `CollectionView` by setting it as the child of a `RefreshView`:

```
<RefreshView IsRefreshing="{Binding IsRefreshing}">
    Command="{Binding RefreshCommand}">
    <CollectionView ItemsSource="{Binding Animals}">
        ...
    </CollectionView>
</RefreshView>
```

The equivalent C# code is:

```
RefreshView refreshView = new RefreshView();
 ICommand refreshCommand = new Command(() =>
{
    // IsRefreshing is true
    // Refresh data here
    refreshView.IsRefreshing = false;
});
refreshView.Command = refreshCommand;

CollectionView collectionView = new CollectionView();
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Animals");
refreshView.Content = collectionView;
// ...
```

When the user initiates a refresh, the `ICommand` defined by the `Command` property is executed, which should refresh the items being displayed. A refresh visualization is shown while the refresh occurs, which consists of an animated progress circle:



The value of the `RefreshView.IsRefreshing` property indicates the current state of the `RefreshView`. When a refresh is triggered by the user, this property will automatically transition to `true`. Once the refresh completes, you should reset the property to `false`.

For more information about `RefreshView`, see [Xamarin.Forms RefreshView](#).

Load data incrementally

`CollectionView` supports incremental data virtualization as the user scrolls. This enables scenarios such as asynchronously loading a page of data from a web service, as the user scrolls. In addition, the point at which more data is loaded is configurable so that users don't see blank space, or are stopped from scrolling.

`CollectionView` defines the following properties to control incremental loading of data:

- `RemainingItemsThreshold`, of type `int`, the threshold of items not yet visible in the list at which the `RemainingItemsThresholdReached` event will be fired.
- `RemainingItemsThresholdReachedCommand`, of type `ICommand`, which is executed when the `RemainingItemsThreshold` is reached.
- `RemainingItemsThresholdReachedCommandParameter`, of type `object`, which is the parameter that's passed to the `RemainingItemsThresholdReachedCommand`.

`CollectionView` also defines a `RemainingItemsThresholdReached` event that is fired when the `CollectionView` is scrolled far enough that `RemainingItemsThreshold` items have not been displayed. This event can be handled to

load more items. In addition, when the `RemainingItemsThresholdReached` event is fired, the `RemainingItemsThresholdReachedCommand` is executed, enabling incremental data loading to take place in a viewmodel.

The default value of the `RemainingItemsThreshold` property is -1, which indicates that the `RemainingItemsThresholdReached` event will never be fired. When the property value is 0, the `RemainingItemsThresholdReached` event will be fired when the final item in the `ItemsSource` is displayed. For values greater than 0, the `RemainingItemsThresholdReached` event will be fired when the `ItemsSource` contains that number of items not yet scrolled to.

NOTE

`CollectionView` validates the `RemainingItemsThreshold` property so that its value is always greater than or equal to -1.

The following XAML example shows a `CollectionView` that loads data incrementally:

```
<CollectionView ItemsSource="{Binding Animals}"
               RemainingItemsThreshold="5"
               RemainingItemsThresholdReached="OnCollectionViewRemainingItemsThresholdReached">
    ...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    RemainingItemsThreshold = 5
};
collectionView.RemainingItemsThresholdReached += OnCollectionViewRemainingItemsThresholdReached;
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Animals");
```

In this code example, the `RemainingItemsThresholdReached` event fires when there are 5 items not yet scrolled to, and in response executes the `OnCollectionViewRemainingItemsThresholdReached` event handler:

```
void OnCollectionViewRemainingItemsThresholdReached(object sender, EventArgs e)
{
    // Retrieve more data here and add it to the CollectionView's ItemsSource collection.
}
```

NOTE

Data can also be loaded incrementally by binding the `RemainingItemsThresholdReachedCommand` to an `ICommand` implementation in the viewmodel.

Related links

- [CollectionView \(sample\)](#)
- [Xamarin.Forms RefreshView](#)
- [Xamarin.Forms SwipeView](#)
- [Xamarin.Forms Data Binding](#)
- [Xamarin.Forms Data Templates](#)

- Create a Xamarin.Forms DataTemplateSelector

Xamarin.Forms CollectionView Layout

8/4/2022 • 11 minutes to read • [Edit Online](#)



[Download the sample](#)

`CollectionView` defines the following properties that control layout:

- `ItemsLayout`, of type `IItemsLayout`, specifies the layout to be used.
- `ItemSizingStrategy`, of type `ItemSizingStrategy`, specifies the item measure strategy to be used.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

By default, a `CollectionView` will display its items in a vertical list. However, any of the following layouts can be used:

- Vertical list – a single column list that grows vertically as new items are added.
- Horizontal list – a single row list that grows horizontally as new items are added.
- Vertical grid – a multi-column grid that grows vertically as new items are added.
- Horizontal grid – a multi-row grid that grows horizontally as new items are added.

These layouts can be specified by setting the `ItemsLayout` property to class that derives from the `ItemsLayout` class. This class defines the following properties:

- `Orientation`, of type `ItemsLayoutOrientation`, specifies the direction in which the `CollectionView` expands as items are added.
- `SnapPointsAlignment`, of type `SnapPointsAlignment`, specifies how snap points are aligned with items.
- `SnapPointsType`, of type `SnapPointsType`, specifies the behavior of snap points when scrolling.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings. For more information about snap points, see [Snap points](#) in the [Xamarin.Forms CollectionView Scrolling](#) guide.

The `ItemsLayoutOrientation` enumeration defines the following members:

- `Vertical` indicates that the `CollectionView` will expand vertically as items are added.
- `Horizontal` indicates that the `CollectionView` will expand horizontally as items are added.

The `LinearItemsLayout` class inherits from the `ItemsLayout` class, and defines an `ItemSpacing` property, of type `double`, that represents the empty space around each item. The default value of this property is 0, and its value must always be greater than or equal to 0. The `LinearItemsLayout` class also defines static `Vertical` and `Horizontal` members. These members can be used to create vertical or horizontal lists, respectively.

Alternatively, a `LinearItemsLayout` object can be created, specifying an `ItemsLayoutOrientation` enumeration member as an argument.

The `GridItemsLayout` class inherits from the `ItemsLayout` class, and defines the following properties:

- `VerticalItemSpacing`, of type `double`, that represents the vertical empty space around each item. The default value of this property is 0, and its value must always be greater than or equal to 0.
- `HorizontalItemSpacing`, of type `double`, that represents the horizontal empty space around each item. The default value of this property is 0, and its value must always be greater than or equal to 0.
- `Span`, of type `int`, that represents the number of columns or rows to display in the grid. The default value

of this property is 1, and its value must always be greater than or equal to 1.

These properties are backed by [BindableProperty](#) objects, which means that the properties can be targets of data bindings.

NOTE

[CollectionView](#) uses the native layout engines to perform layout.

Vertical list

By default, [CollectionView](#) will display its items in a vertical list layout. Therefore, it's not necessary to set the [ItemsLayout](#) property to use this layout:

```
<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Grid Padding="10">
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="Auto" />
                </Grid.ColumnDefinitions>
                <Image Grid.RowSpan="2"
                      Source="{Binding ImageUrl}"
                      Aspect="AspectFill"
                      HeightRequest="60"
                      WidthRequest="60" />
                <Label Grid.Column="1"
                      Text="{Binding Name}"
                      FontAttributes="Bold" />
                <Label Grid.Row="1"
                      Grid.Column="1"
                      Text="{Binding Location}"
                      FontAttributes="Italic"
                      VerticalOptions="End" />
            </Grid>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

However, for completeness, in XAML a [CollectionView](#) can be set to display its items in a vertical list by setting its [ItemsLayout](#) property to [VerticalList](#):

```
<CollectionView ItemsSource="{Binding Monkeys}"
               ItemsLayout="VerticalList">
    ...
</CollectionView>
```

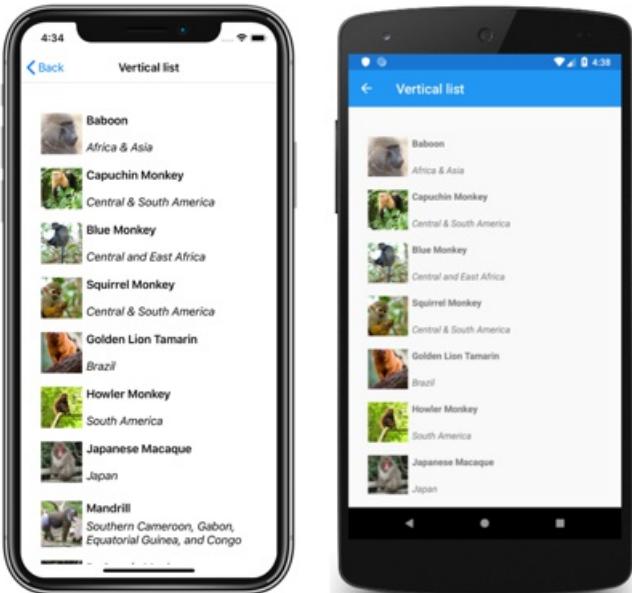
Alternatively, this can also be accomplished by setting the [ItemsLayout](#) property to a [LinearItemsLayout](#) object, specifying the [Vertical](#) [ItemsLayoutOrientation](#) enumeration member as the [Orientation](#) property value:

```
<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical" />
    </CollectionView.ItemsLayout>
    ...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    ...
    ItemsLayout = LinearItemsLayout.Vertical
};
```

This results in a single column list, which grows vertically as new items are added:



Horizontal list

In XAML, a `CollectionView` can display its items in a horizontal list by setting its `ItemsLayout` property to `HorizontalList`:

```

<CollectionView ItemsSource="{Binding Monkeys}"
    ItemsLayout="HorizontalList">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Grid Padding="10">
                <Grid.RowDefinitions>
                    <RowDefinition Height="35" />
                    <RowDefinition Height="35" />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="70" />
                    <ColumnDefinition Width="140" />
                </Grid.ColumnDefinitions>
                <Image Grid.RowSpan="2"
                    Source="{Binding ImageUrl}"
                    Aspect="AspectFill"
                    HeightRequest="60"
                    WidthRequest="60" />
                <Label Grid.Column="1"
                    Text="{Binding Name}"
                    FontAttributes="Bold"
                    LineBreakMode="TailTruncation" />
                <Label Grid.Row="1"
                    Grid.Column="1"
                    Text="{Binding Location}"
                    LineBreakMode="TailTruncation"
                    FontAttributes="Italic"
                    VerticalOptions="End" />
            </Grid>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>

```

Alternatively, this layout can also be accomplished by setting the `ItemsLayout` property to a `LinearItemsLayout` object, specifying the `Horizontal` `ItemsLayoutOrientation` enumeration member as the `Orientation` property value:

```

<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Horizontal" />
    </CollectionView.ItemsLayout>
    ...
</CollectionView>

```

The equivalent C# code is:

```

CollectionView collectionView = new CollectionView
{
    ...
    ItemsLayout = LinearItemsLayout.Horizontal
};

```

This results in a single row list, which grows horizontally as new items are added:



Vertical grid

In XAML, a `CollectionView` can display its items in a vertical grid by setting its `ItemsLayout` property to `VerticalGrid`:

```
<CollectionView ItemsSource="{Binding Monkeys}"
    ItemsLayout="VerticalGrid, 2">
<CollectionView.ItemTemplate>
    <DataTemplate>
        <Grid Padding="10">
            <Grid.RowDefinitions>
                <RowDefinition Height="35" />
                <RowDefinition Height="35" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="70" />
                <ColumnDefinition Width="80" />
            </Grid.ColumnDefinitions>
            <Image Grid.RowSpan="2"
                Source="{Binding ImageUrl}"
                Aspect="AspectFill"
                HeightRequest="60"
                WidthRequest="60" />
            <Label Grid.Column="1"
                Text="{Binding Name}"
                FontAttributes="Bold"
                LineBreakMode="TailTruncation" />
            <Label Grid.Row="1"
                Grid.Column="1"
                Text="{Binding Location}"
                LineBreakMode="TailTruncation"
                FontAttributes="Italic"
                VerticalOptions="End" />
        </Grid>
    </DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>
```

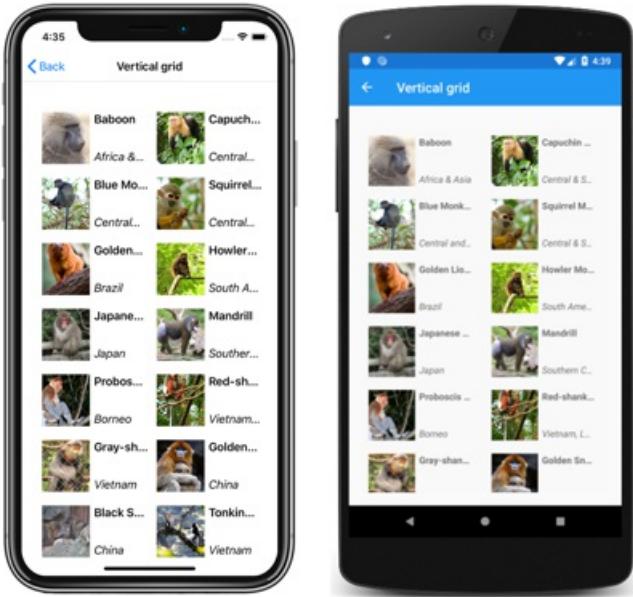
Alternatively, this layout can also be accomplished by setting the `ItemsLayout` property to a `GridItemsLayout` object whose `Orientation` property is set to `Vertical`:

```
<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemsLayout>
        <GridItemsLayout Orientation="Vertical"
            Span="2" />
    </CollectionView.ItemsLayout>
    ...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    ...
    ItemsLayout = new GridItemsLayout(2, ItemsLayoutOrientation.Vertical)
};
```

By default, a vertical `GridItemsLayout` will display items in a single column. However, this example sets the `GridItemsLayout.Span` property to 2. This results in a two-column grid, which grows vertically as new items are added:



Horizontal grid

In XAML, a `CollectionView` can display its items in a horizontal grid by setting its `ItemsLayout` property to `HorizontalGrid`:

```
<CollectionView ItemsSource="{Binding Monkeys}"
    ItemsLayout="HorizontalGrid, 4">
<CollectionView.ItemTemplate>
    <DataTemplate>
        <Grid Padding="10">
            <Grid.RowDefinitions>
                <RowDefinition Height="35" />
                <RowDefinition Height="35" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="70" />
                <ColumnDefinition Width="140" />
            </Grid.ColumnDefinitions>
            <Image Grid.RowSpan="2"
                Source="{Binding ImageUrl}"
                Aspect="AspectFill"
                HeightRequest="60"
                WidthRequest="60" />
            <Label Grid.Column="1"
                Text="{Binding Name}"
                FontAttributes="Bold"
                LineBreakMode="TailTruncation" />
            <Label Grid.Row="1"
                Grid.Column="1"
                Text="{Binding Location}"
                LineBreakMode="TailTruncation"
                FontAttributes="Italic"
                VerticalOptions="End" />
        </Grid>
    </DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>
```

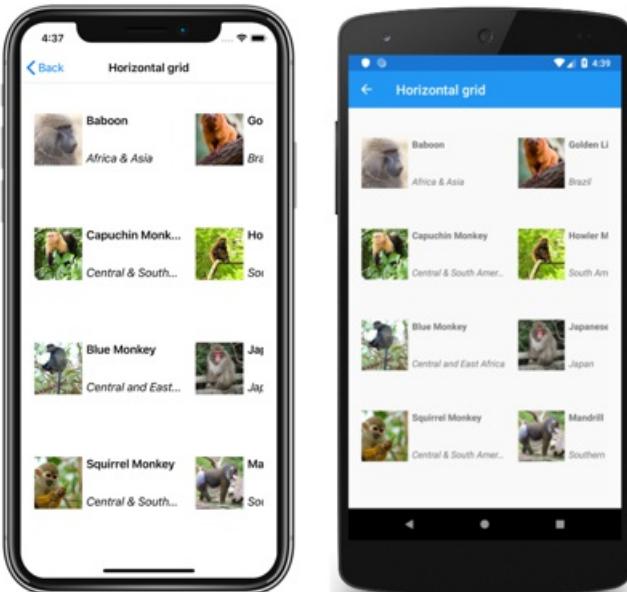
Alternatively, this layout can also be accomplished by setting the `ItemsLayout` property to a `GridItemsLayout` object whose `Orientation` property is set to `Horizontal`:

```
<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemsLayout>
        <GridItemsLayout Orientation="Horizontal"
            Span="4" />
    </CollectionView.ItemsLayout>
    ...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    ...
    ItemsLayout = new GridItemsLayout(4, ItemsLayoutOrientation.Horizontal)
};
```

By default, a horizontal `GridItemsLayout` will display items in a single row. However, this example sets the `GridItemsLayout.Span` property to 4. This results in a four-row grid, which grows horizontally as new items are added:



Headers and footers

`CollectionView` can present a header and footer that scroll with the items in the list. The header and footer can be strings, views, or `DataTemplate` objects.

`CollectionView` defines the following properties for specifying the header and footer:

- `Header`, of type `object`, specifies the string, binding, or view that will be displayed at the start of the list.
- `HeaderTemplate`, of type `DataTemplate`, specifies the `DataTemplate` to use to format the `Header`.
- `Footer`, of type `object`, specifies the string, binding, or view that will be displayed at the end of the list.
- `FooterTemplate`, of type `DataTemplate`, specifies the `DataTemplate` to use to format the `Footer`.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

When a header is added to a layout that grows horizontally, from left to right, the header is displayed to the left of the list. Similarly, when a footer is added to a layout that grows horizontally, from left to right, the footer is displayed to the right of the list.

Display strings in the header and footer

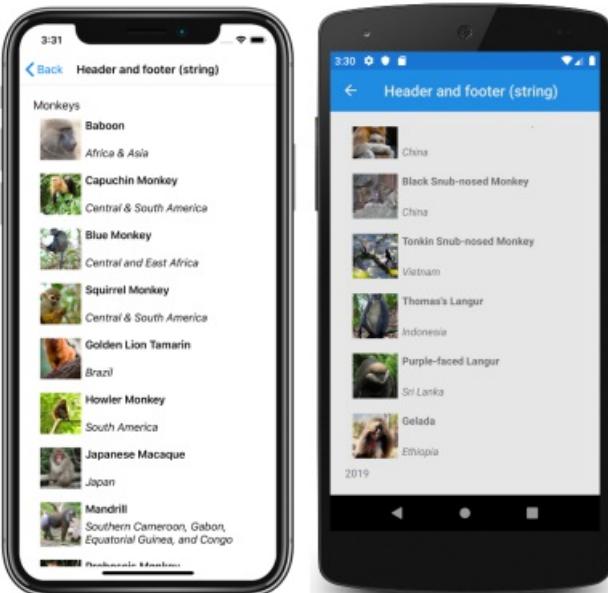
The `Header` and `Footer` properties can be set to `string` values, as shown in the following example:

```
<CollectionView ItemsSource="{Binding Monkeys}"
    Header="Monkeys"
    Footer="2019">
    ...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    Header = "Monkeys",
    Footer = "2019"
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
```

This code results in the following screenshots, with the header shown in the iOS screenshot, and the footer shown in the Android screenshot:



Display views in the header and footer

The `Header` and `Footer` properties can each be set to a view. This can be a single view, or a view that contains multiple child views. The following example shows the `Header` and `Footer` properties each set to a `StackLayout` object that contains a `Label` object:

```

<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.Header>
        <StackLayout BackgroundColor="LightGray">
            <Label Margin="10,0,0,0"
                  Text="Monkeys"
                  FontSize="Small"
                  FontAttributes="Bold" />
        </StackLayout>
    </CollectionView.Header>
    <CollectionView.Footer>
        <StackLayout BackgroundColor="LightGray">
            <Label Margin="10,0,0,0"
                  Text="Friends of Xamarin Monkey"
                  FontSize="Small"
                  FontAttributes="Bold" />
        </StackLayout>
    </CollectionView.Footer>
    ...
</CollectionView>

```

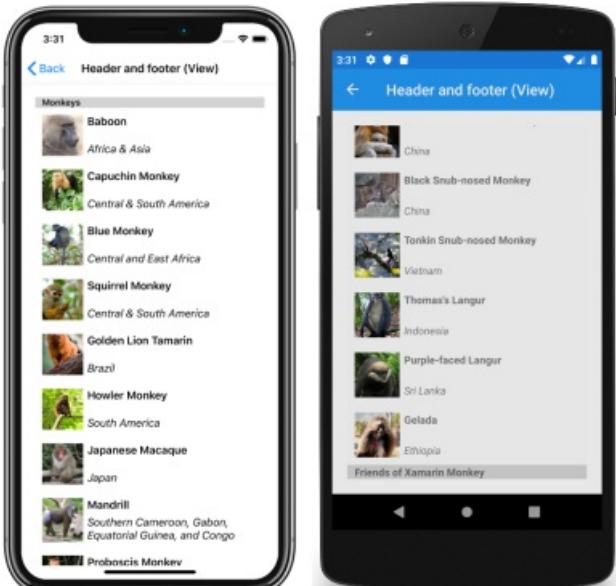
The equivalent C# code is:

```

CollectionView collectionView = new CollectionView
{
    Header = new StackLayout
    {
        Children =
        {
            new Label { Text = "Monkeys", ... }
        }
    },
    Footer = new StackLayout
    {
        Children =
        {
            new Label { Text = "Friends of Xamarin Monkey", ... }
        }
    }
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");

```

This code results in the following screenshots, with the header shown in the iOS screenshot, and the footer shown in the Android screenshot:



Display a templated header and footer

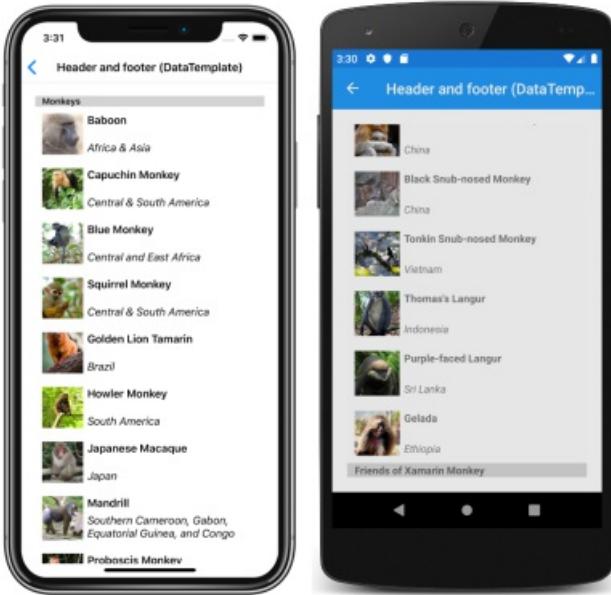
The `HeaderTemplate` and `FooterTemplate` properties can be set to `DataTemplate` objects that are used to format the header and footer. In this scenario, the `Header` and `Footer` properties must bind to the current source for the templates to be applied, as shown in the following example:

```
<CollectionView ItemsSource="{Binding Monkeys}"
    Header="{Binding .}"
    Footer="{Binding .}">
<CollectionView.HeaderTemplate>
    <DataTemplate>
        <StackLayout BackgroundColor="LightGray">
            <Label Margin="10,0,0,0"
                Text="Monkeys"
                FontSize="Small"
                FontAttributes="Bold" />
        </StackLayout>
    </DataTemplate>
</CollectionView.HeaderTemplate>
<CollectionView.FooterTemplate>
    <DataTemplate>
        <StackLayout BackgroundColor="LightGray">
            <Label Margin="10,0,0,0"
                Text="Friends of Xamarin Monkey"
                FontSize="Small"
                FontAttributes="Bold" />
        </StackLayout>
    </DataTemplate>
</CollectionView.FooterTemplate>
...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    HeaderTemplate = new DataTemplate(() =>
    {
        return new StackLayout { };
    }),
    FooterTemplate = new DataTemplate(() =>
    {
        return new StackLayout { };
    })
};
collectionView.SetBinding(ItemsView.HeaderProperty, ".");
collectionView.SetBinding(ItemsView.FooterProperty, ".");
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
```

This code results in the following screenshots, with the header shown in the iOS screenshot, and the footer shown in the Android screenshot:



Item spacing

By default, there is no space between each item in a `CollectionView`. This behavior can be changed by setting properties on the items layout used by the `CollectionView`.

When a `CollectionView` sets its `ItemsLayout` property to a `LinearItemsLayout` object, the `LinearItemsLayout.ItemSpacing` property can be set to a `double` value that represents the space between items:

```
<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical"
            ItemSpacing="20" />
    </CollectionView.ItemsLayout>
    ...
</CollectionView>
```

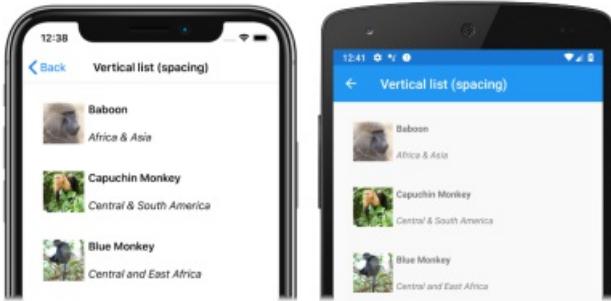
NOTE

The `LinearItemsLayout.ItemSpacing` property has a validation callback set, which ensures that the value of the property is always greater than or equal to 0.

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    ...
    ItemsLayout = new LinearItemsLayout(ItemsLayoutOrientation.Vertical)
    {
        ItemSpacing = 20
    }
};
```

This code results in a vertical single column list, that has a spacing of 20 between items:



When a `CollectionView` sets its `ItemsLayout` property to a `GridItemsLayout` object, the `GridItemsLayout.VerticalItemSpacing` and `GridItemsLayout.HorizontalItemSpacing` properties can be set to double values that represent the empty space vertically and horizontally between items:

```
<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemsLayout>
        <GridItemsLayout Orientation="Vertical"
            Span="2"
            VerticalItemSpacing="20"
            HorizontalItemSpacing="30" />
    </CollectionView.ItemsLayout>
    ...
</CollectionView>
```

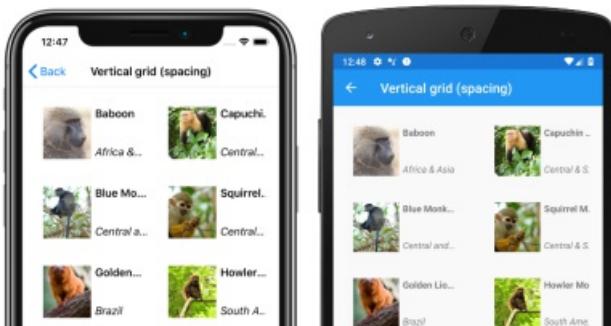
NOTE

The `GridItemsLayout.VerticalItemSpacing` and `GridItemsLayout.HorizontalItemSpacing` properties have validation callbacks set, which ensure that the values of the properties are always greater than or equal to 0.

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    ...
    ItemsLayout = new GridItemsLayout(2, ItemsLayoutOrientation.Vertical)
    {
        VerticalItemSpacing = 20,
        HorizontalItemSpacing = 30
    }
};
```

This code results in a vertical two-column grid, that has a vertical spacing of 20 between items, and a horizontal spacing of 30 between items:



Item sizing

By default, each item in a `CollectionView` is individually measured and sized, provided that the UI elements in the `DataTemplate` don't specify fixed sizes. This behavior, which can be changed, is specified by the `CollectionView.ItemSizingStrategy` property value. This property value can be set to one of the `ItemSizingStrategy` enumeration members:

- `MeasureAllItems` – each item is individually measured. This is the default value.
- `MeasureFirstItem` – only the first item is measured, with all subsequent items being given the same size as the first item.

IMPORTANT

The `MeasureFirstItem` sizing strategy will result in increased performance when used in situations where the item size is intended to be uniform across all items.

The following code example shows setting the `ItemSizingStrategy` property:

```
<CollectionView ...  
    ItemSizingStrategy="MeasureFirstItem">  
    ...  
</CollectionView>
```

The equivalent C# code is:

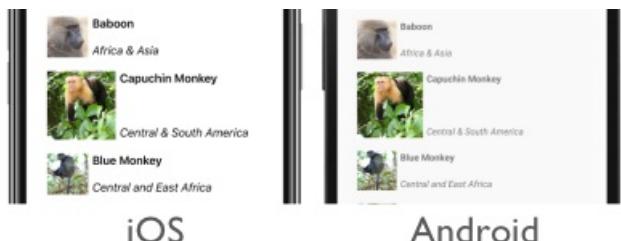
```
CollectionView collectionView = new CollectionView  
{  
    ...  
    ItemSizingStrategy = ItemSizingStrategy.MeasureFirstItem  
};
```

Dynamic resizing of items

Items in a `CollectionView` can be dynamically resized at runtime by changing layout related properties of elements within the `DataTemplate`. For example, the following code example changes the `HeightRequest` and `WidthRequest` properties of an `Image` object:

```
void OnImageTapped(object sender, EventArgs e)  
{  
    Image image = sender as Image;  
    image.HeightRequest = image.WidthRequest = image.HeightRequest.Equals(60) ? 100 : 60;  
}
```

The `OnImageTapped` event handler is executed in response to an `Image` object being tapped, and changes the dimensions of the image so that it's more easily viewed:

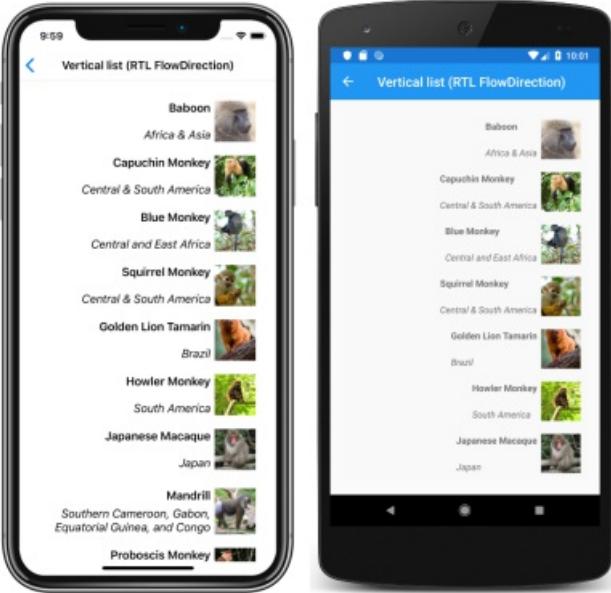


Right-to-left layout

`CollectionView` can layout its content in a right-to-left flow direction by setting its `FlowDirection` property to `RightToLeft`. However, the `FlowDirection` property should ideally be set on a page or root layout, which causes all the elements within the page, or root layout, to respond to the flow direction:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="CollectionViewDemos.Views.VerticalListFlowDirectionPage"
    Title="Vertical list (RTL FlowDirection)"
    FlowDirection="RightToLeft">
    <StackLayout Margin="20">
        <CollectionView ItemsSource="{Binding Monkeys}">
            ...
        </CollectionView>
    </StackLayout>
</ContentPage>
```

The default `FlowDirection` for an element with a parent is `MatchParent`. Therefore, the `CollectionView` inherits the `FlowDirection` property value from the `StackLayout`, which in turn inherits the `FlowDirection` property value from the `ContentPage`. This results in the right-to-left layout shown in the following screenshots:



For more information about flow direction, see [Right-to-left localization](#).

Related links

- [CollectionView \(sample\)](#)
- [Right-to-left localization](#)
- [Xamarin.Forms CollectionView Scrolling](#)

Xamarin.Forms CollectionView Selection

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

`CollectionView` defines the following properties that control item selection:

- `SelectionMode`, of type `SelectionMode`, the selection mode.
- `SelectedItem`, of type `object`, the selected item in the list. This property has a default binding mode of `TwoWay`, and has a `null` value when no item is selected.
- `SelectedItems`, of type `IList<object>`, the selected items in the list. This property has a default binding mode of `OneWay`, and has a `null` value when no items are selected.
- `SelectionChangedCommand`, of type `ICommand`, which is executed when the selected item changes.
- `SelectionChangedCommandParameter`, of type `object`, which is the parameter that's passed to the `SelectionChangedCommand`.

All of these properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

By default, `CollectionView` selection is disabled. However, this behavior can be changed by setting the `SelectionMode` property value to one of the `SelectionMode` enumeration members:

- `None` – indicates that items cannot be selected. This is the default value.
- `Single` – indicates that a single item can be selected, with the selected item being highlighted.
- `Multiple` – indicates that multiple items can be selected, with the selected items being highlighted.

`CollectionView` defines a `SelectionChanged` event that is fired when the `SelectedItem` property changes, either due to the user selecting an item from the list, or when an application sets the property. In addition, this event is also fired when the `SelectedItems` property changes. The `SelectionChangedEventArgs` object that accompanies the `SelectionChanged` event has two properties, both of type `IReadOnlyList<object>`:

- `PreviousSelection` – the list of items that were selected, before the selection changed.
- `CurrentSelection` – the list of items that are selected, after the selection change.

In addition, `CollectionView` has a `UpdateSelectedItems` method that updates the `SelectedItems` property with a list of selected items, while only firing a single change notification.

Single selection

When the `SelectionMode` property is set to `Single`, a single item in the `CollectionView` can be selected. When an item is selected, the `SelectedItem` property will be set to the value of the selected item. When this property changes, the `SelectionChangedCommand` is executed (with the value of the `SelectionChangedCommandParameter` being passed to the `ICommand`), and the `SelectionChanged` event fires.

The following XAML example shows a `CollectionView` that can respond to single item selection:

```

<CollectionView ItemsSource="{Binding Monkeys}"
    SelectionMode="Single"
    SelectionChanged="OnCollectionViewSelectionChanged">
...
</CollectionView>

```

The equivalent C# code is:

```

CollectionView collectionView = new CollectionView
{
    SelectionMode = SelectionMode.Single
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
collectionView.SelectionChanged += OnCollectionViewSelectionChanged;

```

In this code example, the `OnCollectionViewSelectionChanged` event handler is executed when the `SelectionChanged` event fires, with the event handler retrieving the previously selected item, and the current selected item:

```

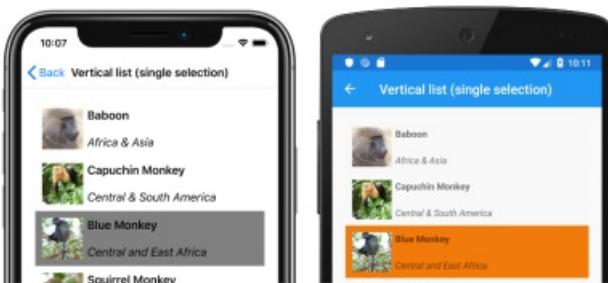
void OnCollectionViewSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    string previous = (e.PreviousSelection.FirstOrDefault() as Monkey)?.Name;
    string current = (e.CurrentSelection.FirstOrDefault() as Monkey)?.Name;
    ...
}

```

IMPORTANT

The `SelectionChanged` event can be fired by changes that occur as a result of changing the `SelectionMode` property.

The following screenshots show single item selection in a `CollectionView`:



Multiple selection

When the `SelectionMode` property is set to `Multiple`, multiple items in the `CollectionView` can be selected. When items are selected, the `SelectedItems` property will be set to the selected items. When this property changes, the `SelectionChangedCommand` is executed (with the value of the `SelectionChangedCommandParameter` being passed to the `ICommand`), and the `SelectionChanged` event fires.

The following XAML example shows a `CollectionView` that can respond to multiple item selection:

```

<CollectionView ItemsSource="{Binding Monkeys}"
    SelectionMode="Multiple"
    SelectionChanged="OnCollectionViewSelectionChanged">
...
</CollectionView>

```

The equivalent C# code is:

```

CollectionView collectionView = new CollectionView
{
    SelectionMode = SelectionMode.Multiple
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
collectionView.SelectionChanged += OnCollectionViewSelectionChanged;

```

In this code example, the `OnCollectionViewSelectionChanged` event handler is executed when the `SelectionChanged` event fires, with the event handler retrieving the previously selected items, and the current selected items:

```

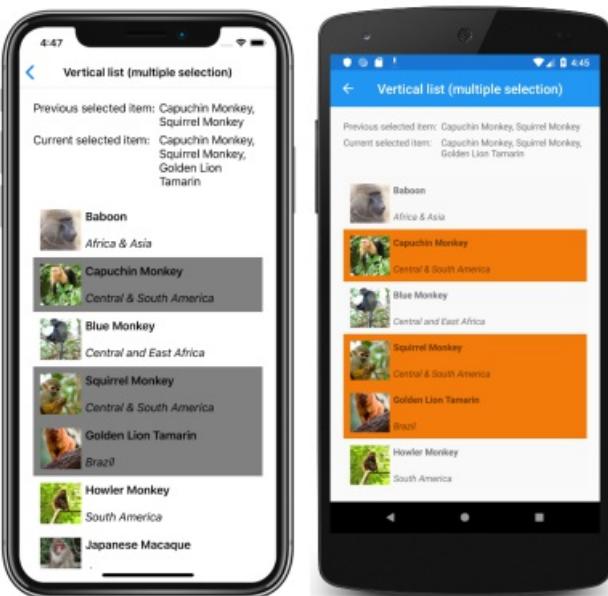
void OnCollectionViewSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    var previous = e.PreviousSelection;
    var current = e.CurrentSelection;
    ...
}

```

IMPORTANT

The `SelectionChanged` event can be fired by changes that occur as a result of changing the `SelectionMode` property.

The following screenshots show multiple item selection in a `CollectionView`:



Single pre-selection

When the `SelectionMode` property is set to `Single`, a single item in the `CollectionView` can be pre-selected by setting the `SelectedItem` property to the item. The following XAML example shows a `CollectionView` that pre-selects a single item:

```

<CollectionView ItemsSource="{Binding Monkeys}"
    SelectionMode="Single"
    SelectedItem="{Binding SelectedMonkey}>
...
</CollectionView>

```

The equivalent C# code is:

```

CollectionView collectionView = new CollectionView
{
    SelectionMode = SelectionMode.Single
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
collectionView.SetBinding(SelectableItemsView.SelectedItemProperty, "SelectedMonkey");

```

NOTE

The `SelectedItem` property has a default binding mode of `TwoWay`.

The `SelectedItem` property data binds to the `SelectedMonkey` property of the connected view model, which is of type `Monkey`. By default, a `TwoWay` binding is used so that if the user changes the selected item, the value of the `SelectedMonkey` property will be set to the selected `Monkey` object. The `SelectedMonkey` property is defined in the `MonkeysViewModel` class, and is set to the fourth item of the `Monkeys` collection:

```

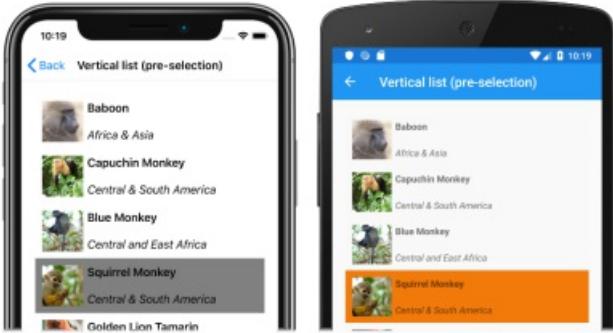
public class MonkeysViewModel : INotifyPropertyChanged
{
    ...
    public ObservableCollection<Monkey> Monkeys { get; private set; }

    Monkey selectedMonkey;
    public Monkey SelectedMonkey
    {
        get
        {
            return selectedMonkey;
        }
        set
        {
            if (selectedMonkey != value)
            {
                selectedMonkey = value;
            }
        }
    }

    public MonkeysViewModel()
    {
        ...
        selectedMonkey = Monkeys.Skip(3).FirstOrDefault();
    }
    ...
}

```

Therefore, when the `CollectionView` appears, the fourth item in the list is pre-selected:



Multiple pre-selection

When the `SelectionMode` property is set to `Multiple`, multiple items in the `CollectionView` can be pre-selected.

The following XAML example shows a `CollectionView` that will enable the pre-selection of multiple items:

```
<CollectionView x:Name="collectionView"
    ItemsSource="{Binding Monkeys}"
    SelectionMode="Multiple"
    SelectedItems="{Binding SelectedMonkeys}">
    ...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    SelectionMode = SelectionMode.Multiple
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
collectionView.SetBinding(SelectableItemsView.SelectedItemsProperty, "SelectedMonkeys");
```

NOTE

The `SelectedItems` property has a default binding mode of `OneWay`.

The `SelectedItems` property data binds to the `SelectedMonkeys` property of the connected view model, which is of type `ObservableCollection<object>`. The `SelectedMonkeys` property is defined in the `MonkeysViewModel` class, and is set to the second, fourth, and fifth items in the `Monkeys` collection:

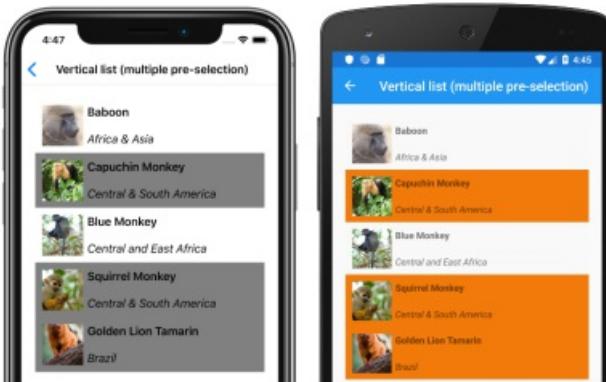
```

namespace CollectionViewDemos.ViewModels
{
    public class MonkeysViewModel : INotifyPropertyChanged
    {
        ...
        ObservableCollection<object> selectedMonkeys;
        public ObservableCollection<object> SelectedMonkeys
        {
            get
            {
                return selectedMonkeys;
            }
            set
            {
                if (selectedMonkeys != value)
                {
                    selectedMonkeys = value;
                }
            }
        }

        public MonkeysViewModel()
        {
            ...
            SelectedMonkeys = new ObservableCollection<object>()
            {
                Monkeys[1], Monkeys[3], Monkeys[4]
            };
        }
        ...
    }
}

```

Therefore, when the `CollectionView` appears, the second, fourth, and fifth items in the list are pre-selected:



Clear selections

The `SelectedItem` and `SelectedItems` properties can be cleared by setting them, or the objects they bind to, to `null`.

Change selected item color

`CollectionView` has a `Selected VisualState` that can be used to initiate a visual change to the selected item in the `CollectionView`. A common use case for this `VisualState` is to change the background color of the selected item, which is shown in the following XAML example:

```

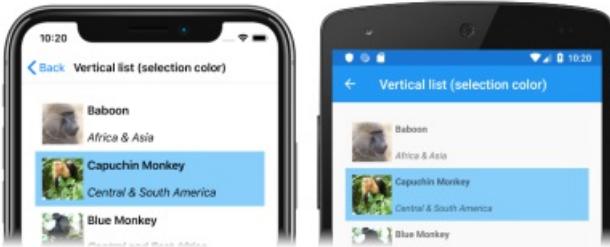
<ContentPage ...>
    <ContentPage.Resources>
        <Style TargetType="Grid">
            <Setter Property="VisualStateManager.VisualStateGroups">
                <VisualStateGroupList>
                    <VisualStateGroup x:Name="CommonStates">
                        <VisualState x:Name="Normal" />
                        <VisualState x:Name="Selected">
                            <VisualState.Setters>
                                <Setter Property="BackgroundColor"
                                       Value="LightSkyBlue" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
                </VisualStateGroupList>
            </Setter>
        </Style>
    </ContentPage.Resources>
    <StackLayout Margin="20">
        <CollectionView ItemsSource="{Binding Monkeys}"
                      SelectionMode="Single">
            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <Grid Padding="10">
                        ...
                    </Grid>
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>
    </StackLayout>
</ContentPage>

```

IMPORTANT

The `Style` that contains the `Selected` `VisualState` must have a `TargetType` property value that's the type of the root element of the `DataTemplate`, which is set as the `ItemTemplate` property value.

In this example, the `Style.TargetType` property value is set to `Grid` because the root element of the `ItemTemplate` is a `Grid`. The `Selected` `VisualState` specifies that when an item in the `CollectionView` is selected, the `BackgroundColor` of the item will be set to `LightSkyBlue`:



For more information about visual states, see [Xamarin.Forms Visual State Manager](#).

Disable selection

`CollectionView` selection is disabled by default. However, if a `CollectionView` has selection enabled, it can be disabled by setting the `SelectionMode` property to `None`:

```

<CollectionView ...
               SelectionMode="None" />

```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView  
{  
    ...  
    SelectionMode = SelectionMode.None  
};
```

When the `SelectionMode` property is set to `None`, items in the `CollectionView` cannot be selected, the `SelectedItem` property will remain `null`, and the `SelectionChanged` event will not be fired.

NOTE

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 `SelectionChanged` event will be fired with an empty `CurrentSelection` property.

Related links

- [CollectionView \(sample\)](#)
- [Xamarin.Forms Visual State Manager](#)

Xamarin.Forms CollectionView EmptyView

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

`CollectionView` defines the following properties that can be used to provide user feedback when there's no data to display:

- `EmptyView`, of type `object`, the string, binding, or view that will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The default value is `null`.
- `EmptyViewTemplate`, of type `DataTemplate`, the template to use to format the specified `EmptyView`. The default value is `null`.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

The main usage scenarios for setting the `EmptyView` property are displaying user feedback when a filtering operation on a `CollectionView` yields no data, and displaying user feedback while data is being retrieved from a web service.

NOTE

The `EmptyView` property can be set to a view that includes interactive content if required.

For more information about data templates, see [Xamarin.Forms Data Templates](#).

Display a string when data is unavailable

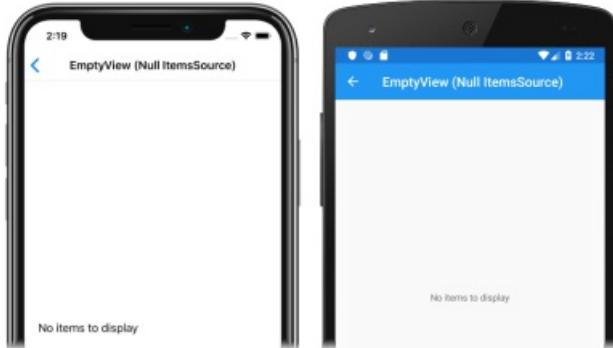
The `EmptyView` property can be set to a string, which will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The following XAML shows an example of this scenario:

```
<CollectionView ItemsSource="{Binding EmptyMonkeys}"
               EmptyView="No items to display" />
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    EmptyView = "No items to display"
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "EmptyMonkeys");
```

The result is that, because the data bound collection is `null`, the string set as the `EmptyView` property value is displayed:



Display views when data is unavailable

The `EmptyView` property can be set to a view, which will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. This can be a single view, or a view that contains multiple child views. The following XAML example shows the `EmptyView` property set to a view that contains multiple child views:

```
<StackLayout Margin="20">
    <SearchBar x:Name="searchBar"
        SearchCommand="{Binding FilterCommand}"
        SearchCommandParameter="{Binding Source={x:Reference searchBar}, Path=Text}"
        Placeholder="Filter" />
    <CollectionView ItemsSource="{Binding Monkeys}">
        <CollectionView.ItemTemplate>
            <DataTemplate>
                ...
            </DataTemplate>
        </CollectionView.ItemTemplate>
        <CollectionView.EmptyView>
            <ContentView>
                <StackLayout HorizontalOptions="CenterAndExpand"
                    VerticalOptions="CenterAndExpand">
                    <Label Text="No results matched your filter."
                        Margin="10,25,10,10"
                        FontAttributes="Bold"
                        FontSize="18"
                        HorizontalOptions="Fill"
                        HorizontalTextAlignment="Center" />
                    <Label Text="Try a broader filter?"
                        FontAttributes="Italic"
                        FontSize="12"
                        HorizontalOptions="Fill"
                        HorizontalTextAlignment="Center" />
                </StackLayout>
            </ContentView>
        </CollectionView.EmptyView>
    </CollectionView>
</StackLayout>
```

In this example, what looks like a redundant `ContentView` has been added as the root element of the `EmptyView`. This is because internally, the `EmptyView` is added to a native container that doesn't provide any context for Xamarin.Forms layout. Therefore, to position the views that comprise your `EmptyView`, you must add a root layout, whose child is a layout that can position itself within the root layout.

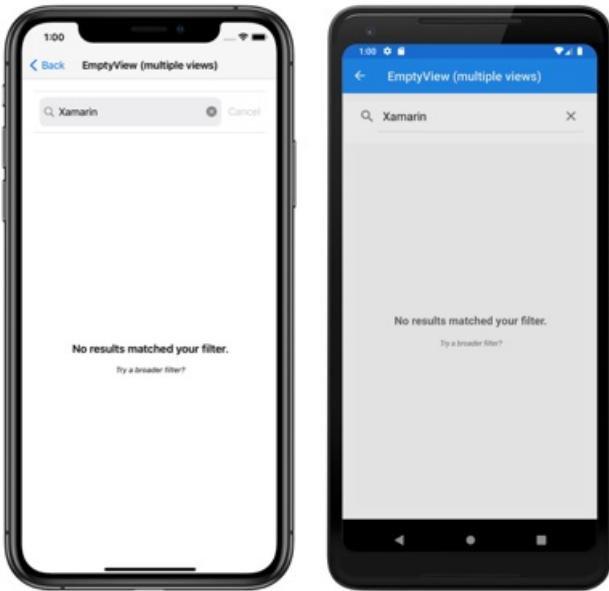
The equivalent C# code is:

```

SearchBar searchBar = new SearchBar { ... };
CollectionView collectionView = new CollectionView
{
    EmptyView = new ContentView
    {
        Content = new StackLayout
        {
            Children =
            {
                new Label { Text = "No results matched your filter.", ... },
                new Label { Text = "Try a broader filter?", ... }
            }
        }
    }
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");

```

When the `SearchBar` executes the `FilterCommand`, the collection displayed by the `CollectionView` is filtered for the search term stored in the `SearchBar.Text` property. If the filtering operation yields no data, the `StackLayout` set as the `EmptyView` property value is displayed:



Display a templated custom type when data is unavailable

The `EmptyView` property can be set to a custom type, whose template is displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The `EmptyViewTemplate` property can be set to a `DataTemplate` that defines the appearance of the `EmptyView`. The following XAML shows an example of this scenario:

```

<StackLayout Margin="20">
    <SearchBar x:Name="searchBar"
        SearchCommand="{Binding FilterCommand}"
        SearchCommandParameter="{Binding Source={x:Reference searchBar}, Path=Text}"
        Placeholder="Filter" />
    <CollectionView ItemsSource="{Binding Monkeys}">
        <CollectionView.ItemTemplate>
            <DataTemplate>
                ...
            </DataTemplate>
        </CollectionView.ItemTemplate>
        <CollectionView.EmptyView>
            <views:FilterData Filter="{Binding Source={x:Reference searchBar}, Path=Text}" />
        </CollectionView.EmptyView>
        <CollectionView.EmptyViewTemplate>
            <DataTemplate>
                <Label Text="{Binding Filter, StringFormat='Your filter term of {0} did not match any
records.'}" Margin="10,25,10,10"
                    FontAttributes="Bold"
                    FontSize="18"
                    HorizontalOptions="Fill"
                    HorizontalTextAlignment="Center" />
            </DataTemplate>
        </CollectionView.EmptyViewTemplate>
    </CollectionView>
</StackLayout>

```

The equivalent C# code is:

```

SearchBar searchBar = new SearchBar { ... };
CollectionView collectionView = new CollectionView
{
    EmptyView = new FilterData { Filter = searchBar.Text },
    EmptyViewTemplate = new DataTemplate(() =>
    {
        return new Label { ... };
    })
};

```

The `FilterData` type defines a `Filter` property, and a corresponding `BindableProperty`:

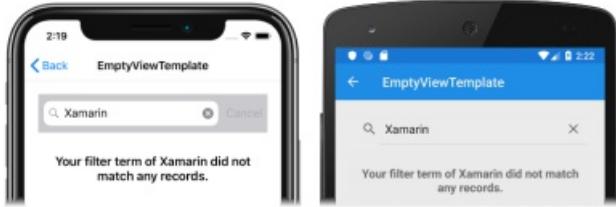
```

public class FilterData : BindableObject
{
    public static readonly BindableProperty FilterProperty = BindableProperty.Create(nameof(Filter),
typeof(string), typeof(FilterData), null);

    public string Filter
    {
        get { return (string)GetValue(FilterProperty); }
        set { SetValue(FilterProperty, value); }
    }
}

```

The `EmptyView` property is set to a `FilterData` object, and the `Filter` property data binds to the `SearchBar.Text` property. When the `SearchBar` executes the `FilterCommand`, the collection displayed by the `CollectionView` is filtered for the search term stored in the `Filter` property. If the filtering operation yields no data, the `Label` defined in the `DataTemplate`, that's set as the `EmptyViewTemplate` property value, is displayed:



NOTE

When displaying a templated custom type when data is unavailable, the `EmptyViewTemplate` property can be set to a view that contains multiple child views.

Choose an EmptyView at runtime

Views that will be displayed as an `EmptyView` when data is unavailable, can be defined as `ContentView` objects in a `ResourceDictionary`. The `EmptyView` property can then be set to a specific `ContentView`, based on some business logic, at runtime. The following XAML shows an example of this scenario:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="CollectionViewDemos.Views.EmptyViewSwapPage"
    Title="EmptyView (swap)">
    <ContentPage.Resources>
        <ContentView x:Key="BasicEmptyView">
            <StackLayout>
                <Label Text="No items to display."
                    Margin="10,25,10,10"
                    FontAttributes="Bold"
                    FontSize="18"
                    HorizontalOptions="Fill"
                    HorizontalTextAlignment="Center" />
            </StackLayout>
        </ContentView>
        <ContentView x:Key="AdvancedEmptyView">
            <StackLayout>
                <Label Text="No results matched your filter."
                    Margin="10,25,10,10"
                    FontAttributes="Bold"
                    FontSize="18"
                    HorizontalOptions="Fill"
                    HorizontalTextAlignment="Center" />
                <Label Text="Try a broader filter?">
                    <FontAttributes="Italic" />
                    <FontSize="12" />
                    <HorizontalOptions="Fill" />
                    <HorizontalTextAlignment="Center" />
                </Label>
            </StackLayout>
        </ContentView>
    </ContentPage.Resources>

    <StackLayout Margin="20">
        <SearchBar x:Name="searchBar"
            SearchCommand="{Binding FilterCommand}"
            SearchCommandParameter="{Binding Source={x:Reference searchBar}, Path=Text}"
            Placeholder="Filter" />
        <StackLayout Orientation="Horizontal">
            <Label Text="Toggle EmptyViews" />
            <Switch Toggled="OnEmptyViewSwitchToggled" />
        </StackLayout>
        <CollectionView x:Name="collectionView"
            ItemsSource="{Binding Monkeys}">
            <CollectionView.ItemTemplate>
                <DataTemplate>
                    ...
                    </DataTemplate>
                </CollectionView.ItemTemplate>
            </CollectionView>
        </StackLayout>
    </ContentPage>

```

This XAML defines two `ContentView` objects in the page-level `ResourceDictionary`, with the `Switch` object controlling which `ContentView` object will be set as the `EmptyView` property value. When the `Switch` is toggled, the `OnEmptyViewSwitchToggled` event handler executes the `ToggleEmptyView` method:

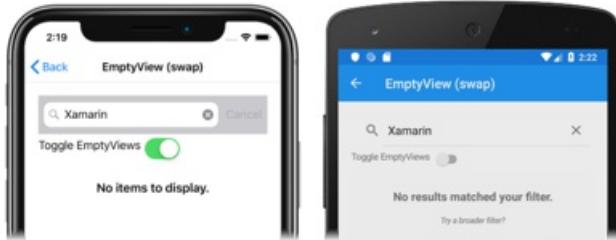
```

void ToggleEmptyView(bool isToggled)
{
    collectionView.EmptyView = isToggled ? Resources["BasicEmptyView"] : Resources["AdvancedEmptyView"];
}

```

The `ToggleEmptyView` method sets the `EmptyView` property of the `collectionView` object to one of the two `ContentView` objects stored in the `ResourceDictionary`, based on the value of the `Switch.IsToggled` property.

When the `SearchBar` executes the `FilterCommand`, the collection displayed by the `CollectionView` is filtered for the search term stored in the `SearchBar.Text` property. If the filtering operation yields no data, the `ContentView` object set as the `EmptyView` property is displayed:



For more information about resource dictionaries, see [Xamarin.Forms Resource Dictionaries](#).

Choose an EmptyViewTemplate at runtime

The appearance of the `EmptyView` can be chosen at runtime, based on its value, by setting the `CollectionView.EmptyViewTemplate` property to a `DataTemplateSelector` object:

```
<ContentPage ...>
    xmlns:controls="clr-namespace:CollectionViewDemos.Controls"
    <ContentPage.Resources>
        <DataTemplate x:Key="AdvancedTemplate">
            ...
        </DataTemplate>

        <DataTemplate x:Key="BasicTemplate">
            ...
        </DataTemplate>

        <controls:SearchTermDataTemplateSelector x:Key="SearchSelector">
            DefaultTemplate="{StaticResource AdvancedTemplate}"
            OtherTemplate="{StaticResource BasicTemplate}" />
    </ContentPage.Resources>

    <StackLayout Margin="20">
        <SearchBar x:Name="searchBar"
            SearchCommand="{Binding FilterCommand}"
            SearchCommandParameter="{Binding Source={x:Reference searchBar}, Path=Text}"
            Placeholder="Filter" />
        <CollectionView ItemsSource="{Binding Monkeys}"
            EmptyView="{Binding Source={x:Reference searchBar}, Path=Text}"
            EmptyViewTemplate="{StaticResource SearchSelector}" />
    </StackLayout>
</ContentPage>
```

The equivalent C# code is:

```
SearchBar searchBar = new SearchBar { ... };
CollectionView collectionView = new CollectionView
{
    EmptyView = searchBar.Text,
    EmptyViewTemplate = new searchTermDataTemplateSelector { ... }
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
```

The `EmptyView` property is set to the `SearchBar.Text` property, and the `EmptyViewTemplate` property is set to a `SearchTermDataTemplateSelector` object.

When the `SearchBar` executes the `FilterCommand`, the collection displayed by the `CollectionView` is filtered for

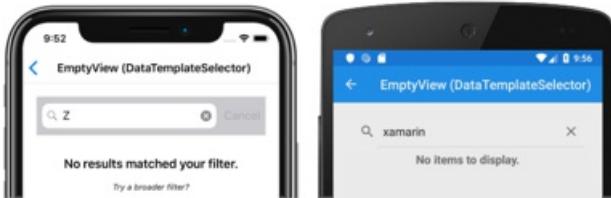
the search term stored in the `SearchBar.Text` property. If the filtering operation yields no data, the `DataTemplate` chosen by the `SearchTermDataTemplateSelector` object is set as the `EmptyViewTemplate` property and displayed.

The following example shows the `SearchTermDataTemplateSelector` class:

```
public class SearchTermDataTemplateSelector : DataTemplateSelector
{
    public DataTemplate DefaultTemplate { get; set; }
    public DataTemplate OtherTemplate { get; set; }

    protected override DataTemplate OnSelectTemplate(object item, BindableObject container)
    {
        string query = (string)item;
        return query.ToLower().Equals("xamarin") ? OtherTemplate : DefaultTemplate;
    }
}
```

The `SearchTermDataTemplateSelector` class defines `DefaultTemplate` and `OtherTemplate` `DataTemplate` properties that are set to different data templates. The `OnSelectTemplate` override returns `DefaultTemplate`, which displays a message to the user, when the search query isn't equal to "xamarin". When the search query is equal to "xamarin", the `OnSelectTemplate` override returns `OtherTemplate`, which displays a basic message to the user:



For more information about data template selectors, see [Create a Xamarin.Forms DataTemplateSelector](#).

Related links

- [CollectionView \(sample\)](#)
- [Xamarin.Forms Data Templates](#)
- [Xamarin.Forms Resource Dictionaries](#)
- [Create a Xamarin.Forms DataTemplateSelector](#)

Xamarin.Forms CollectionView Scrolling

8/4/2022 • 8 minutes to read • [Edit Online](#)



[Download the sample](#)

`CollectionView` defines two `ScrollTo` methods, that scroll items into view. One of the overloads scrolls the item at the specified index into view, while the other scrolls the specified item into view. Both overloads have additional arguments that can be specified to indicate the group the item belongs to, the exact position of the item after the scroll has completed, and whether to animate the scroll.

`CollectionView` defines a `ScrollToRequested` event that is fired when one of the `ScrollTo` methods is invoked. The `ScrollToEventArgs` object that accompanies the `ScrollToRequested` event has many properties, including `IsAnimated`, `Index`, `Item`, and `ScrollToPosition`. These properties are set from the arguments specified in the `ScrollTo` method calls.

In addition, `CollectionView` defines a `Scrolled` event that is fired to indicate that scrolling occurred. The `ItemsViewScrolledEventArgs` object that accompanies the `Scrolled` event has many properties. For more information, see [Detect scrolling](#).

`CollectionView` also defines a `ItemsUpdatingScrollMode` property that represents the scrolling behavior of the `CollectionView` when new items are added to it. For more information about this property, see [Control scroll position when new items are added](#).

When a user swipes to initiate a scroll, the end position of the scroll can be controlled so that items are fully displayed. This feature is known as snapping, because items snap to position when scrolling stops. For more information, see [Snap points](#).

`CollectionView` can also load data incrementally as the user scrolls. For more information, see [Load data incrementally](#).

Detect scrolling

`CollectionView` defines a `Scrolled` event which is fired to indicate that scrolling occurred. The `ItemsViewScrolledEventArgs` class, which represents the object that accompanies the `Scrolled` event, defines the following properties:

- `HorizontalDelta`, of type `double`, represents the change in the amount of horizontal scrolling. This is a negative value when scrolling left, and a positive value when scrolling right.
- `VerticalDelta`, of type `double`, represents the change in the amount of vertical scrolling. This is a negative value when scrolling upwards, and a positive value when scrolling downwards.
- `HorizontalOffset`, of type `double`, defines the amount by which the list is horizontally offset from its origin.
- `VerticalOffset`, of type `double`, defines the amount by which the list is vertically offset from its origin.
- `FirstVisibleItemIndex`, of type `int`, is the index of the first item that's visible in the list.
- `CenterItemIndex`, of type `int`, is the index of the center item that's visible in the list.
- `LastVisibleItemIndex`, of type `int`, is the index of the last item that's visible in the list.

The following XAML example shows a `CollectionView` that sets an event handler for the `Scrolled` event:

```
<CollectionView Scrolled="OnCollectionViewScrolled">
    ...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView();
collectionView.Scrolled += OnCollectionViewScrolled;
```

In this code example, the `OnCollectionViewScrolled` event handler is executed when the `Scrolled` event fires:

```
void OnCollectionViewScrolled(object sender, ItemsViewScrolledEventArgs e)
{
    // Custom logic
}
```

IMPORTANT

The `Scrolled` event is fired for user initiated scrolls, and for programmatic scrolls.

Scroll an item at an index into view

The first `ScrollTo` method overload scrolls the item at the specified index into view. Given a `CollectionView` object named `collectionView`, the following example shows how to scroll the item at index 12 into view:

```
collectionView.ScrollTo(12);
```

Alternatively, an item in grouped data can be scrolled into view by specifying the item and group indexes. The following example shows how to scroll the third item in the second group into view:

```
// Items and groups are indexed from zero.
collectionView.ScrollTo(2, 1);
```

NOTE

The `ScrollToRequested` event is fired when the `ScrollTo` method is invoked.

Scroll an item into view

The second `ScrollTo` method overload scrolls the specified item into view. Given a `CollectionView` object named `collectionView`, the following example shows how to scroll the Proboscis Monkey item into view:

```
MonkeysViewModel viewModel = BindingContext as MonkeysViewModel;
Monkey monkey = viewModel.Monkeys.FirstOrDefault(m => m.Name == "Proboscis Monkey");
collectionView.ScrollTo(monkey);
```

Alternatively, an item in grouped data can be scrolled into view by specifying the item and the group. The following example shows how to scroll the Proboscis Monkey item in the Monkeys group into view:

```
GroupedAnimalsViewModel viewModel = BindingContext as GroupedAnimalsViewModel;
AnimalGroup group = viewModel.Animals.FirstOrDefault(a => a.Name == "Monkeys");
Animal monkey = group.FirstOrDefault(m => m.Name == "Proboscis Monkey");
collectionView.ScrollTo(monkey, group);
```

NOTE

The `ScrollToRequested` event is fired when the `ScrollTo` method is invoked.

Disable scroll animation

A scrolling animation is displayed when scrolling an item into view. However, this animation can be disabled by setting the `animate` argument of the `ScrollTo` method to `false`:

```
collectionView.ScrollTo(monkey, animate: false);
```

Control scroll position

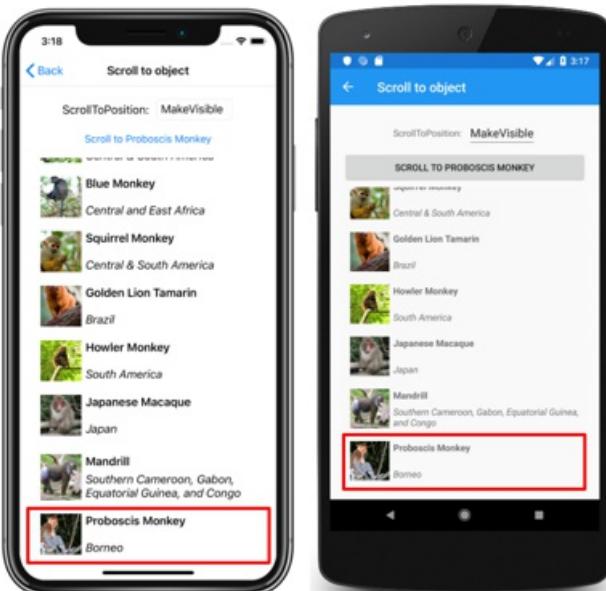
When scrolling an item into view, the exact position of the item after the scroll has completed can be specified with the `position` argument of the `ScrollTo` methods. This argument accepts a `ScrollToPosition` enumeration member.

MakeVisible

The `ScrollToPosition.MakeVisible` member indicates that the item should be scrolled until it's visible in the view:

```
collectionView.ScrollTo(monkey, position: ScrollToPosition.MakeVisible);
```

This example code results in the minimal scrolling required to scroll the item into view:



NOTE

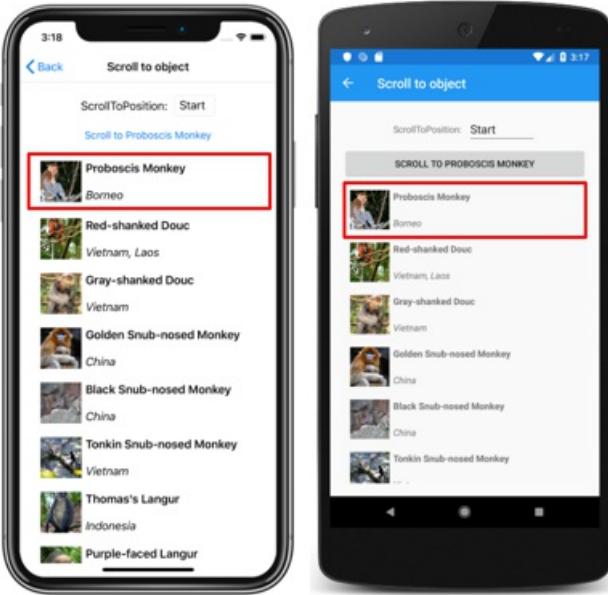
The `ScrollToPosition.MakeVisible` member is used by default, if the `position` argument is not specified when calling the `ScrollTo` method.

Start

The `ScrollToPosition.Start` member indicates that the item should be scrolled to the start of the view:

```
collectionView.ScrollTo(monkey, position: ScrollToPosition.Start);
```

This example code results in the item being scrolled to the start of the view:

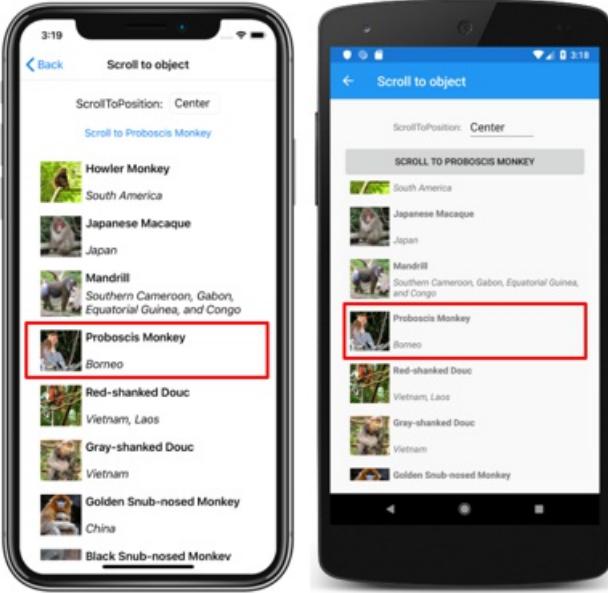


Center

The `ScrollToPosition.Center` member indicates that the item should be scrolled to the center of the view:

```
collectionView.ScrollTo(monkey, position: ScrollToPosition.Center);
```

This example code results in the item being scrolled to the center of the view:

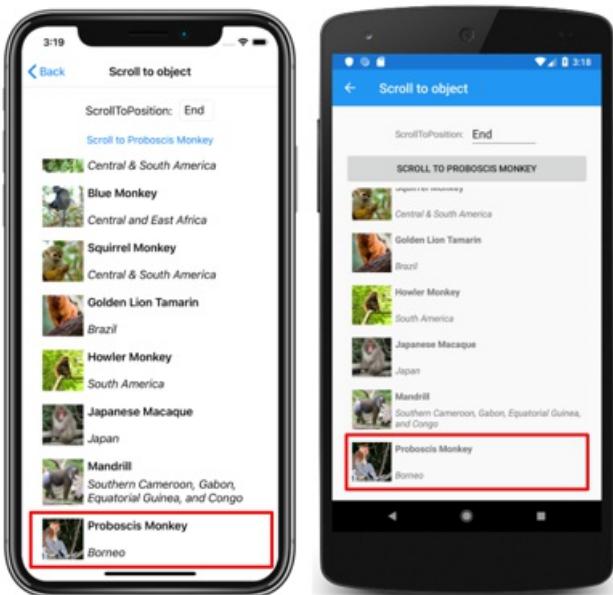


End

The `ScrollToPosition.End` member indicates that the item should be scrolled to the end of the view:

```
collectionView.ScrollTo(monkey, position: ScrollToPosition.End);
```

This example code results in the item being scrolled to the end of the view:



Control scroll position when new items are added

`CollectionView` defines a `ItemsUpdatingScrollMode` property, which is backed by a bindable property. This property gets or sets a `ItemsUpdatingScrollMode` enumeration value that represents the scrolling behavior of the `CollectionView` when new items are added to it. The `ItemsUpdatingScrollMode` enumeration defines the following members:

- `KeepItemsInView` keeps the first item in the list displayed when new items are added.
- `KeepScrollOffset` ensures that the current scroll position is maintained when new items are added.
- `KeepLastItemInView` adjusts the scroll offset to keep the last item in the list displayed when new items are added.

The default value of the `ItemsUpdatingScrollMode` property is `KeepItemsInView`. Therefore, when new items are added to a `CollectionView` the first item in the list will remain displayed. To ensure that the last item in the list is displayed when new items are added, set the `ItemsUpdatingScrollMode` property to `KeepLastItemInView`:

```
<CollectionView ItemsUpdatingScrollMode="KeepLastItemInView">
    ...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    ItemsUpdatingScrollMode = ItemsUpdatingScrollMode.KeepLastItemInView
};
```

Scroll bar visibility

`CollectionView` defines `HorizontalScrollBarVisibility` and `VerticalScrollBarVisibility` properties, which are backed by bindable properties. These properties get or set a `ScrollBarVisibility` enumeration value that represents when the horizontal, or vertical, scroll bar is visible. The `ScrollBarVisibility` enumeration defines the following members:

- `Default` indicates the default scroll bar behavior for the platform, and is the default value for the

`HorizontalScrollBarVisibility` and `VerticalScrollBarVisibility` properties.

- `Always` indicates that scroll bars will be visible, even when the content fits in the view.
- `Never` indicates that scroll bars will not be visible, even if the content doesn't fit in the view.

Snap points

When a user swipes to initiate a scroll, the end position of the scroll can be controlled so that items are fully displayed. This feature is known as snapping, because items snap to position when scrolling stops, and is controlled by the following properties from the `ItemsLayout` class:

- `SnapPointsType`, of type `SnapPointsType`, specifies the behavior of snap points when scrolling.
- `SnapPointsAlignment`, of type `SnapPointsAlignment`, specifies how snap points are aligned with items.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

NOTE

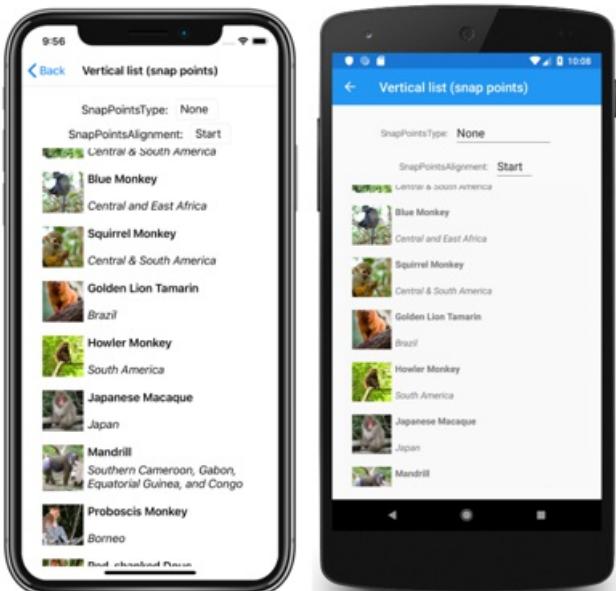
When snapping occurs, it will occur in the direction that produces the least amount of motion.

Snap points type

The `SnapPointsType` enumeration defines the following members:

- `None` indicates that scrolling does not snap to items.
- `Mandatory` indicates that content always snaps to the closest snap point to where scrolling would naturally stop, along the direction of inertia.
- `MandatorySingle` indicates the same behavior as `Mandatory`, but only scrolls one item at a time.

By default, the `SnapPointsType` property is set to `SnapPointsType.None`, which ensures that scrolling does not snap items, as shown in the following screenshots:



Snap points alignment

The `SnapPointsAlignment` enumeration defines `Start`, `Center`, and `End` members.

IMPORTANT

The value of the `SnapPointsAlignment` property is only respected when the `SnapPointsType` property is set to `Mandatory`, or `MandatorySingle`.

Start

The `SnapPointsAlignment.Start` member indicates that snap points are aligned with the leading edge of items.

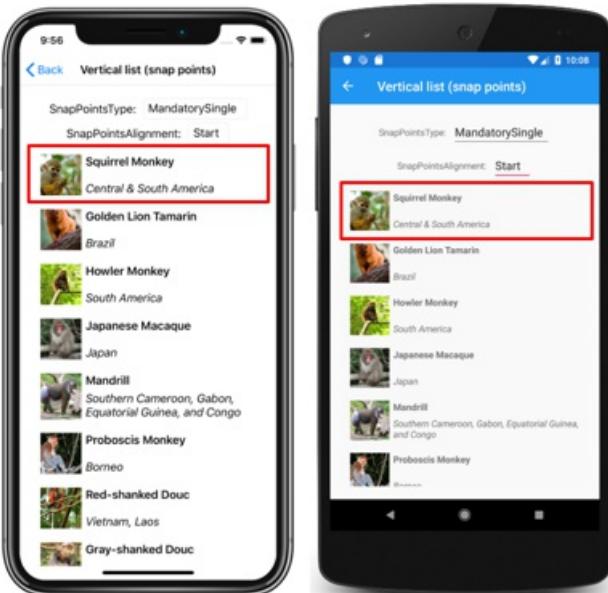
By default, the `SnapPointsAlignment` property is set to `SnapPointsAlignment.Start`. However, for completeness, the following XAML example shows how to set this enumeration member:

```
<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical"
            SnapPointsType="MandatorySingle"
            SnapPointsAlignment="Start" />
    </CollectionView.ItemsLayout>
    ...
</CollectionView>
```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    ItemsLayout = new LinearItemsLayout(ItemsLayoutOrientation.Vertical)
    {
        SnapPointsType = SnapPointsType.MandatorySingle,
        SnapPointsAlignment = SnapPointsAlignment.Start
    },
    // ...
};
```

When a user swipes to initiate a scroll, the top item will be aligned with the top of the view:



Center

The `SnapPointsAlignment.Center` member indicates that snap points are aligned with the center of items. The following XAML example shows how to set this enumeration member:

```

<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical"
            SnapPointsType="MandatorySingle"
            SnapPointsAlignment="Center" />
    </CollectionView.ItemsLayout>
    ...
</CollectionView>

```

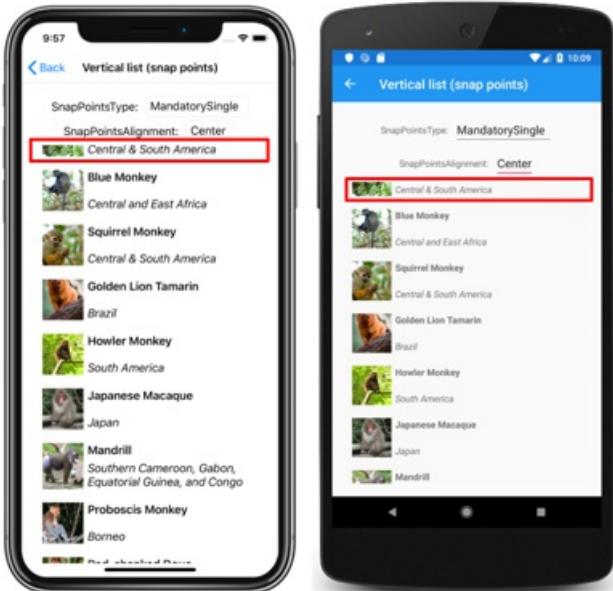
The equivalent C# code is:

```

CollectionView collectionView = new CollectionView
{
    ItemsLayout = new LinearItemsLayout(ItemsLayoutOrientation.Vertical)
    {
        SnapPointsType = SnapPointsType.MandatorySingle,
        SnapPointsAlignment = SnapPointsAlignment.Center
    },
    // ...
};

```

When a user swipes to initiate a scroll, the top item will be center aligned at the top of the view:



End

The `SnapPointsAlignment.End` member indicates that snap points are aligned with the trailing edge of items. The following XAML example shows how to set this enumeration member:

```

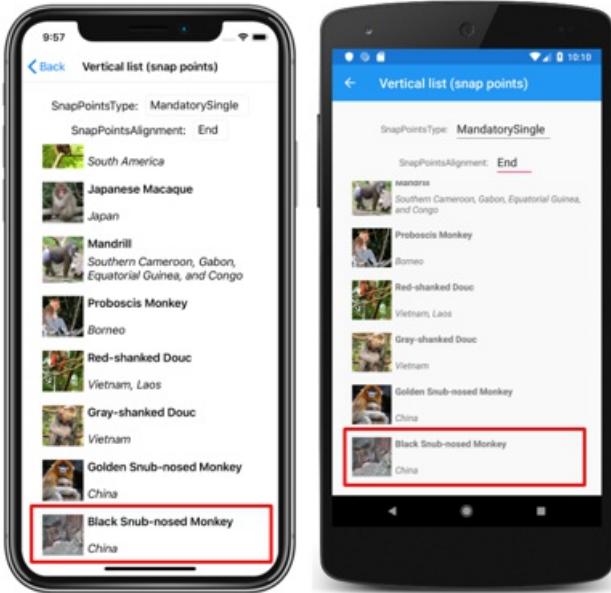
<CollectionView ItemsSource="{Binding Monkeys}">
    <CollectionView.ItemsLayout>
        <LinearItemsLayout Orientation="Vertical"
            SnapPointsType="MandatorySingle"
            SnapPointsAlignment="End" />
    </CollectionView.ItemsLayout>
    ...
</CollectionView>

```

The equivalent C# code is:

```
CollectionView collectionView = new CollectionView
{
    ItemsLayout = new LinearItemsLayout(ItemsLayoutOrientation.Vertical)
    {
        SnapPointsType = SnapPointsType.MandatorySingle,
        SnapPointsAlignment = SnapPointsAlignment.End
    },
    // ...
};
```

When a user swipes to initiate a scroll, the bottom item will be aligned with the bottom of the view:



Related links

- [CollectionView \(sample\)](#)

Xamarin.Forms CollectionView Grouping

8/4/2022 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

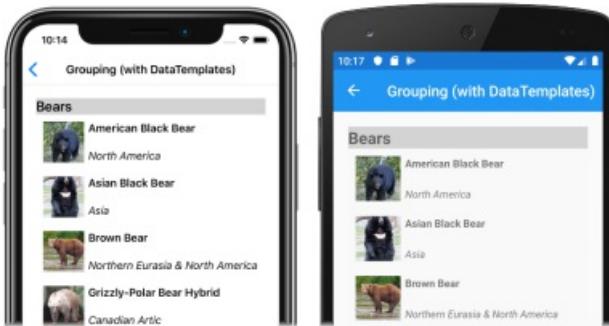
Large data sets can often become unwieldy when presented in a continually scrolling list. In this scenario, organizing the data into groups can improve the user experience by making it easier to navigate the data.

`CollectionView` supports displaying grouped data, and defines the following properties that control how it will be presented:

- `IsGrouped`, of type `bool`, indicates whether the underlying data should be displayed in groups. The default value of this property is `false`.
- `GroupHeaderTemplate`, of type `DataTemplate`, the template to use for the header of each group.
- `GroupFooterTemplate`, of type `DataTemplate`, the template to use for the footer of each group.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings.

The following screenshots show a `CollectionView` displaying grouped data:



For more information about data templates, see [Xamarin.Forms Data Templates](#).

Group data

Data must be grouped before it can be displayed. This can be accomplished by creating a list of groups, where each group is a list of items. The list of groups should be an `IEnumerable<T>` collection, where `T` defines two pieces of data:

- A group name.
- An `IEnumerable` collection that defines the items belonging to the group.

The process for grouping data, therefore, is to:

- Create a type that models a single item.
- Create a type that models a single group of items.
- Create an `IEnumerable<T>` collection, where `T` is the type that models a single group of items. This collection is therefore a collection of groups, which stores the grouped data.
- Add data to the `IEnumerable<T>` collection.

Example

When grouping data, the first step is to create a type that models a single item. The following example shows

the `Animal` class from the sample application:

```
public class Animal
{
    public string Name { get; set; }
    public string Location { get; set; }
    public string Details { get; set; }
    public string ImageUrl { get; set; }
}
```

The `Animal` class models a single item. A type that models a group of items can then be created. The following example shows the `AnimalGroup` class from the sample application:

```
public class AnimalGroup : List<Animal>
{
    public string Name { get; private set; }

    public AnimalGroup(string name, List<Animal> animals) : base(animals)
    {
        Name = name;
    }
}
```

The `AnimalGroup` class inherits from the `List<T>` class and adds a `Name` property that represents the group name.

An `IEnumerable<T>` collection of groups can then be created:

```
public List<AnimalGroup> Animals { get; private set; } = new List<AnimalGroup>();
```

This code defines a collection named `Animals`, where each item in the collection is an `AnimalGroup` object. Each `AnimalGroup` object comprises a name, and a `List<Animal>` collection that defines the `Animal` objects in the group.

Grouped data can then be added to the `Animals` collection:

```

Animals.Add(new AnimalGroup("Bears", new List<Animal>
{
    new Animal
    {
        Name = "American Black Bear",
        Location = "North America",
        Details = "Details about the bear go here.",
        ImageUrl = "https://upload.wikimedia.org/wikipedia/commons/0/08/01_Schwarzbär.jpg"
    },
    new Animal
    {
        Name = "Asian Black Bear",
        Location = "Asia",
        Details = "Details about the bear go here.",
        ImageUrl =
"https://upload.wikimedia.org/wikipedia/commons/thumb/b/b7/Ursus_thibetanus_3_%28Wroclaw_zoo%29.JPG/180px-Ursus_thibetanus_3_%28Wroclaw_zoo%29.JPG"
    },
    // ...
}));
```



```

Animals.Add(new AnimalGroup("Monkeys", new List<Animal>
{
    new Animal
    {
        Name = "Baboon",
        Location = "Africa & Asia",
        Details = "Details about the monkey go here.",
        ImageUrl =
"https://upload.wikimedia.org/wikipedia/commons/thumb/f/fc/Papio_anubis_%28Serengeti%2C_2009%29.jpg/200px-Papio_anubis_%28Serengeti%2C_2009%29.jpg"
    },
    new Animal
    {
        Name = "Capuchin Monkey",
        Location = "Central & South America",
        Details = "Details about the monkey go here.",
        ImageUrl = "https://upload.wikimedia.org/wikipedia/commons/thumb/4/40/Capuchin_Costa_Rica.jpg/200px-Capuchin_Costa_Rica.jpg"
    },
    new Animal
    {
        Name = "Blue Monkey",
        Location = "Central and East Africa",
        Details = "Details about the monkey go here.",
        ImageUrl = "https://upload.wikimedia.org/wikipedia/commons/thumb/8/83/BlueMonkey.jpg/220px-BlueMonkey.jpg"
    },
    // ...
}));
```

This code creates two groups in the `Animals` collection. The first `AnimalGroup` is named `Bears`, and contains a `List<Animal>` collection of bear details. The second `AnimalGroup` is named `Monkeys`, and contains a `List<Animal>` collection of monkey details.

Display grouped data

`CollectionView` will display grouped data, provided that the data has been grouped correctly, by setting the `IsGrouped` property to `true`:

```

<CollectionView ItemsSource="{Binding Animals}"
    IsGrouped="true">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Grid Padding="10">
                ...
                <Image Grid.RowSpan="2"
                    Source="{Binding ImageUrl}"
                    Aspect="AspectFill"
                    HeightRequest="60"
                    WidthRequest="60" />
                <Label Grid.Column="1"
                    Text="{Binding Name}"
                    FontAttributes="Bold" />
                <Label Grid.Row="1"
                    Grid.Column="1"
                    Text="{Binding Location}"
                    FontAttributes="Italic"
                    VerticalOptions="End" />
            </Grid>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>

```

The equivalent C# code is:

```

CollectionView collectionView = new CollectionView
{
    IsGrouped = true
};
collectionView.SetBinding(ItemsView.ItemsSourceProperty, "Animals");
// ...

```

The appearance of each item in the `CollectionView` is defined by setting the `CollectionView.ItemTemplate` property to a `DataTemplate`. For more information, see [Define item appearance](#).

NOTE

By default, `CollectionView` will display the group name in the group header and footer. This behavior can be changed by customizing the group header and group footer.

Customize the group header

The appearance of each group header can be customized by setting the `CollectionView.GroupHeaderTemplate` property to a `DataTemplate`:

```

<CollectionView ItemsSource="{Binding Animals}"
    IsGrouped="true">
    ...
    <CollectionView.GroupHeaderTemplate>
        <DataTemplate>
            <Label Text="{Binding Name}"
                BackgroundColor="LightGray"
                FontSize="Large"
                FontAttributes="Bold" />
        </DataTemplate>
    </CollectionView.GroupHeaderTemplate>
</CollectionView>

```

In this example, each group header is set to a `Label` that displays the group name, and that has other appearance properties set. The following screenshots show the customized group header:

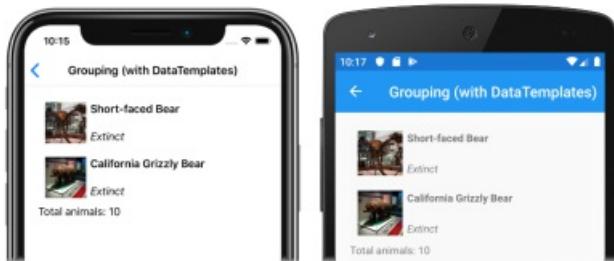


Customize the group footer

The appearance of each group footer can be customized by setting the `CollectionView.GroupFooterTemplate` property to a `DataTemplate`:

```
<CollectionView ItemsSource="{Binding Animals}"
    IsGrouped="true">
    ...
    <CollectionView.GroupFooterTemplate>
        <DataTemplate>
            <Label Text="{Binding Count, StringFormat='Total animals: {0:D}'}"
                Margin="0,0,0,10" />
        </DataTemplate>
    </CollectionView.GroupFooterTemplate>
</CollectionView>
```

In this example, each group footer is set to a `Label` that displays the number of items in the group. The following screenshots show the customized group footer:



Empty groups

When a `CollectionView` displays grouped data, it will display any groups that are empty. Such groups will be displayed with a group header and footer, indicating that the group is empty. The following screenshots show an empty group:



NOTE

On iOS 10 and lower, group headers and footers for empty groups may all be displayed at the top of the `CollectionView`.

Group without templates

`CollectionView` can display correctly grouped data without setting the `CollectionView.ItemTemplate` property to a `DataTemplate`:

```
<CollectionView ItemsSource="{Binding Animals}"  
    IsGrouped="true" />
```

In this scenario, meaningful data can be displayed by overriding the `ToString` method in the type that models a single item, and the type that models a single group of items.

Related links

- [CollectionView \(sample\)](#)
- [Xamarin.Forms Data Templates](#)

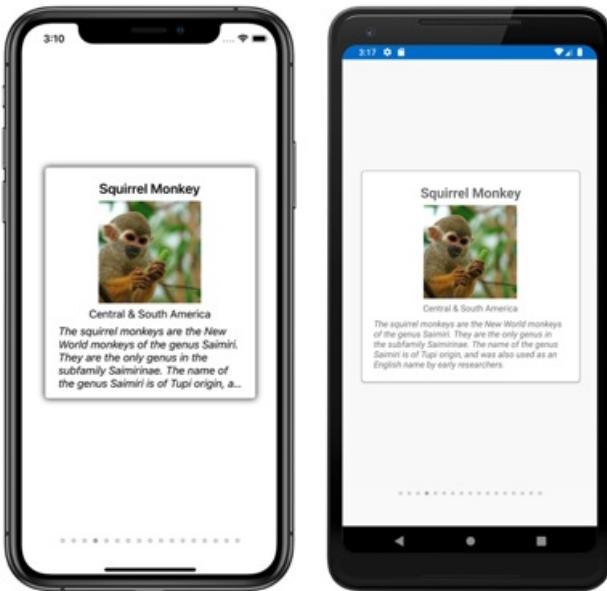
Xamarin.Forms IndicatorView

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

The `IndicatorView` is a control that displays indicators that represent the number of items, and current position, in a `CarouselView`:



`IndicatorView` defines the following properties:

- `Count`, of type `int`, the number of indicators.
- `HideSingle`, of type `bool`, indicates whether the indicator should be hidden when only one exists. The default value is `true`.
- `IndicatorColor`, of type `Color`, the color of the indicators.
- `IndicatorSize`, of type `double`, the size of the indicators. The default value is 6.0.
- `IndicatorLayout`, of type `Layout<View>`, defines the layout class used to render the `IndicatorView`. This property is set by Xamarin.Forms, and does not typically need to be set by developers.
- `IndicatorTemplate`, of type `DataTemplate`, the template that defines the appearance of each indicator.
- `IndicatorsShape`, of type `IndicatorShape`, the shape of each indicator.
- `ItemsSource`, of type `IEnumerable`, the collection that indicators will be displayed for. This property will automatically be set when the `CarouselView.IndicatorView` property is set.
- `MaximumVisible`, of type `int`, the maximum number of visible indicators. The default value is `int.MaxValue`.
- `Position`, of type `int`, the currently selected indicator index. This property uses a `TwoWay` binding. This property will automatically be set when the `CarouselView.IndicatorView` property is set.
- `SelectedIndicatorColor`, of type `Color`, the color of the indicator that represents the current item in the `CarouselView`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

Create an IndicatorView

The following example shows how to instantiate an `IndicatorView` in XAML:

```
<StackLayout>
    <CarouselView ItemsSource="{Binding Monkeys}"
                  IndicatorView="indicatorView">
        <CarouselView.ItemTemplate>
            <!-- DataTemplate that defines item appearance -->
        </CarouselView.ItemTemplate>
    </CarouselView>
    <IndicatorView x:Name="indicatorView"
                  IndicatorColor="LightGray"
                  SelectedIndicatorColor="DarkGray"
                  HorizontalOptions="Center" />
</StackLayout>
```

In this example, the `IndicatorView` is rendered beneath the `CarouselView`, with an indicator for each item in the `CarouselView`. The `IndicatorView` is populated with data by setting the `CarouselView.IndicatorView` property to the `IndicatorView` object. Each indicator is a light gray circle, while the indicator that represents the current item in the `CarouselView` is dark gray.

IMPORTANT

Setting the `CarouselView.IndicatorView` property results in the `IndicatorView.Position` property binding to the `CarouselView.Position` property, and the `IndicatorView.ItemsSource` property binding to the `CarouselView.ItemsSource` property.

Change indicator shape

The `IndicatorView` class has an `IndicatorsShape` property, which determines the shape of the indicators. This property can be set to one of the `IndicatorShape` enumeration members:

- `Circle` specifies that the indicator shapes will be circular. This is the default value of the `IndicatorView.IndicatorsShape` property.
- `Square` indicates that the indicator shapes will be square.

The following example shows an `IndicatorView` configured to use square indicators:

```
<IndicatorView x:Name="indicatorView"
               IndicatorsShape="Square"
               IndicatorColor="LightGray"
               SelectedIndicatorColor="DarkGray" />
```

Change indicator size

The `IndicatorView` class has an `IndicatorSize` property, of type `double`, which determines the size of the indicators in device-independent units. The default value of this property is 6.0.

The following example shows an `IndicatorView` configured to display larger indicators:

```
<IndicatorView x:Name="indicatorView"
               IndicatorSize="18" />
```

Limit the number of indicators displayed

The `IndicatorView` class has a `MaximumVisible` property, of type `int`, which determines the maximum number of visible indicators.

The following example shows an `IndicatorView` configured to display a maximum of six indicators:

```
<IndicatorView x:Name="indicatorView"
    MaximumVisible="6" />
```

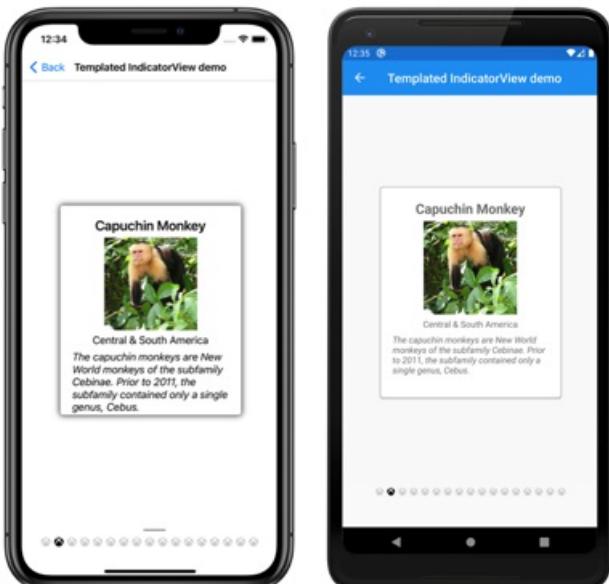
Define indicator appearance

The appearance of each indicator can be defined by setting the `IndicatorView.IndicatorTemplate` property to a `DataTemplate`:

```
<StackLayout>
    <CarouselView ItemsSource="{Binding Monkeys}"
        IndicatorView="indicatorView">
        <CarouselView.ItemTemplate>
            <!-- DataTemplate that defines item appearance -->
        </CarouselView.ItemTemplate>
    </CarouselView>
    <IndicatorView x:Name="indicatorView"
        Margin="0,0,0,40"
        IndicatorColor="Transparent"
        SelectedIndicatorColor="Transparent"
        HorizontalOptions="Center">
        <IndicatorView.IndicatorTemplate>
            <DataTemplate>
                <Label Text="" 
                    FontFamily="{OnPlatform iOS=Ionicons, Android=ionicons.ttf#}, Size=12" />
            </DataTemplate>
        </IndicatorView.IndicatorTemplate>
    </IndicatorView>
</StackLayout>
```

The elements specified in the `DataTemplate` define the appearance of each indicator. In this example, each indicator is a `Label` that displays a font icon.

The following screenshots show indicators rendered using a font icon:



Set visual states

`IndicatorView` has a `Selected` visual state that can be used to initiate a visual change to the indicator for the current position in the `IndicatorView`. A common use case for this `VisualStyle` is to change the color of the indicator that represents the current position:

```
<ContentPage ...>
    <ContentPage.Resources>
        <Style x:Key="IndicatorLabelStyle"
            TargetType="Label">
            <Setter Property="VisualStateManager.VisualStateGroups">
                <VisualStateGroupList>
                    <VisualStateGroup x:Name="CommonStates">
                        <VisualState x:Name="Normal">
                            <VisualState.Setters>
                                <Setter Property="TextColor"
                                    Value="LightGray" />
                            </VisualState.Setters>
                        </VisualState>
                        <VisualState x:Name="Selected">
                            <VisualState.Setters>
                                <Setter Property="TextColor"
                                    Value="Black" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
                </VisualStateGroupList>
            </Setter>
        </Style>
    </ContentPage.Resources>

    <StackLayout>
        ...
        <IndicatorView x:Name="indicatorView"
            Margin="0,0,0,40"
            IndicatorColor="Transparent"
            SelectedIndicatorColor="Transparent"
            HorizontalOptions="Center">
            <IndicatorView.IndicatorTemplate>
                <DataTemplate>
                    <Label Text="">
                        <FontFamily>{OnPlatform iOS=Ionicons, Android=ionicons.ttf#}</FontFamily>
                        <Size>12</Size>
                        <Style>{StaticResource IndicatorLabelStyle}</Style>
                    </Label>
                </DataTemplate>
            </IndicatorView.IndicatorTemplate>
        </IndicatorView>
    </StackLayout>
</ContentPage>
```

In this example, the `Selected` visual state specifies that the indicator that represents the current position will have its `TextColor` set to black. Otherwise the `TextColor` of the indicator will be light gray:



For more information about visual states, see [Xamarin.Forms Visual State Manager](#).

Related links

- [IndicatorView \(sample\)](#)
- [Xamarin.Forms CarouselView](#)
- [Xamarin.Forms Visual State Manager](#)

Xamarin.Forms ListView

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

`ListView` is a view for presenting lists of data, especially long lists that require scrolling.

IMPORTANT

`CollectionView` is a view for presenting lists of data using different layout specifications. It aims to provide a more flexible, and performant alternative to `ListView`. For more information, see [Xamarin.Forms CollectionView](#).

Use cases

A `ListView` control can be used in any situation where you're displaying scrollable lists of data. The `ListView` class supports context actions and data binding.

The `ListView` control shouldn't be confused with the `TableView` control. The `TableView` control is a better option whenever you have a non-bound list of options or data because it allows predefined options to be specified in XAML. For example, the iOS settings app, which has a mostly predefined set of options, is better suited to use a `TableView` than a `ListView`.

The `ListView` class doesn't support defining list items in XAML, you must use the `ItemsSource` property or data binding with an `ItemTemplate` to define items in the list.

A `ListView` is best suited for a collections consisting of a single data type. This requirement is because only one type of cell can be used for each row in the list. The `TableView` control can support multiple cell types, so it is a better option when you need to display multiple data types.

For more information about binding data to a `ListView` instance, see [ListView data sources](#).

Components

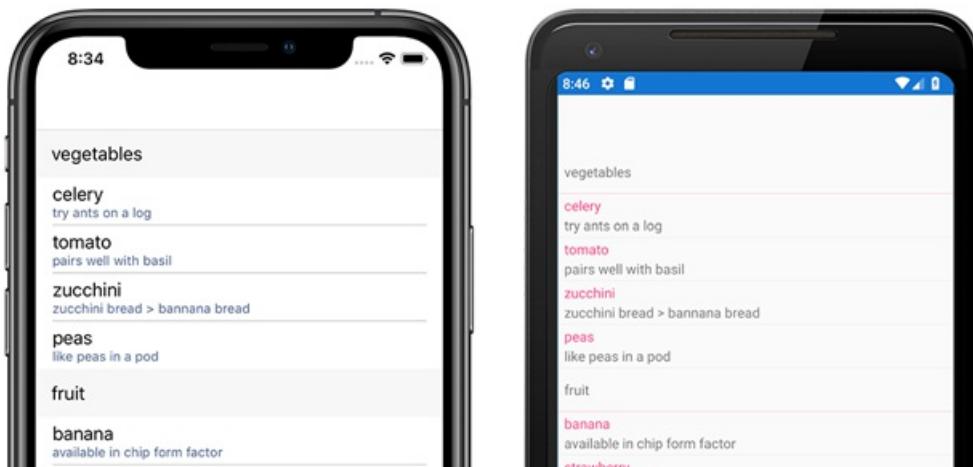
The `ListView` control has a number of components available to exercise the native functionality of each platform. These components are defined in the following sections.

Headers and footers

Header and footer components display at the beginning and end of a list, separate from list's data. Headers and footers can be bound to a separate data source from the ListView's data source.

Groups

Data in a `ListView` can be grouped for easier navigation. Groups are typically data bound. The following screenshot shows a `ListView` with grouped data:

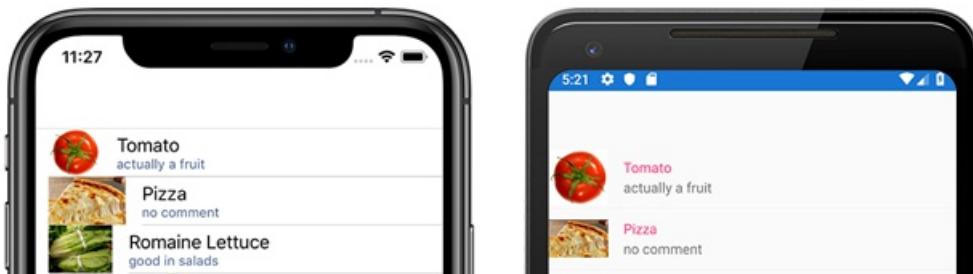


Cells

Data items in a `ListView` are called 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 cells**, such as the `TextCell` and `ImageCell`, correspond to native controls and are especially performant.
 - A `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.
 - An `ImageCell` displays an image with text. Appears as a `TextCell` with an image on the left.
- **Custom cells** are used to present complex data. For example, a custom cell could be used to present a list of songs that includes the album and artist.

The following screenshot shows a `ListView` with `ImageCell` items:



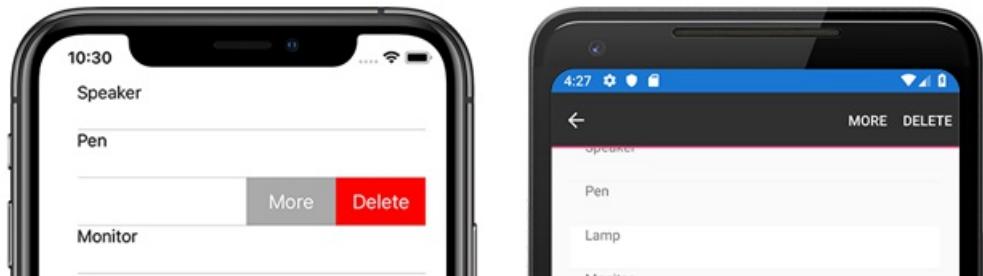
To learn more about customizing cells in a `ListView`, see [Customizing ListView Cell Appearance](#).

Functionality

The `ListView` class supports a number of interaction styles.

- **Pull-to-refresh** allows the user to pull the `ListView` down to refresh the contents.
- **Context actions** allow the developer to specify custom actions on individual list items. For example, you can implement swipe-to-action on iOS, or long-tap actions on Android.
- **Selection** allow the developer to attach functionality to selection and deselection events on list items.

The following screenshot shows a `ListView` with context actions:



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

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

A Xamarin.Forms `ListView` is used for displaying lists of data. This article explains how to populate a `ListView` with data and how to bind data to the selected item.

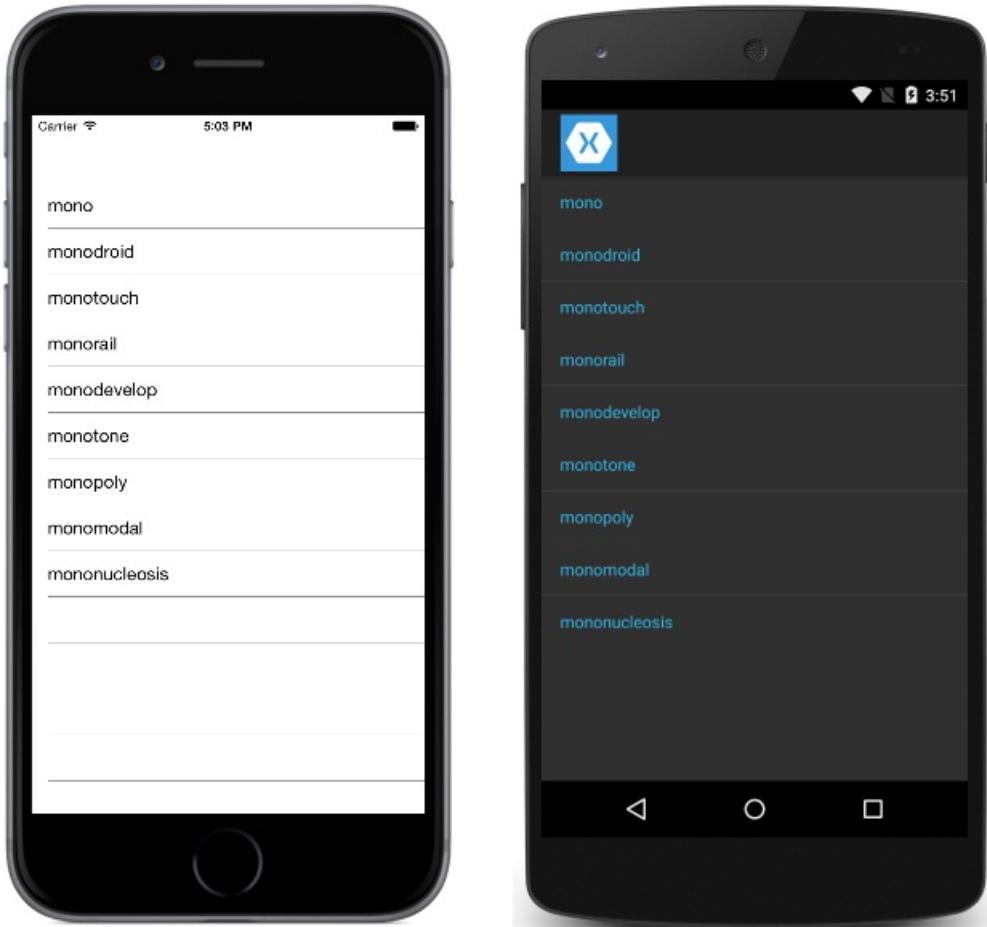
ItemsSource

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"
};
```



This 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<Employee> employees = new ObservableCollection<Employee>();
listView.ItemsSource = employees;

//Mr. Mono will be added to the ListView because it uses an ObservableCollection
employees.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;}
}
```

An `ObservableCollection<Employee>` is created, set as the `ListView ItemsSource`, and the list is populated with data:

```
ObservableCollection<Employee> employees = new ObservableCollection<Employee>();
public ObservableCollection<Employee> Employees { get { return employees; } }

public EmployeeListPage()
{
    EmployeeView.ItemsSource = employees;

    // ObservableCollection allows items to be added after ItemsSource
    // is set and the UI will react to changes
    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"});
}
```

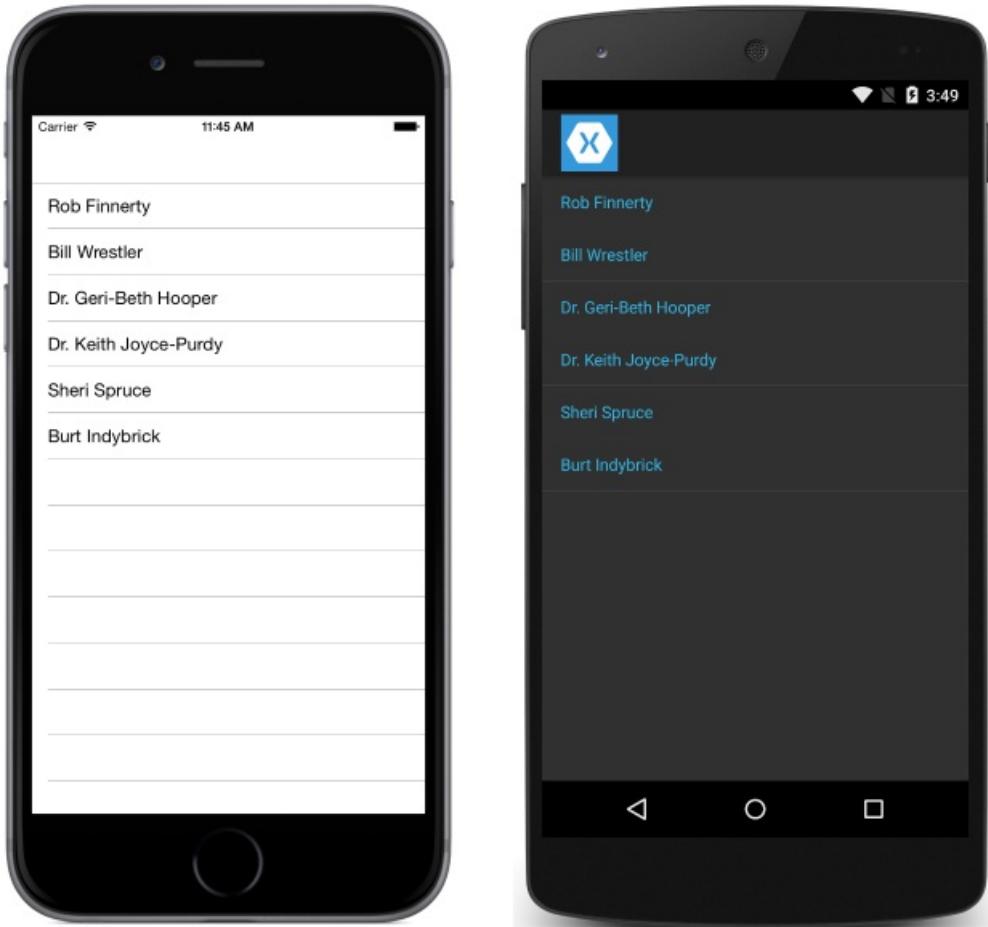
WARNING

While a `ListView` will update in response to changes in its underlying `ObservableCollection`, a `ListView` will not update if a different `ObservableCollection` instance is assigned to the original `ObservableCollection` reference (e.g. `employees = otherObservableCollection;`).

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"
        ItemsSource="{Binding Employees}">
        <ListView.ItemTemplate>
            <DataTemplate>
                <TextCell Text="{Binding DisplayName}" />
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>
```

This XAML example 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 results in the following screenshots:



WARNING

`ObservableCollection` is not thread safe. Modifying an `ObservableCollection` causes UI updates to happen on the same thread that performed the modifications. If the thread is not the primary UI thread, it will cause an exception.

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

8/4/2022 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `ListView` class is used to present scrollable lists, which can be customized through the use of `ViewCell` elements. A `ViewCell` element can display text and images, indicate a true/false state, and receive user input.

Built in Cells

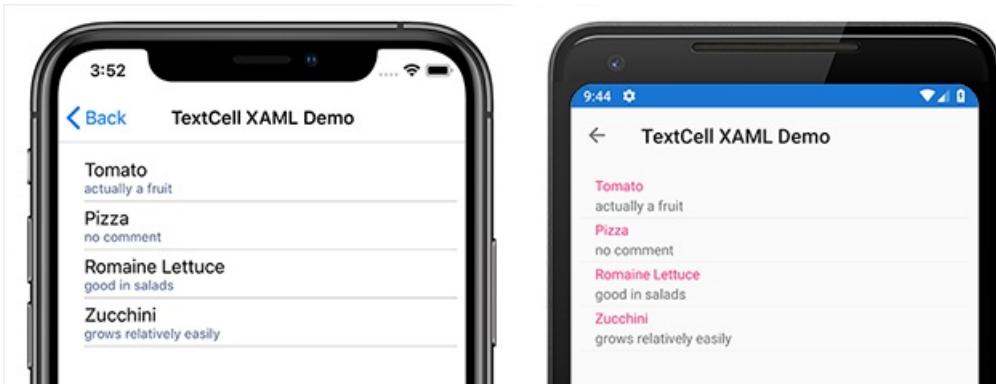
Xamarin.Forms comes with built-in cells that work for many applications:

- `TextCell` controls are used for displaying text with an optional second line for detail text.
- `ImageCell` controls are similar to `TextCell`s but include an image to the left of the text.
- `SwitchCell` controls are used to present and capture on/off or true/false states.
- `EntryCell` controls are used to present text data that the user can edit.

The `SwitchCell` and `EntryCell` controls are more commonly used in the context of a `TableView`.

TextCell

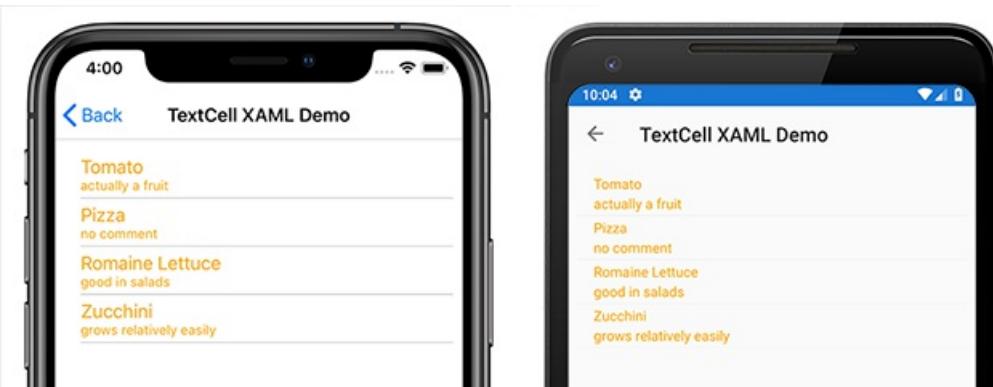
`TextCell` is a cell for displaying text, optionally with a second line as detail text. The following screenshot shows `TextCell` items on iOS and Android:



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 the following properties:

- `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

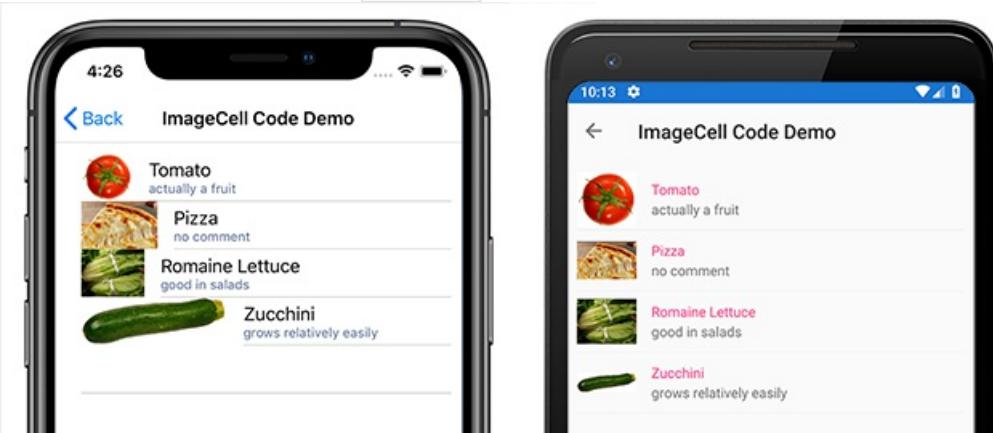
The following screenshot shows `TextCell` items with customized color properties:



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.

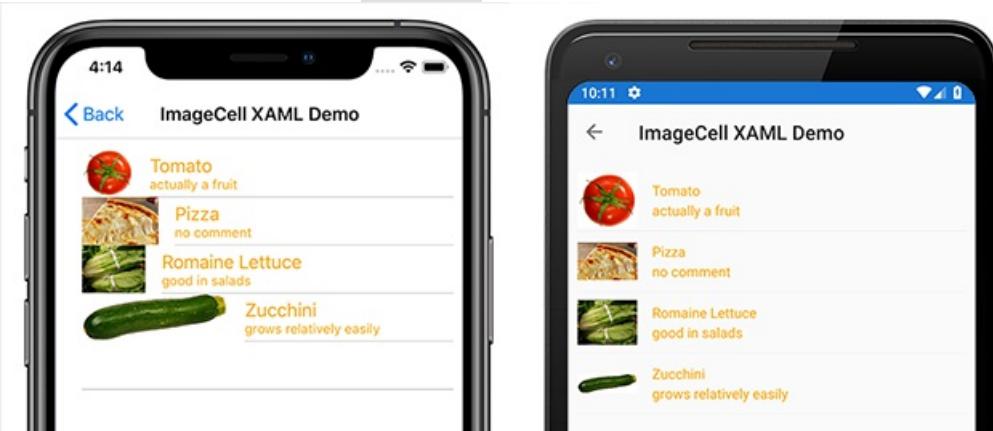
The following screenshot shows `ImageCell` items on iOS and Android:



`ImageCell` is useful when you need to display a list of data with a visual aspect, such as a list of contacts or movies. `ImageCell`s 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

The following screenshot shows `ImageCell` items with customized color properties:



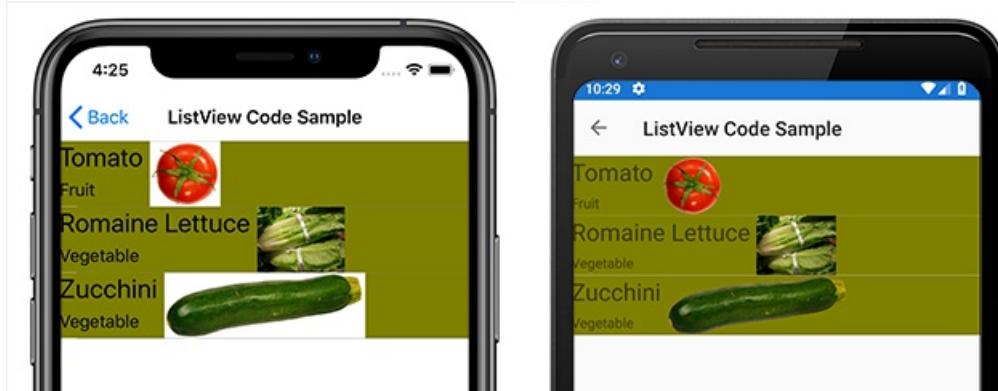
Custom Cells

Custom cells allow you to create cell layouts that aren't supported by the built-in cells. 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 offers a [caching behavior](#) on the `ListView` control which can improve scrolling performance for some types of custom cells.

The following screenshot shows an example of a custom cell:



XAML

The custom cell shown in the previous screenshot can be created 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="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 works as follows:

- The custom cell is nested inside a `DataTemplate`, which is inside `ListView.ItemTemplate`. This is the same process as using any built-in cell.
- `ViewCell` is the type of the custom cell. The child of the `DataTemplate` element must be of, or derive from, the `ViewCell` class.

- Inside the `ViewCell`, layout can be managed by any Xamarin.Forms layout. In this example, layout is managed by a `StackLayout`, which allows the background color to be customized.

NOTE

Any property of `StackLayout` that is bindable can be bound inside a custom cell. However, this capability is not shown in the XAML example.

Code

A custom cell can also be created in code. First, a custom class that derives from `ViewCell` must be created:

```
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 the page constructor, the `ListView`'s `ItemTemplate` property is set to a `DataTemplate` with the `CustomCell` type specified:

```
public partial class ImageCellPage : ContentPage
{
    public ImageCellPage ()
    {
        InitializeComponent ();
        listView.ItemTemplate = new DataTemplate (typeof(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 \(sample\)](#)

ListView appearance

8/4/2022 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `ListView` allows you to customize the presentation of the list, in addition to the `ViewCell` instances for each row in the list.

Grouping

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 ("Alpha", "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 method is possible because `PageTypeGroup` inherits from `List<PageModel>`.

Here is the XAML for displaying the grouped list:

```

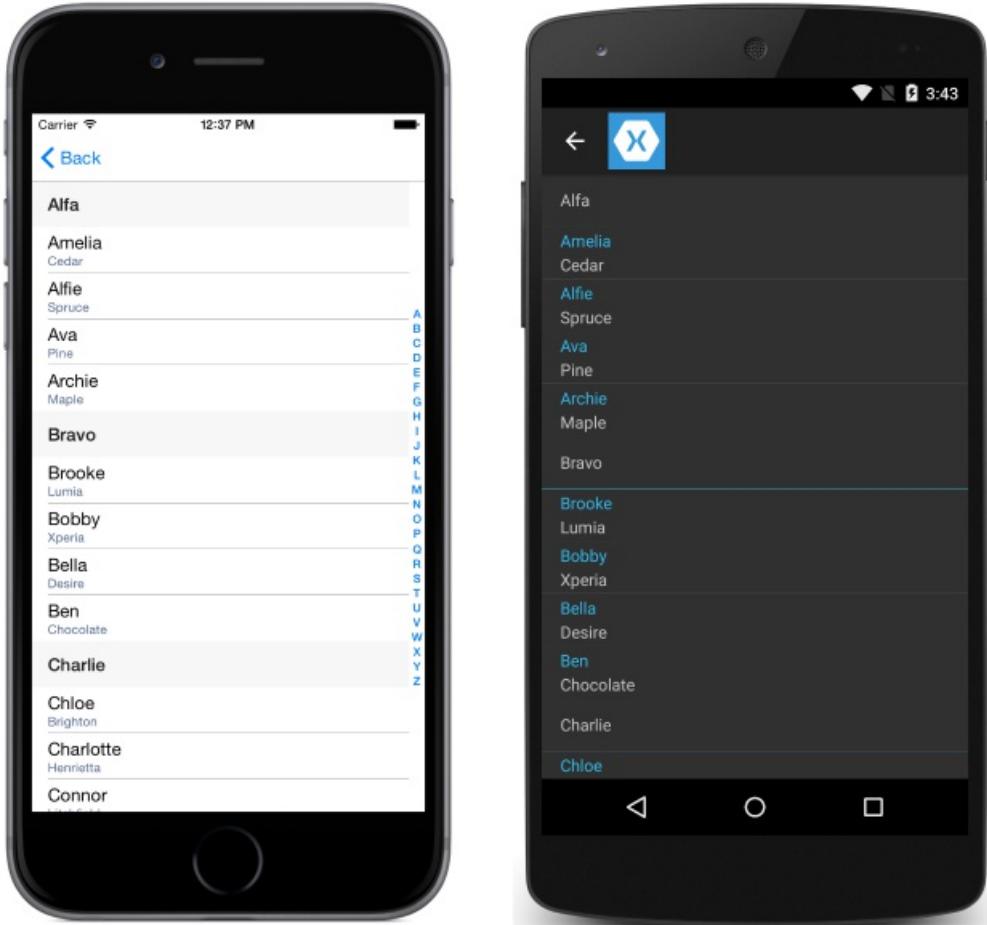
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DemoListView.GroupingViewPage"
<ContentPage.Content>
    <ListView x:Name="GroupedView"
        GroupDisplayBinding="{Binding Title}"
        GroupShortNameBinding="{Binding ShortName}"
        IsGroupingEnabled="true">
        <ListView.ItemTemplate>
            <DataTemplate>
                <TextCell Text="{Binding Title}"
                    Detail="{Binding Subtitle}" />
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage.Content>
</ContentPage>

```

This XAML performs the following actions:

- 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

This following screenshot shows the resulting UI:



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. This behavior is separate from [section groups](#).

You can set the `Header` and/or `Footer` to a `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 basic 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"
};

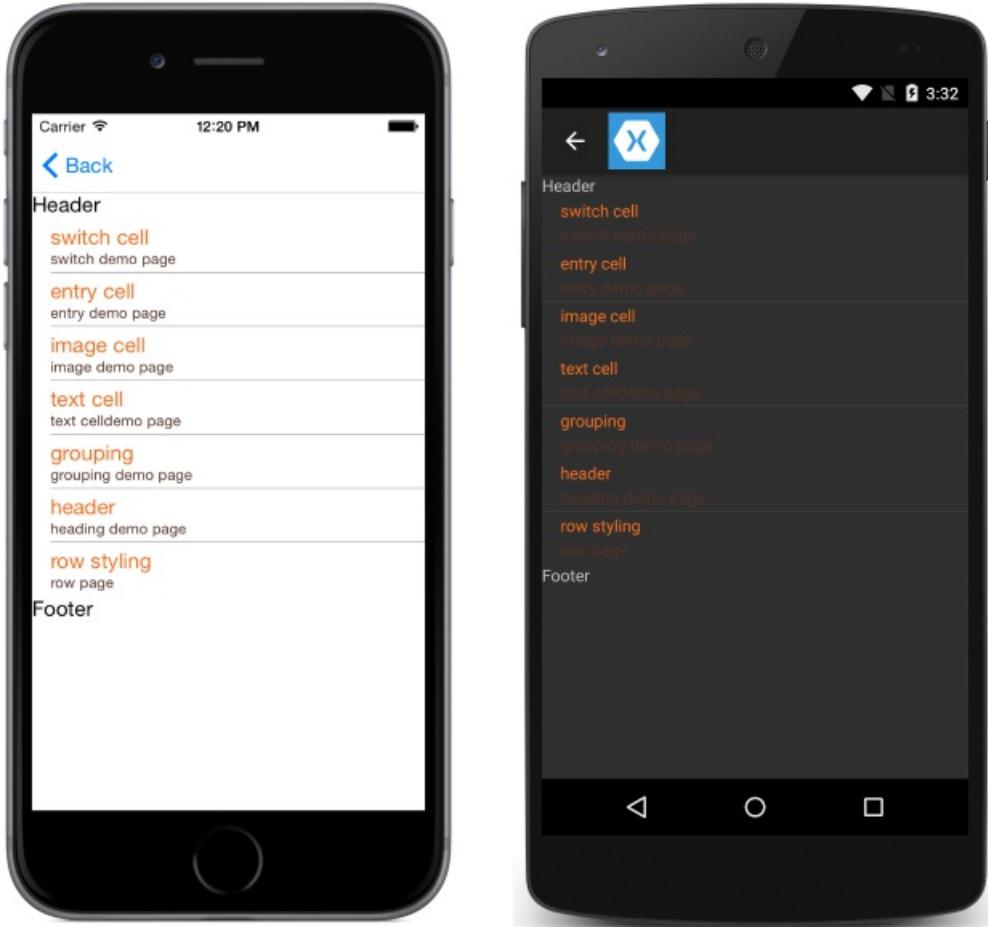
```

In XAML:

```

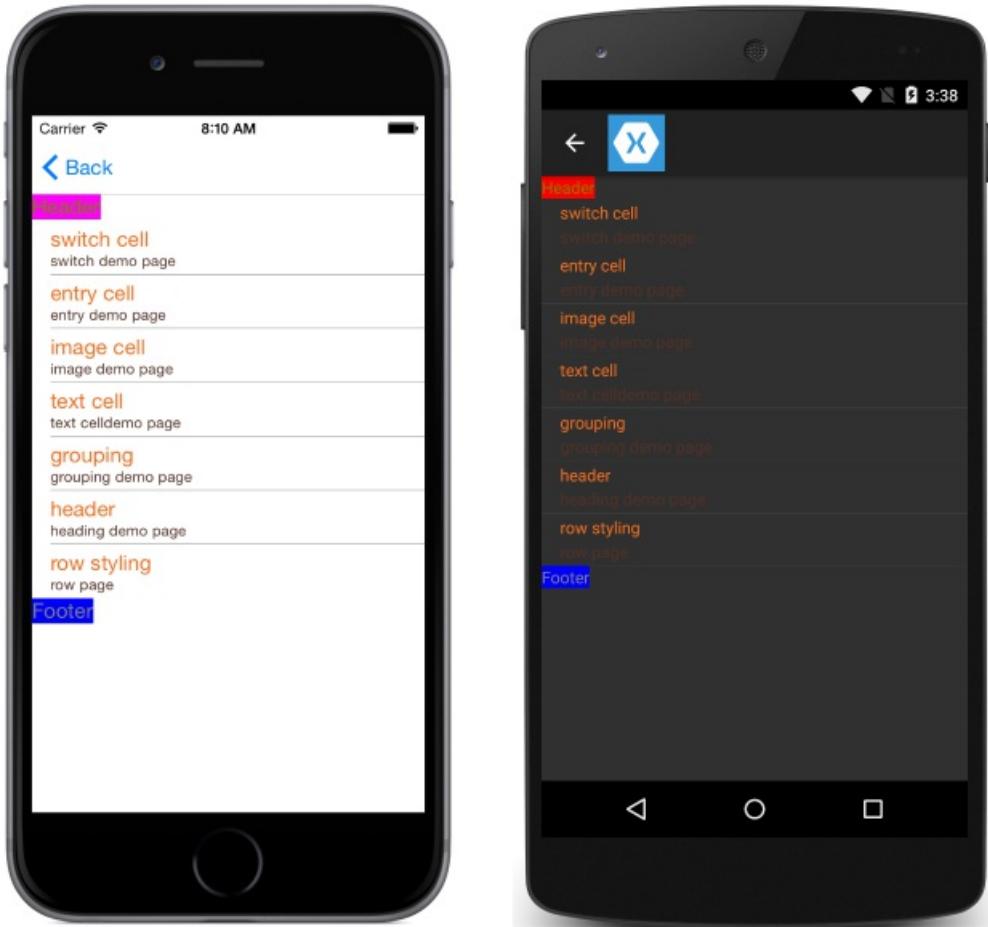
<ListView x:Name="HeaderList"
    Header="Header"
    Footer="Footer">
    ...
</ListView>

```



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>
```



Scrollbar visibility

The `ListView` class has `HorizontalScrollBarVisibility` and `VerticalScrollBarVisibility` properties, which get or set a `ScrollBarVisibility` value that represents when the horizontal, or vertical, scroll bar is visible. Both properties can be set to the following values:

- `Default` indicates the default scroll bar behavior for the platform, and is the default value for the `HorizontalScrollBarVisibility` and `VerticalScrollBarVisibility` properties.
- `Always` indicates that scroll bars will be visible, even when the content fits in the view.
- `Never` indicates that scroll bars will not be visible, even if the content doesn't fit in the view.

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.

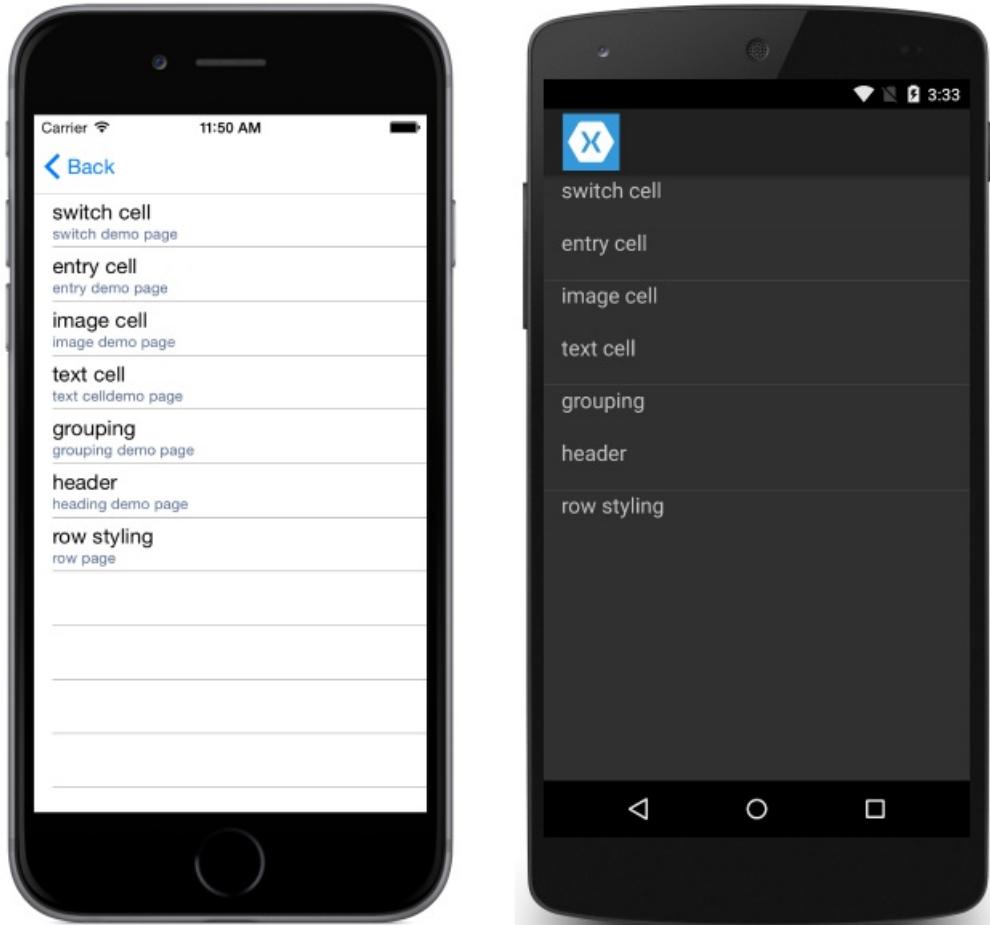
Default Visibility:

C#:

```
SeparatorDemoListView.SeparatorVisibility = SeparatorVisibility.Default;
```

XAML:

```
<ListView x:Name="SeparatorDemoListView" SeparatorVisibility="Default" />
```



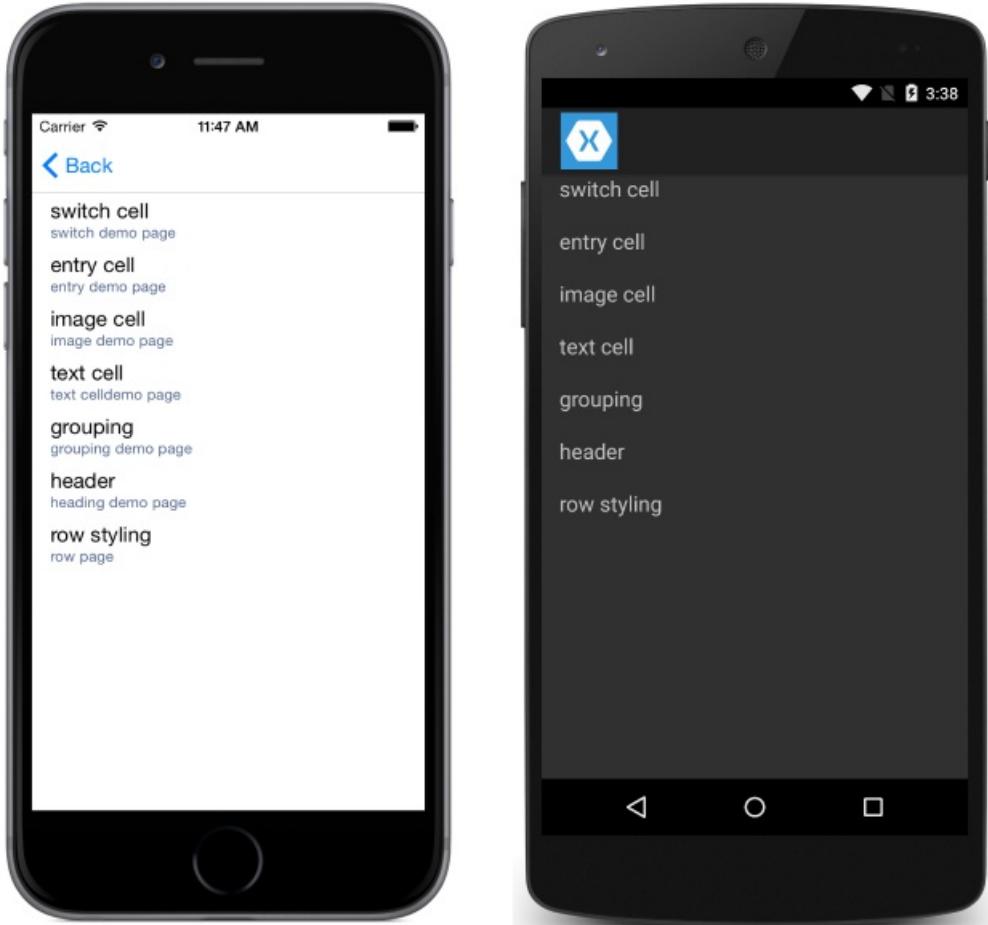
None:

C#:

```
SeparatorDemoListView.SeparatorVisibility = SeparatorVisibility.None;
```

XAML:

```
<ListView x:Name="SeparatorDemoListView" SeparatorVisibility="None" />
```



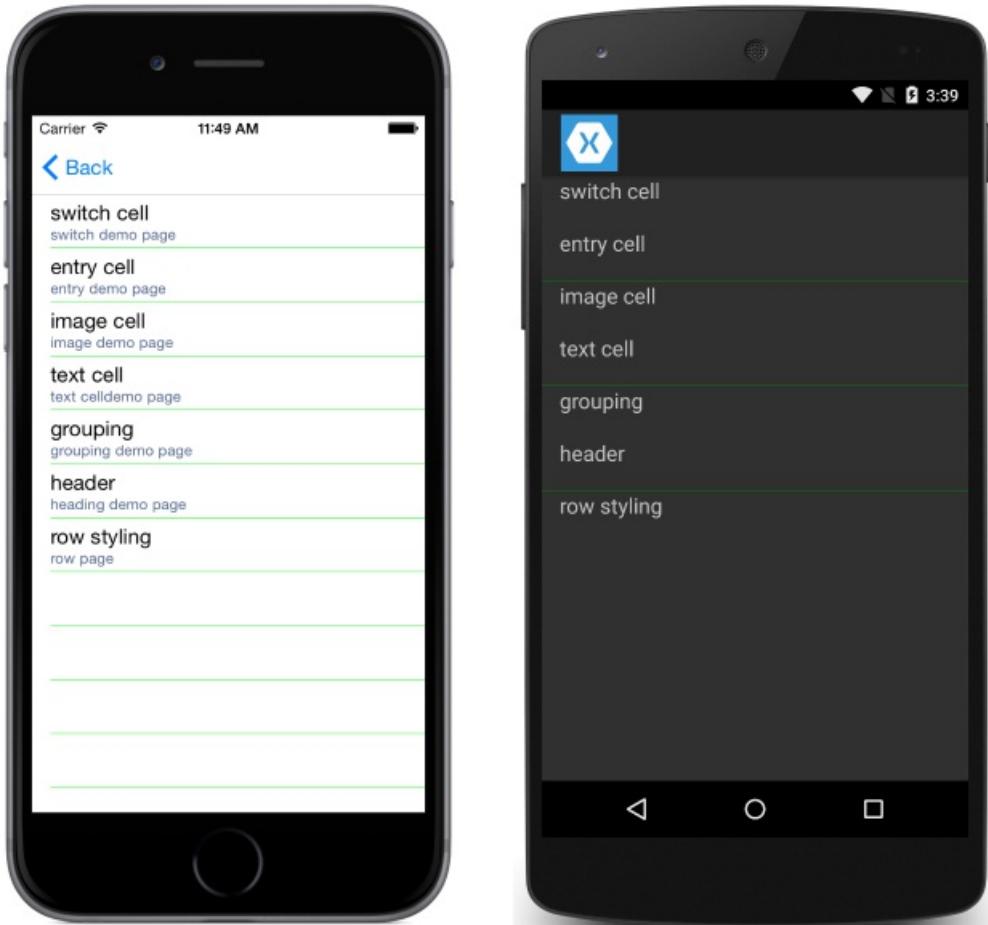
You can also set the color of the separator line via the `SeparatorColor` property:

C#:

```
SeparatorDemoListView.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 height

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`.

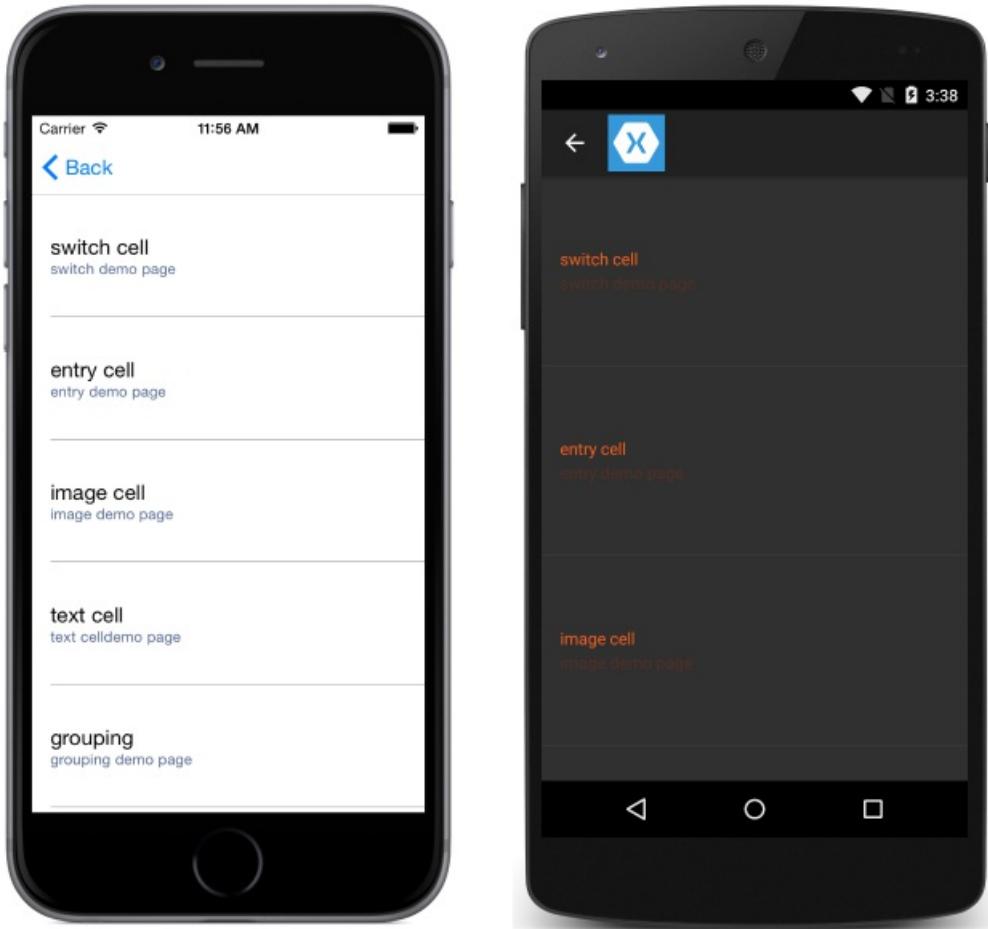
Custom fixed row height

C#:

```
RowHeightDemoListView.RowHeight = 100;
```

XAML:

```
<ListView x:Name="RowHeightDemoListView" RowHeight="100" />
```



Uneven rows

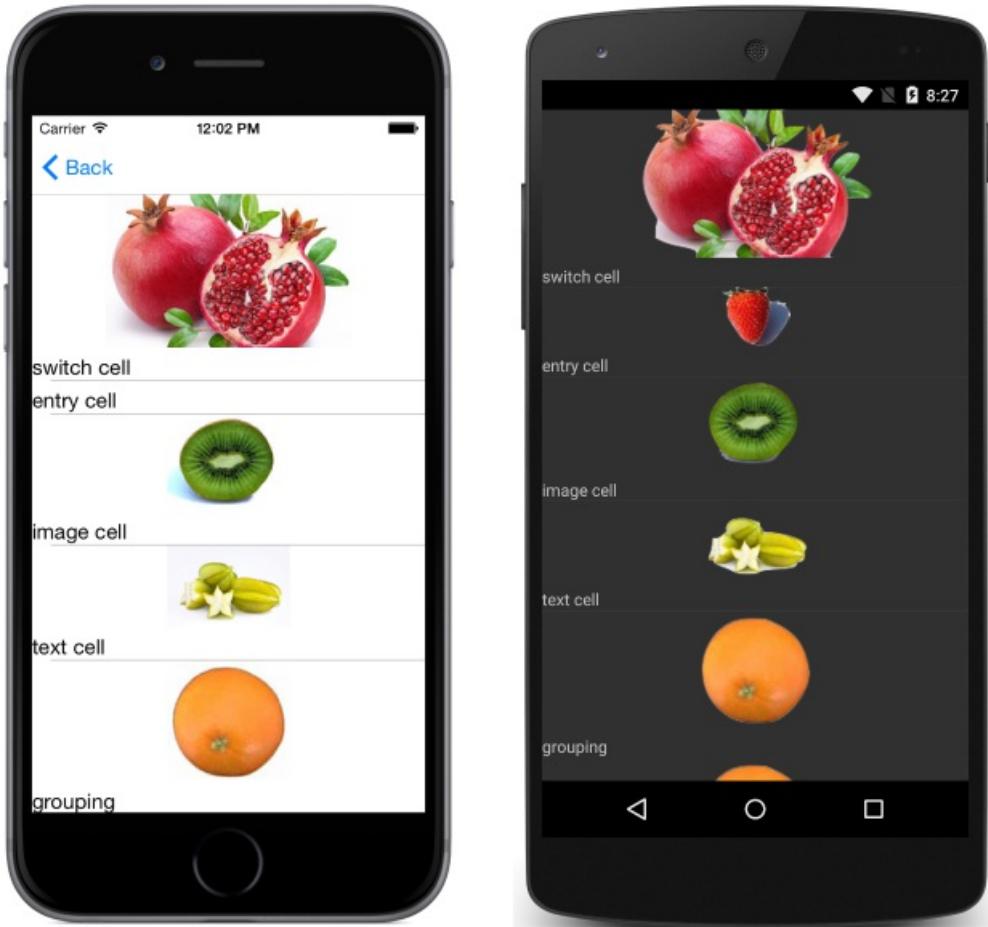
If you'd like individual rows to have different heights, you can set the `HasUnevenRows` property to `true`. Row heights don't have to be manually set once `HasUnevenRows` has been set to `true`, because the heights will be automatically calculated by Xamarin.Forms.

C#:

```
RowHeightDemoListView.HasUnevenRows = true;
```

XAML:

```
<ListView x:Name="RowHeightDemoListView" HasUnevenRows="true" />
```



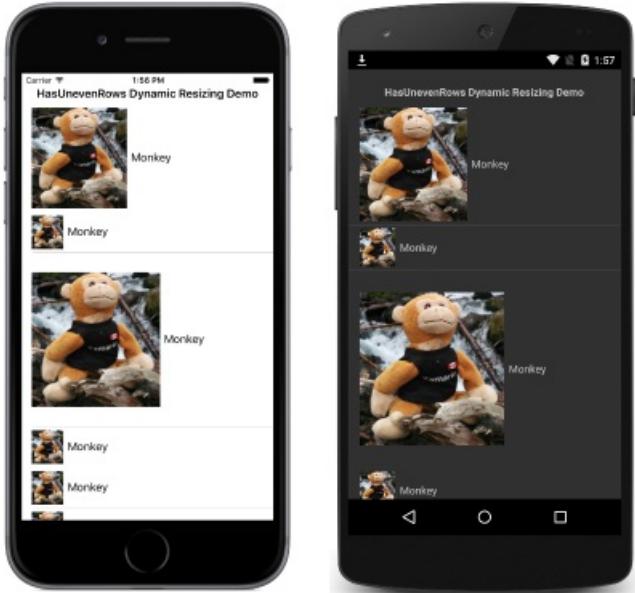
Resize rows at runtime

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.



WARNING

Overuse of runtime row resizing can cause performance degradation.

Related links

- [Grouping \(sample\)](#)
- [Custom Renderer View \(sample\)](#)
- [Dynamic Resizing of Rows \(sample\)](#)
- [1.4 release notes](#)
- [1.3 release notes](#)

ListView interactivity

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

The Xamarin.Forms `ListView` class supports user interaction with the data it presents.

Selection and 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.

Tapping the same item twice will fire two `ItemTapped` events, but will only fire a single `ItemSelected` event.

NOTE

The `ItemTappedEventArgs` class, which contains the event arguments for the `ItemTapped` event, has `Group` and `Item` properties, and an `ItemIndex` property whose value represents the index in the `ListView` of the tapped item. Similarly, the `SelectedItemChangedEventArgs` class, which contains the event arguments for the `ItemSelected` event, has a `SelectedItem` property, and a `SelectedItemIndex` property whose value represents the index in the `ListView` of the selected item.

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:



Disable selection

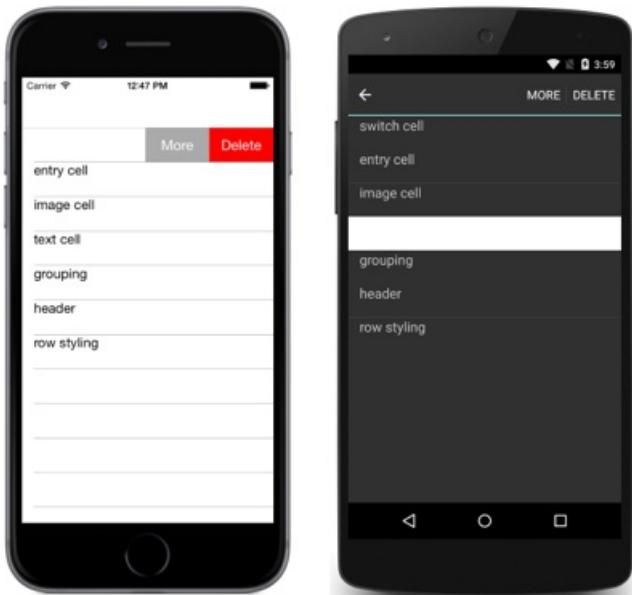
To disable `ListView` selection set the `SelectionMode` property to `None`:

```
<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` elements. Tap events for `MenuItem` objects 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. The `CommandParameter` property is available on `MenuItem` to store objects, such as the object behind the `MenuItem`'s `ViewCell`. The `CommandParameter` property can be set in both XAML and C#.

XAML

`MenuItem` elements can be created in a XAML collection. 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.

Code

Context actions can be implemented in any `cell` subclass (as long as it isn't being used as a group header) by creating `MenuItem` instances 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);
```

Pull to refresh

Users have come to expect that pulling down on a list of data will refresh that list. The `ListView` control supports this out-of-the-box. To enable pull-to-refresh functionality, set `IsPullToRefreshEnabled` to `true`:

```
<ListView ...
    IsPullToRefreshEnabled="true" />
```

The equivalent C# code is:

```
listView.IsPullToRefreshEnabled = true;
```

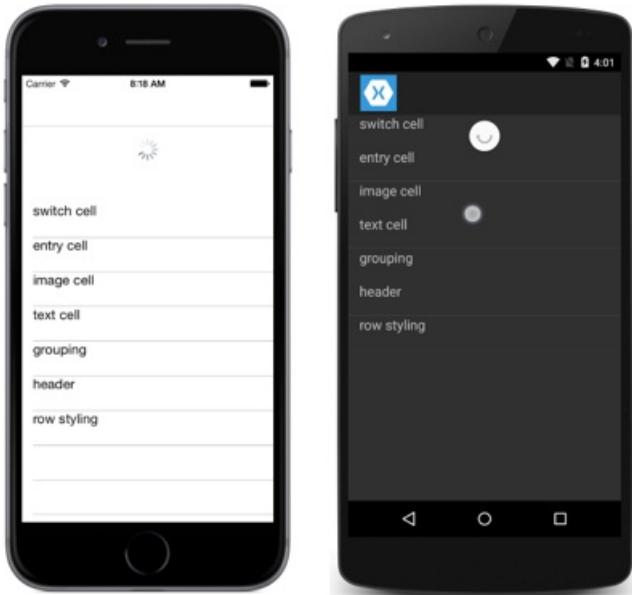
A spinner appears during the refresh, which is black by default. However, the spinner color can be changed on iOS and Android by setting the `RefreshControlColor` property to a `Color`:

```
<ListView ...
    IsPullToRefreshEnabled="true"
    RefreshControlColor="Red" />
```

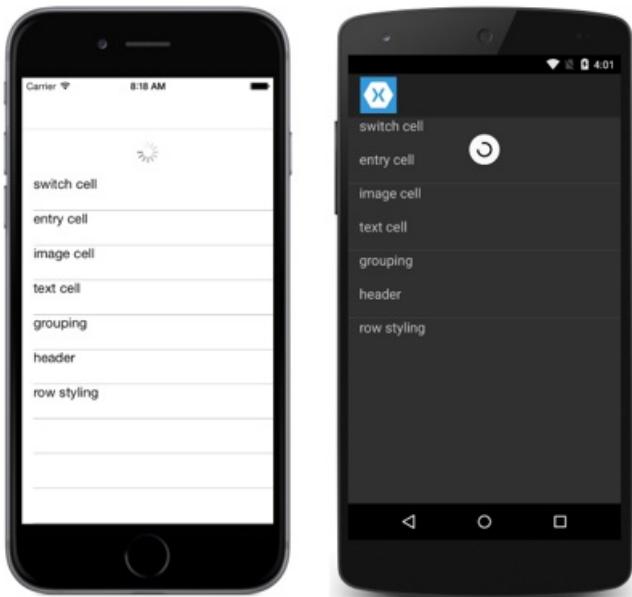
The equivalent C# code is:

```
listView.RefreshControlColor = Color.Red;
```

The following screenshots show pull-to-refresh as the user is pulling:



The following screenshots show pull-to-refresh after the user has released the pull, with the spinner being shown while the `ListView` is updating:



`ListView` fires the `Refreshing` event to initiate the refresh, and the `IsRefreshing` property will be set to `true`. Whatever code is required to refresh the contents of the `ListView` should then be executed by the event handler for the `Refreshing` event, or by the method executed by the `RefreshCommand`. Once the `ListView` is refreshed, the `IsRefreshing` property should be set to `false`, or the `EndRefresh` method should be called, to indicate that the refresh is complete.

NOTE

When defining a `RefreshCommand`, the `CanExecute` method of the command can be specified to enable or disable the command.

Detect scrolling

`ListView` defines a `Scrolled` event that's fired to indicate that scrolling occurred. The following XAML example shows a `ListView` that sets an event handler for the `Scrolled` event:

```
<ListView Scrolled="OnListViewScrolled">
...
</ListView>
```

The equivalent C# code is:

```
ListView listView = new ListView();
listView.Scrolled += OnListViewScrolled;
```

In this code example, the `OnListViewScrolled` event handler is executed when the `Scrolled` event fires:

```
void OnListViewScrolled(object sender, ScrolledEventArgs e)
{
    Debug.WriteLine("ScrollX: " + e.ScrollX);
    Debug.WriteLine("ScrollY: " + e.ScrollY);
}
```

The `onListViewScrolled` event handler outputs the values of the `scrolledEventArgs` object that accompanies the event.

Related links

- [ListView Interactivity \(sample\)](#)

ListView performance

8/4/2022 • 7 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

The Xamarin.Forms `ListView` is a powerful view for displaying data, but 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 complex measurement. Fortunately, there are techniques you can use to avoid poor performance.

Caching strategy

Lists are often used to display much more data than fits onscreen. For example, a music app might have a library of songs with thousands of entries. Creating an item for every entry would waste valuable memory and perform poorly. Creating and destroying rows constantly would require the application to instantiate and cleanup objects constantly, which would also perform poorly.

To conserve memory, the native `ListView` equivalents for each platform have built-in features for reusing rows. Only the cells visible on screen are loaded in memory and the **content** is loaded into existing cells. This pattern prevents the application from instantiating thousands of objects, saving time and memory.

Xamarin.Forms permits `ListView` cell reuse 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 be used in the following circumstances:

- Each cell has a large number of bindings (20-30+).
- The cell template changes frequently.
- 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're set up 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 doesn't always offer a performance improvement, and testing should be performed to determine any improvements. However, it's the preferred choice, and should be used in the following circumstances:

- Each cell has a small to moderate number of bindings.
- Each cell's [BindingContext](#) defines all of the cell data.
- 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 problem can be avoided 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`.

Set 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 XAML below:

```
<ListView CachingStrategy="RecycleElement">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                ...
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

This method has the same effect as setting the caching strategy argument in the constructor in C#.

Set 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's 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>

```

ListView performance suggestions

There are many techniques for improving the performance of a `ListView`. The following suggestions may improve the performance of your `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`. Changing layout incurs large measurement 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 specific, complex design presented in your cells.

`AbsoluteLayout` has the potential to perform layouts without a single measure call, making it highly performant. If `AbsoluteLayout` cannot be used, consider `RelativeLayout`. If using `RelativeLayout`, passing Constraints directly will be considerably faster than using the expression API. This method is faster 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 Picker

8/4/2022 • 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 the following properties:

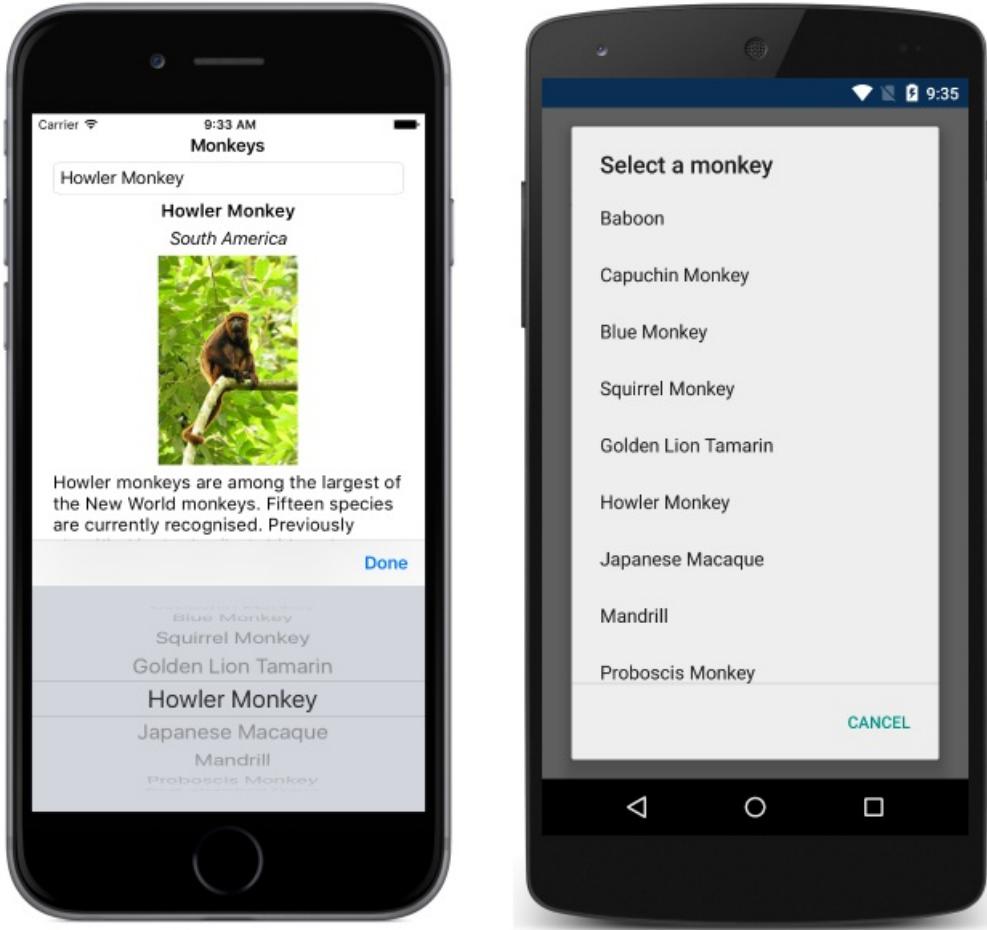
- `CharacterSpacing`, of type `double`, is the spacing between characters of the item displayed by the `Picker`.
- `FontAttributes` of type `FontAttributes`, which defaults to `FontAttributes.None`.
- `FontFamily` of type `string`, which defaults to `null`.
- `FontSize` of type `double`, which defaults to -1.0.
- `HorizontalTextAlignment`, of type `TextAlignment`, is the horizontal alignment of the text displayed by the `Picker`.
- `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`.
- `Title` of type `string`, which defaults to `null`.
- `TitleColor` of type `Color`, the color used to display the `Title` text.
- `VerticalTextAlignment`, of type `TextAlignment`, is the vertical alignment of the text displayed by the `Picker`.

All of the 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. 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

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

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"
        TitleColor="Red">
    <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", TitleColor = Color.Red };
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 be also 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"
        TitleColor="Red"
        ItemsSource="{Binding Monkeys}"
        ItemDisplayBinding="{Binding Name}" />
```

The equivalent C# code is shown below:

```
var picker = new Picker { Title = "Select a monkey", TitleColor = Color.Red };
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"
        TitleColor="Red"
        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", TitleColor = Color.Red };
picker.SetBinding(Picker.ItemsSourceProperty, "Monkeys");
picker.SetBinding(Picker.SelectedItemProperty, "SelectedMonkey");
picker.ItemDisplayBinding = new Binding("Name");

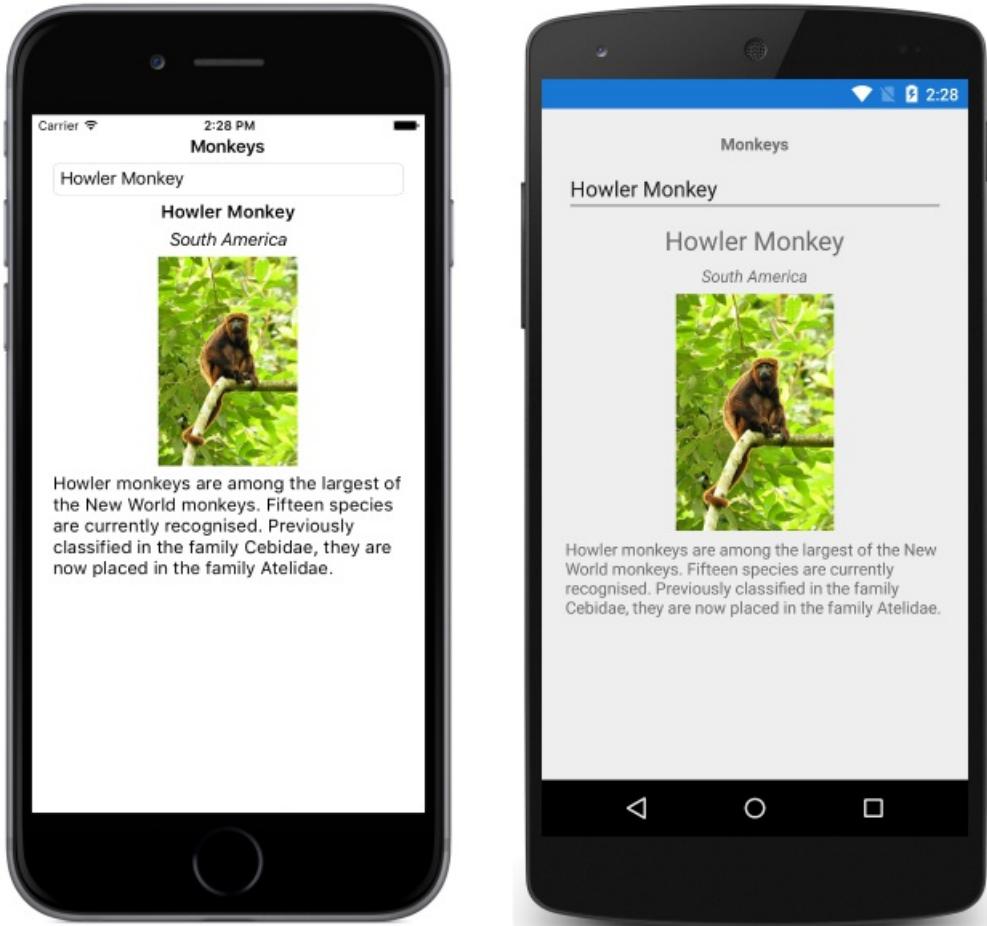
var nameLabel = new Label { ... };
nameLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Name");

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.

Related links

- [Monkey App \(sample\)](#)
- [Bindable Picker \(sample\)](#)
- [Picker API](#)

Adding Data to a Picker's Items Collection

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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"
        TitleColor="Red">
    <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", TitleColor = Color.Red };
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.

Related links

- [Picker Demo \(sample\)](#)
- [Picker](#)

Xamarin.Forms TableView

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

`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 manually added as children.



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.

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 `TableSection` instances. Each `TableSection` consists of a heading and one or more `ViewCell` instances:

```
<TableView Intent="Settings">
    <TableRoot>
        <TableSection Title="Ring">
            <SwitchCell Text="New Voice Mail" />
            <SwitchCell Text="New Mail" On="true" />
        </TableSection>
    </TableRoot>
</TableView>
```

The equivalent C# code is:

```
Content = new TableView
{
    Root = new TableRoot
    {
        new TableSection("Ring")
        {
            // TableSection constructor takes title as an optional parameter
            new SwitchCell { Text = "New Voice Mail" },
            new SwitchCell { Text = "New Mail", On = true }
        }
    },
    Intent = TableIntent.Settings
};
```

Appearance

`TableView` exposes the `Intent` property, which can be set to any of the `TableIntent` enumeration members:

- `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` value 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.

In addition, the color of the text displayed for each `TableSection` can be changed by setting the `TextColor` property to a `Color`.

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, `SwitchCell` and `EntryCell` are the most relevant for a `TableView` scenario.

See [ListView Cell Appearance](#) for a detailed description of `TextCell` and `ImageCell`.

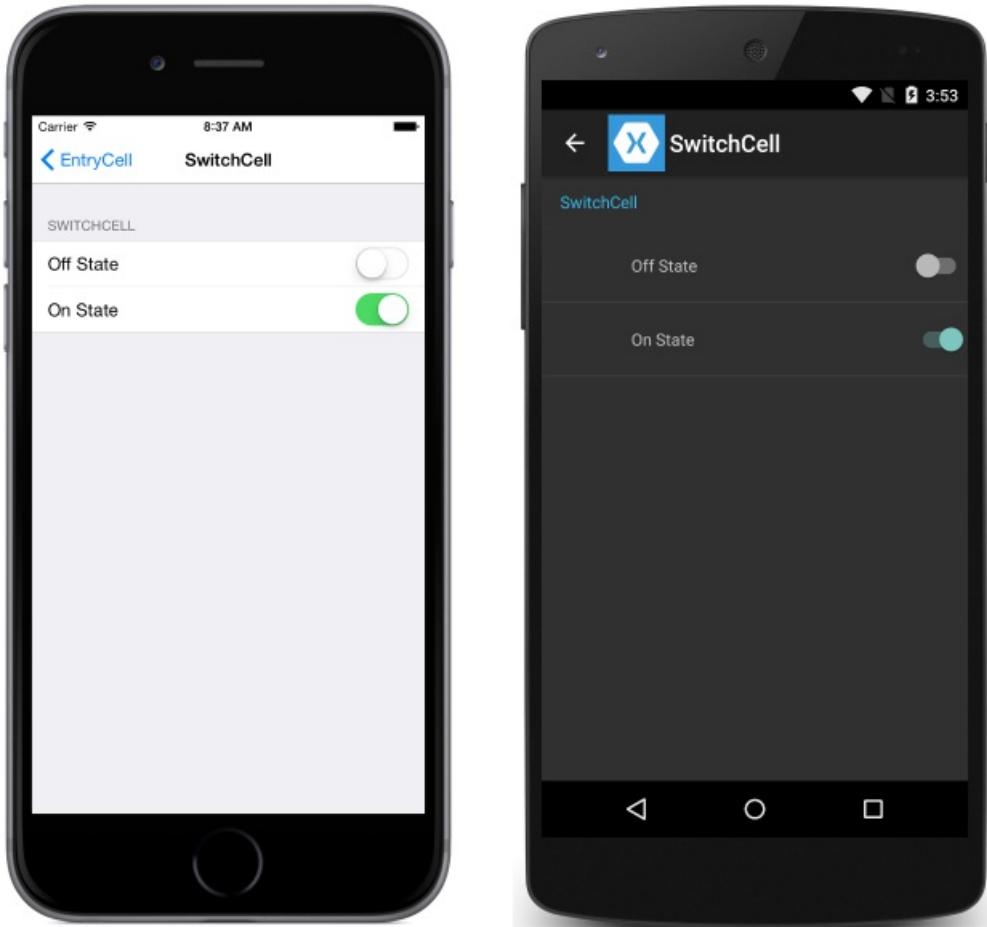
SwitchCell

`SwitchCell` is the control to use for presenting and capturing an on/off or `true` / `false` state. It defines the following properties:

- `Text` – text to display beside the switch.
- `On` – whether the switch is displayed as on or off.
- `OnColor` – the `Color` of the switch when it's in the on position.

All of these properties are bindable.

`SwitchCell` also exposes the `OnChanged` event, allowing you to respond to changes in the cell's state.

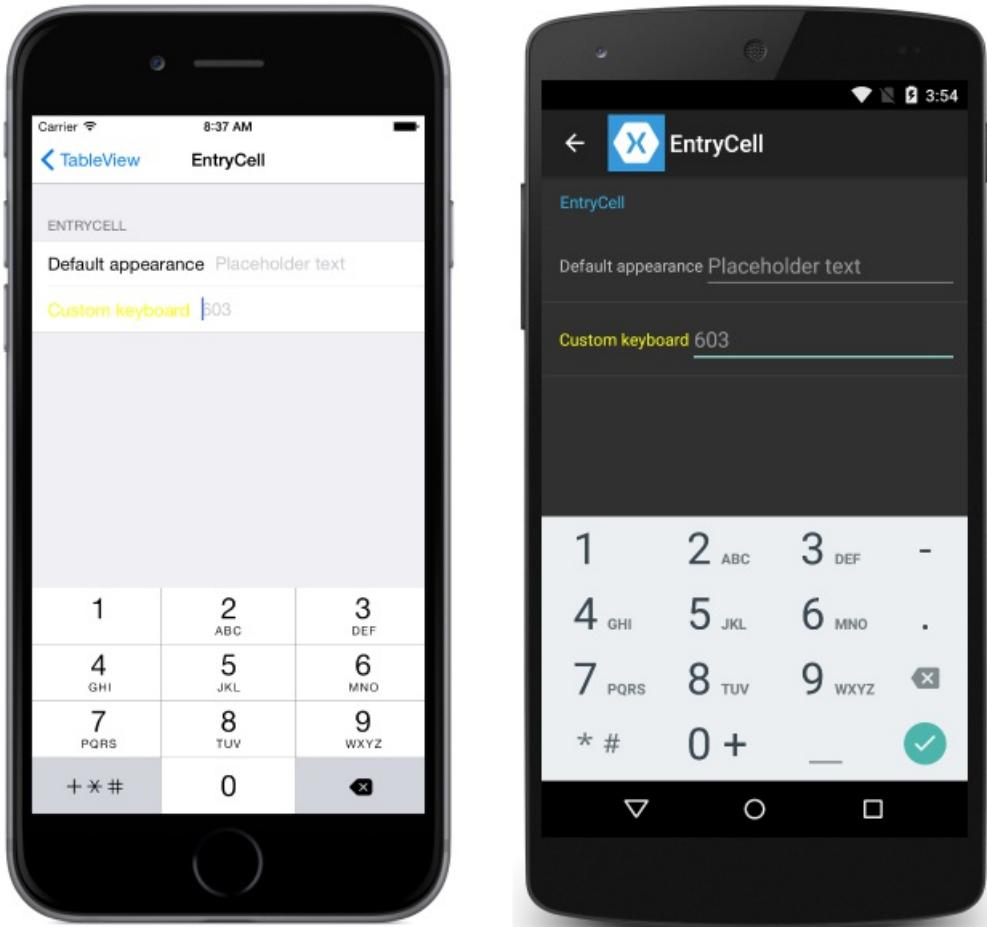


EntryCell

`EntryCell` is useful when you need to display text data that the user can edit. It defines the following properties:

- `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 left 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. Values are center, left, or right aligned. [See the API docs](#).
- `VerticalTextAlignment` – The vertical alignment of the text. Values are `Start`, `Center`, or `End`.

`EntryCell` also exposes the `Completed` event, which is fired when the user hits the 'done' button on the keyboard while editing text.

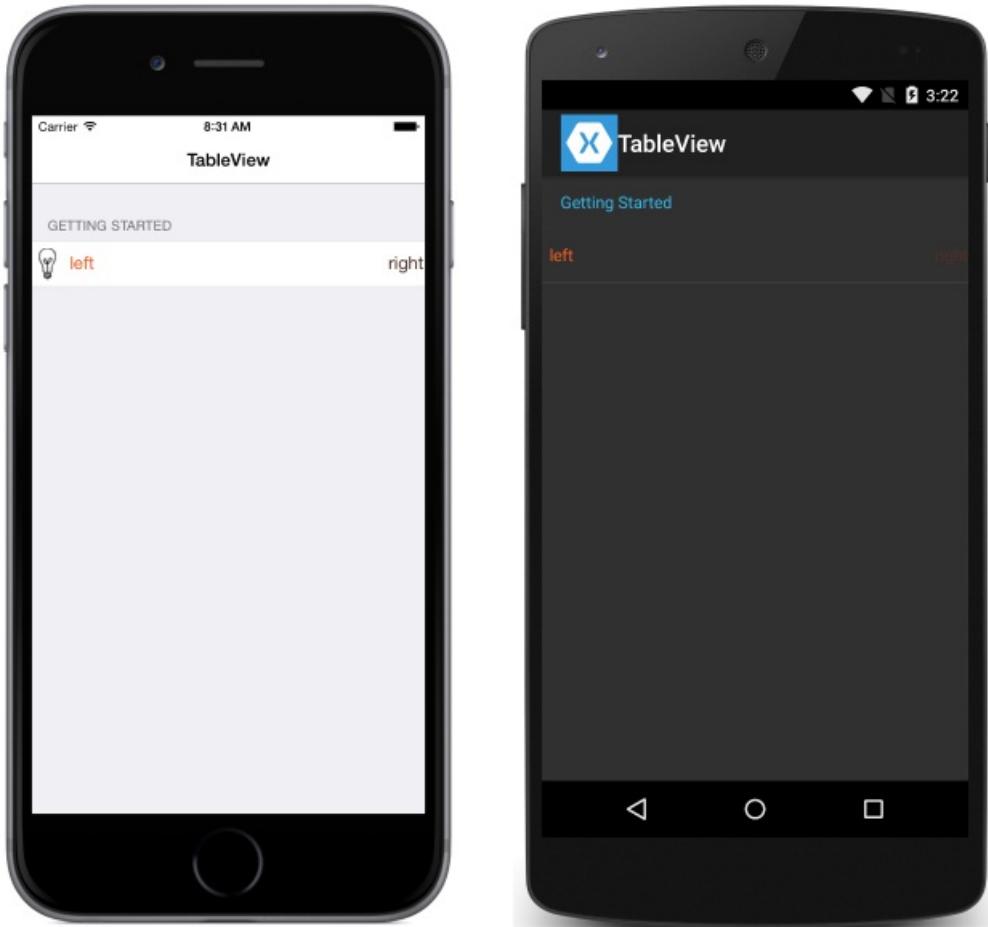


Custom cells

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:



The following example shows the XAML used to create the `TableView` in the screenshots above:

```
<?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">
    <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>
```

The equivalent C# code is:

```

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 root element under the `TableView` is the `TableRoot`, and there is a `TableSection` immediately underneath the `TableRoot`. The `ViewCell` is defined directly under the `TableSection`, and a `StackLayout` is used to manage the layout of the custom cell, although any layout could be used here.

NOTE

Unlike `ListView`, `TableView` does not require that custom (or any) cells are defined in an `ItemTemplate`.

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="OnTableViewCellTapped">
                    <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 `onTableViewCellTapped` event handler is executed:

```

void OnTableViewCellTapped(object sender, EventArgs e)
{
    _target.IsVisible = !_target.IsVisible;
    _viewCell.ForceUpdateSize();
}

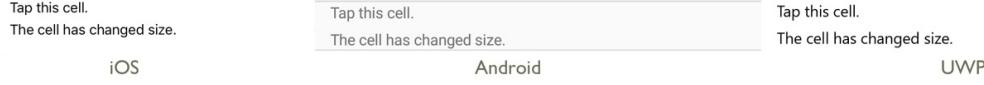
```

The `onTableViewCellTapped` 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 \(sample\)](#)

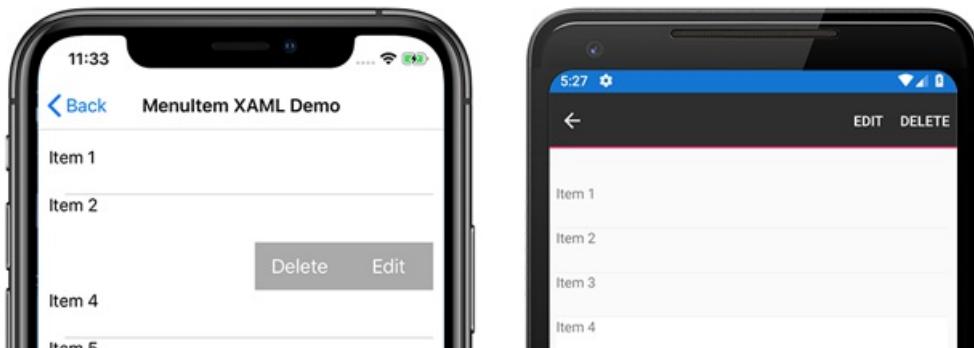
Xamarin.Forms MenuItem

8/4/2022 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `MenuItem` class defines menu items for menus such as `ListView` item context menus and Shell application flyout menus.

The following screenshots show `MenuItem` objects in a `ListView` context menu on iOS and Android:



The `MenuItem` class defines the following properties:

- `Command` is an `ICommand` that allows binding user actions, such as finger taps or clicks, to commands defined on a viewmodel.
- `CommandParameter` is an `object` that specifies the parameter that should be passed to the `Command`.
- `IconImageSource` is an `ImageSource` value that defines the display icon.
- `IsDestructive` is a `bool` value that indicates whether the `MenuItem` removes its associated UI element from the list.
- `IsEnabled` is a `bool` value that indicates whether this object responds to user input.
- `Text` is a `string` value that specifies the display text.

These properties are backed by `BindableProperty` objects so the `MenuItem` instance can be the target of data bindings.

Create a MenuItem

`MenuItem` objects can be used within a context menu on a `ListView` object's items. The most common pattern is to create `MenuItem` objects within a `ViewCell` instance, which is used as the `DataTemplate` object for the `ListView`'s `ItemTemplate`. When the `ListView` object is populated it will create each item using the `DataTemplate`, exposing the `MenuItem` choices when the context menu is activated for an item.

The following example shows `MenuItem` instantiation within the context of a `ListView` object:

```

<ListView>
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <ViewCell.ContextActions>
                    <MenuItem Text="Context Menu Option" />
                </ViewCell.ContextActions>
                <Label Text="{Binding .}" />
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>

```

A `MenuItem` can also be created in code:

```

// A function returns a ViewCell instance that
// is used as the template for each list item
DataTemplate dataTemplate = new DataTemplate(() =>
{
    // A Label displays the list item text
    Label label = new Label();
    label.SetBinding(Label.TextProperty, ".");

    // A ViewCell serves as the DataTemplate
    ViewCell viewCell = new ViewCell
    {
        View = label
    };

    // Add a MenuItem instance to the ContextActions
    MenuItem menuItem = new MenuItem
    {
        Text = "Context Menu Option"
    };
    viewCell.ContextActions.Add(menuItem);

    // The function returns the custom ViewCell
    // to the DataTemplate constructor
    return viewCell;
});

// Finally, the dataTemplate is provided to
// the ListView object
ListView listView = new ListView
{
    ...
    ItemTemplate = dataTemplate
};

```

Define MenuItem behavior with events

The `MenuItem` class exposes a `Clicked` event. An event handler can be attached to this event to react to taps or clicks on the `MenuItem` instance in XAML:

```

<MenuItem ...
    Clicked="OnItemClicked" />

```

An event handler can also be attached in code:

```
MenuItem item = new MenuItem { ... }
item.Clicked += OnItemClicked;
```

Previous examples referenced an `OnItemClicked` event handler. The following code shows an example implementation:

```
void OnItemClicked(object sender, EventArgs e)
{
    // The sender is the menuItem
    MenuItem menuItem = sender as MenuItem;

    // Access the list item through the BindingContext
    var contextItem = menuItem.BindingContext;

    // Do something with the contextItem here
}
```

Define MenuItem behavior with MVVM

The `MenuItem` class supports the Model-View-ViewModel (MVVM) pattern through `BindableProperty` objects and the `ICommand` interface. The following XAML shows `MenuItem` instances bound to commands defined on a viewmodel:

```
<ContentPage.BindingContext>
    <viewmodels:ListPageViewModel />
</ContentPage.BindingContext>

<StackLayout>
    <Label Text="{Binding Message}" ... />
    <ListView ItemsSource="{Binding Items}">
        <ListView.ItemTemplate>
            <DataTemplate>
                <ViewCell>
                    <ViewCell.ContextActions>
                        <MenuItem Text="Edit"
                            IconImageSource="icon.png"
                            Command="{Binding Source={x:Reference contentPage},
Path=BindingContext.EditCommand}"
                            CommandParameter="{Binding .}"/>
                        <MenuItem Text="Delete"
                            Command="{Binding Source={x:Reference contentPage},
Path=BindingContext.DeleteCommand}"
                            CommandParameter="{Binding .}"/>
                    </ViewCell.ContextActions>
                    <Label Text="{Binding .}" />
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</StackLayout>
```

In the previous example, two `MenuItem` objects are defined with their `Command` and `CommandParameter` properties bound to commands on the.viewmodel. The.viewmodel contains the commands referenced in the XAML:

```

public class ListPageViewModel : INotifyPropertyChanged
{
    ...
    public ICommand EditCommand => new Command<string>((string item) =>
    {
        Message = $"Edit command was called on: {item}";
    });

    public ICommand DeleteCommand => new Command<string>((string item) =>
    {
        Message = $"Delete command was called on: {item}";
    });
}

```

The sample application includes a `DataService` class used to get a list of items for populating the `ListView` objects. A.viewmodel is instantiated, with items from the `DataService` class, and set as the `BindingContext` in the code-behind:

```

public MenuItem.XamlMvvmPage()
{
    InitializeComponent();
    BindingContext = new ListPageViewModel(DataService.GetListItems());
}

```

MenuItem icons

WARNING

`MenuItem` objects only display icons on Android. On other platforms, only the text specified by the `Text` property will be displayed.

Icons are specified using the `IconImageSource` property. If an icon is specified, the text specified by the `Text` property will not be displayed. The following screenshot shows a `MenuItem` with an icon on Android:



For more information on using images in Xamarin.Forms, see [Images in Xamarin.Forms](#).

Enable or disable a MenuItem at runtime

To enable or disable a `MenuItem` at runtime, bind its `Command` property to an `ICommand` implementation, and ensure that a `canExecute` delegate enables and disables the `ICommand` as appropriate.

IMPORTANT

Do not bind the `IsEnabled` property to another property when using the `Command` property to enable or disable the `MenuItem`.

The following example shows a `MenuItem` whose `Command` property binds to an `ICommand` named `MyCommand`:

```
<MenuItem Text="My menu item"  
         Command="{Binding MyCommand}" />
```

The `ICommand` implementation requires a `canExecute` delegate that returns the value of a `bool` property to enable and disable the `MenuItem`:

```
public class MyViewModel : INotifyPropertyChanged  
{  
    bool isMenuItemEnabled = false;  
    public bool IsMenuItemEnabled  
    {  
        get { return isMenuItemEnabled; }  
        set  
        {  
            isMenuItemEnabled = value;  
            MyCommand.ChangeCanExecute();  
        }  
    }  
  
    public Command MyCommand { get; private set; }  
  
    public MyViewModel()  
    {  
        MyCommand = new Command(() =>  
        {  
            // Execute logic here  
        },  
        () => IsMenuItemEnabled);  
    }  
}
```

In this example, the `MenuItem` is disabled until the `IsMenuItemEnabled` property is set. When this occurs, the `Command.ChangeCanExecute` method is called which causes the `canExecute` delegate for `MyCommand` to be re-evaluated.

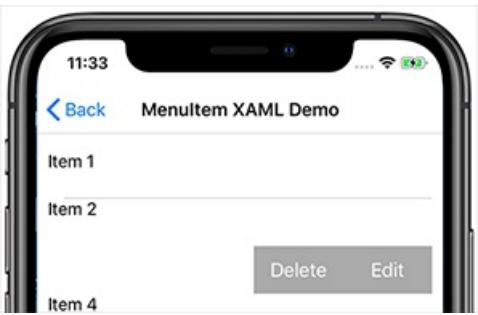
Cross-platform context menu behavior

Context menus are accessed and displayed differently on each platform.

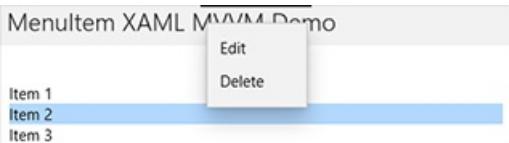
On Android, the context menu is activated by long-press on a list item. The context menu replaces the title and navigation bar area and `MenuItem` options are displayed as horizontal buttons.



On iOS, the context menu is activated by swiping on a list item. The context menu is displayed on the list item and `MenuItem`s are displayed as horizontal buttons.



On UWP, the context menu is activated by right-clicking on a list item. The context menu is displayed near the cursor as a vertical list.



Related links

- [MenuItem Demos](#)
- [Images in Xamarin.Forms](#)

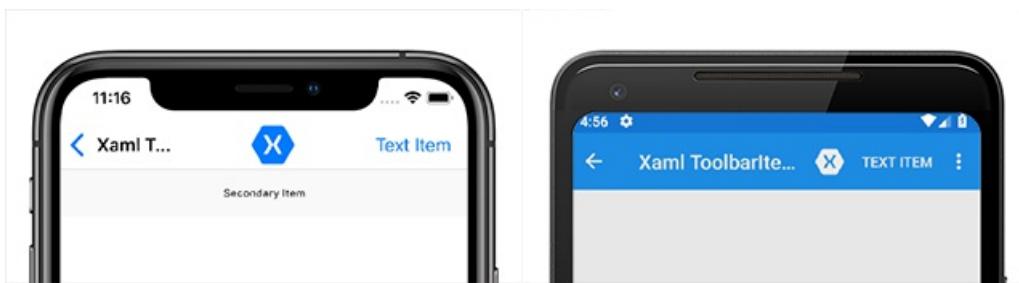
Xamarin.Forms ToolbarItem

8/4/2022 • 3 minutes to read • [Edit Online](#)

 [Download the sample](#)

The `Xamarin.Forms ToolbarItem` class is a special type of button that can be added to a `Page` object's `ToolbarItems` collection. Each `ToolbarItem` object will appear as a button in the application's navigation bar. A `ToolbarItem` instance can have an icon and appear as a primary or secondary menu item. The `ToolbarItem` class inherits from `MenuItem`.

The following screenshots show `ToolbarItem` objects in the navigation bar on iOS and Android:



The `ToolbarItem` class defines the following properties:

- `Order` is a `ToolbarItemOrder` enum value that determines whether the `ToolbarItem` instance displays in the primary or secondary menu.
- `Priority` is an `integer` value that determines the display order of items in a `Page` object's `ToolbarItems` collection.

The `ToolbarItem` class inherits the following typically-used properties from the `MenuItem` class:

- `Command` is an `ICommand` that allows binding user actions, such as finger taps or clicks, to commands defined on a viewmodel.
- `CommandParameter` is an `object` that specifies the parameter that should be passed to the `Command`.
- `IconImageSource` is an `ImageSource` value that determines the display icon on a `ToolbarItem` object.
- `Text` is a `string` that determines the display text on a `ToolbarItem` object.

These properties are backed by `BindableProperty` objects so a `ToolbarItem` instance can be the target of data bindings.

NOTE

An alternative to creating a toolbar from `ToolbarItem` objects is to set the `NavigationPage.TitleView` attached property to a layout class that contains multiple views. For more information, see [Displaying Views in the Navigation Bar](#).

Create a ToolbarItem

A `ToolbarItem` object can be instantiated in XAML. The `Text` and `IconImageSource` properties can be set to determine how the button is displayed in the navigation bar. The following example shows how to instantiate a `ToolbarItem` with some common properties set, and add it to a `ContentPage`'s `ToolbarItems` collection:

```
<ContentPage.ToolbarItems>
    <ToolbarItem Text="Example Item"
        IconImageSource="example_icon.png"
        Order="Primary"
        Priority="0" />
</ContentPage.ToolbarItems>
```

This example will result in a `ToolbarItem` object that has text, an icon and appears first in the primary navigation bar area. A `ToolbarItem` can also be created in code and added to the `ToolbarItems` collection:

```
ToolbarItem item = new ToolbarItem
{
    Text = "Example Item",
    IconImageSource = ImageSource.FromFile("example_icon.png"),
    Order = ToolbarItemOrder.Primary,
    Priority = 0
};

// "this" refers to a Page object
this.ToolbarItems.Add(item);
```

The file represented by the `string`, provided as the `IconImageSource` property, must exist in each platform project.

NOTE

Image assets are handled differently on each platform. An `ImageSource` can come from sources including a local file or embedded resource, a URI, or a stream. For more information about setting the `IconImageSource` property and Images in Xamarin.Forms, see [Images in Xamarin.Forms](#).

Define button behavior

The `ToolbarItem` class inherits the `Clicked` event from the `MenuItem` class. An event handler can be attached to the `Clicked` event to react to taps or clicks on `ToolbarItem` instances in XAML:

```
<ToolbarItem ...
    Clicked="OnItemClicked" />
```

An event handler can also be attached in code:

```
ToolbarItem item = new ToolbarItem { ... }
item.Clicked += OnItemClicked;
```

Previous examples referenced an `OnItemClicked` event handler. The following code shows an example implementation:

```
void OnItemClicked(object sender, EventArgs e)
{
    ToolbarItem item = (ToolbarItem)sender;
    messageLabel.Text = $"You clicked the \"{item.Text}\" toolbar item.";
}
```

`ToolbarItem` objects can also use the `Command` and `CommandParameter` properties to react to user input without event handlers. For more information about the `ICommand` interface and MVVM data-binding, see

Enable or disable a ToolbarItem at runtime

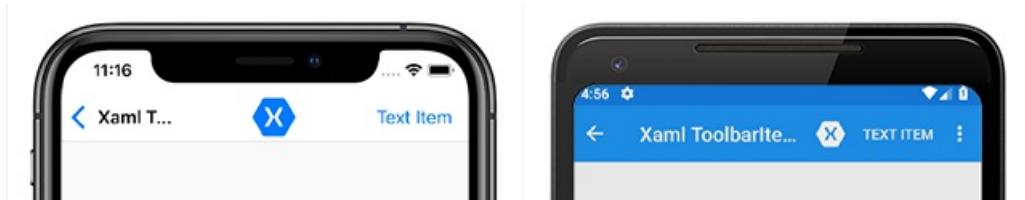
To enable or disable a `ToolbarItem` at runtime, bind its `Command` property to an `ICommand` implementation, and ensure that a `canExecute` delegate enables and disables the `ICommand` as appropriate.

For more information, see [Enable or disable a MenuItem at runtime](#).

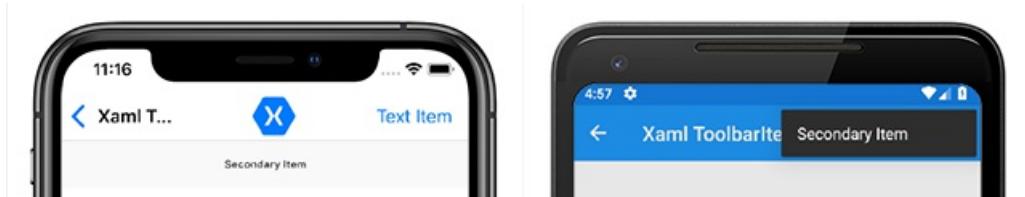
Primary and secondary menus

The `ToolbarItemOrder` enum has `Default`, `Primary`, and `Secondary` values.

When the `Order` property is set to `Primary`, the `ToolbarItem` object will appear in the main navigation bar on all platforms. `ToolbarItem` objects are prioritized over the page title, which will be truncated to make room for the items. The following screenshots show `ToolbarItem` objects in the primary menu on iOS and Android:



When the `Order` property is set to `Secondary`, behavior varies across platforms. On UWP and Android, the `Secondary` items menu appears as three dots that can be tapped or clicked to reveal items in a vertical list. On iOS, the `Secondary` items menu appears below the navigation bar as a horizontal list. The following screenshots show a secondary menu on iOS and Android:



WARNING

Icon behavior in `ToolbarItem` objects that have their `Order` property set to `Secondary` is inconsistent across platforms. Avoid setting the `IconImageSource` property on items that appear in the secondary menu.

Related links

- [ToolbarItem Demos](#)
- [Images in Xamarin.Forms](#)
- [Xamarin.Forms MenuItem](#)

Animation in Xamarin.Forms

8/4/2022 • 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

8/4/2022 • 9 minutes to read • [Edit Online](#)



[Download the sample](#)

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:

- `CancelAnimations` cancels any animations.
- `FadeTo` animates the `Opacity` 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`.
- `ScaleTo` animates the `Scale` property of a `VisualElement`.
- `ScaleXTo` animates the `ScaleX` property of a `VisualElement`.
- `ScaleYTo` animates the `ScaleY` property of a `VisualElement`.
- `TranslateTo` animates the `TranslationX` and `TranslationY` properties of a `VisualElement`.

By default, each animation will take 250 milliseconds. However, a duration for each animation can be specified when creating the animation.

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.

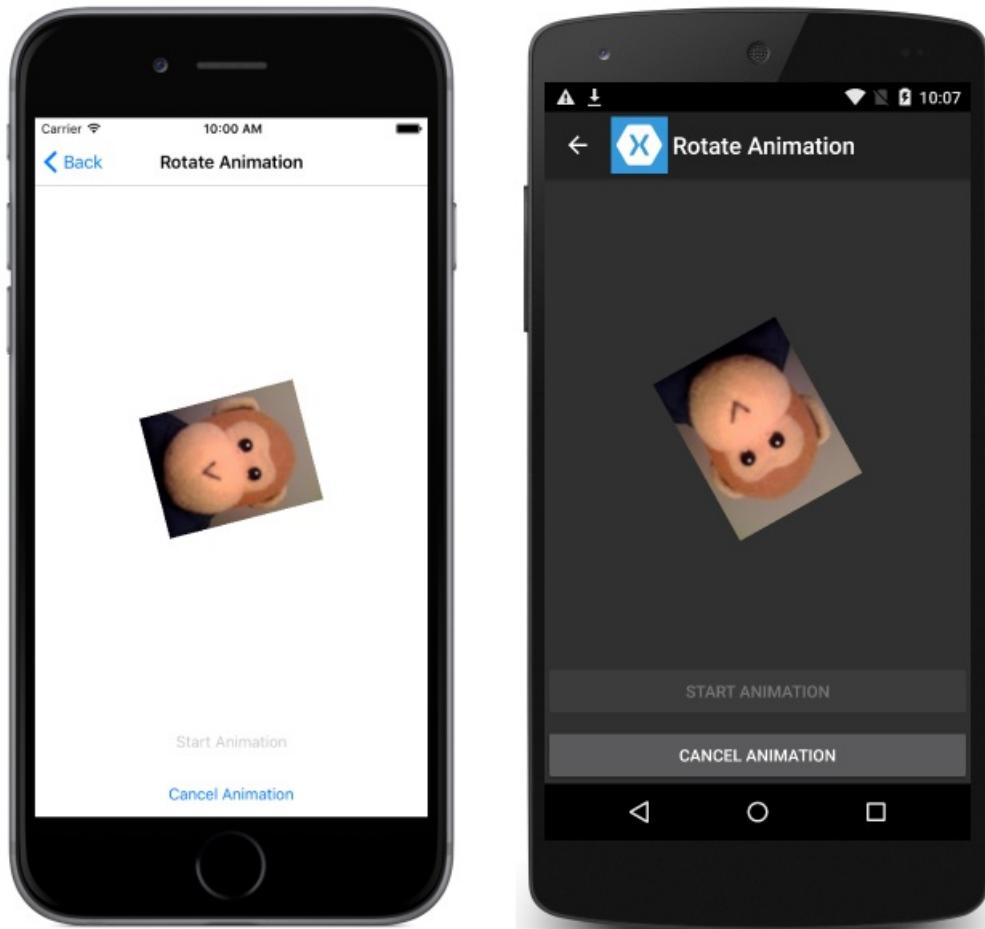
Rotation

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:



NOTE

In addition to the [RotateTo](#) method, there are also [RotateXTo](#) and [RotateYTo](#) methods that animate the [RotationX](#) and [RotationY](#) properties, respectively.

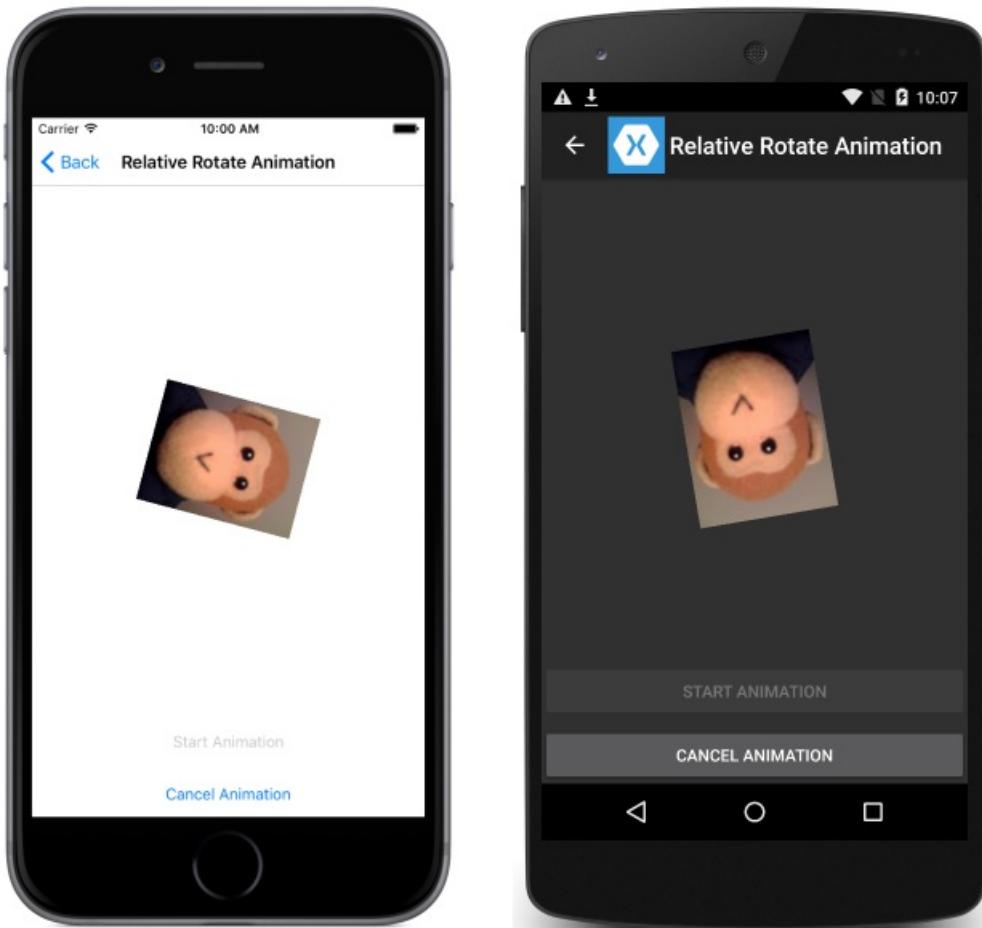
Relative Rotation

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:



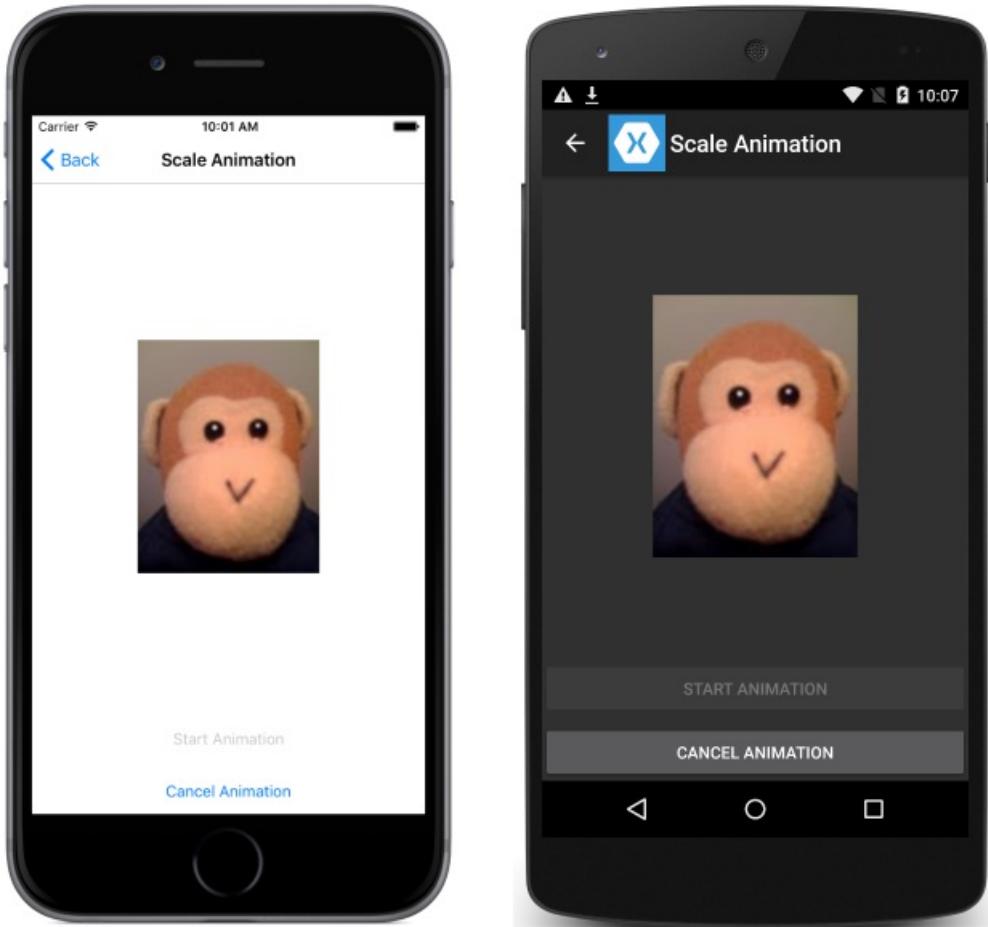
Scaling

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

In addition to the `ScaleTo` method, there are also `ScaleXTo` and `ScaleYTo` methods that animate the `ScaleX` and `ScaleY` properties, respectively.

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.

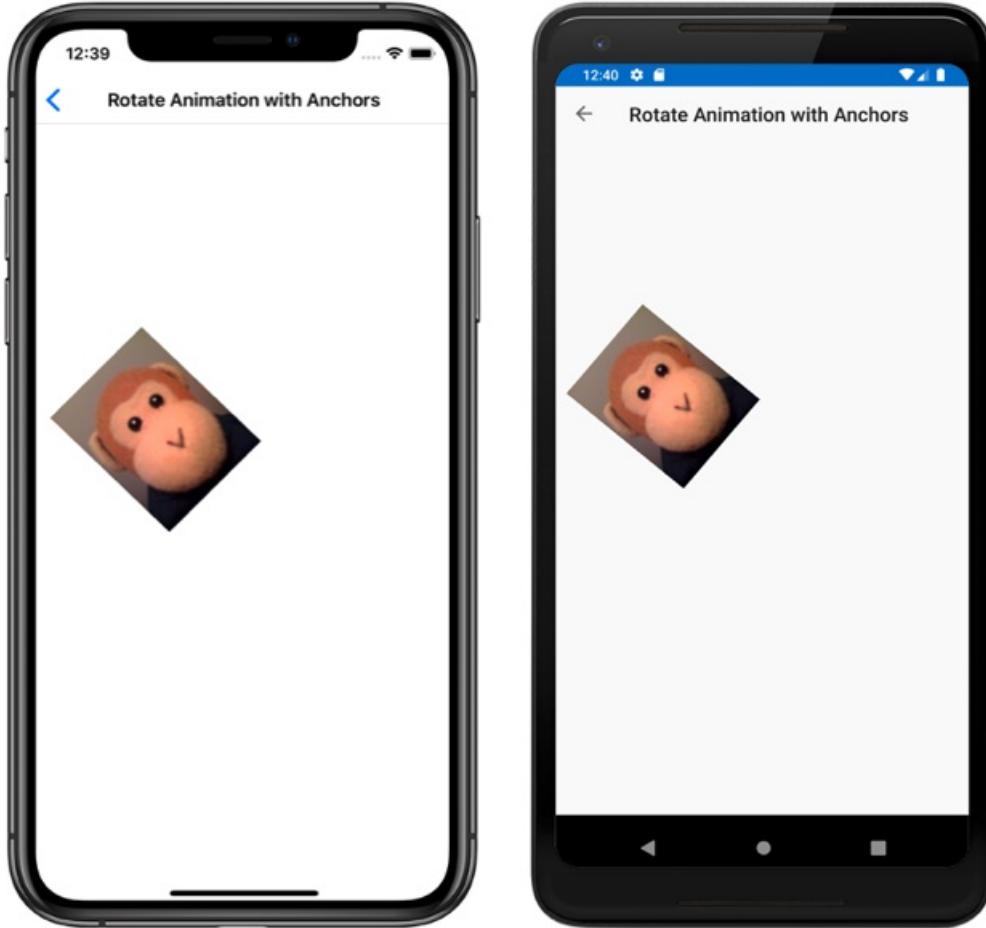
Scaling and Rotation with Anchors

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:

```
double radius = Math.Min(layout.Width, layout.Height) / 2;
image.AnchorY = radius / 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:



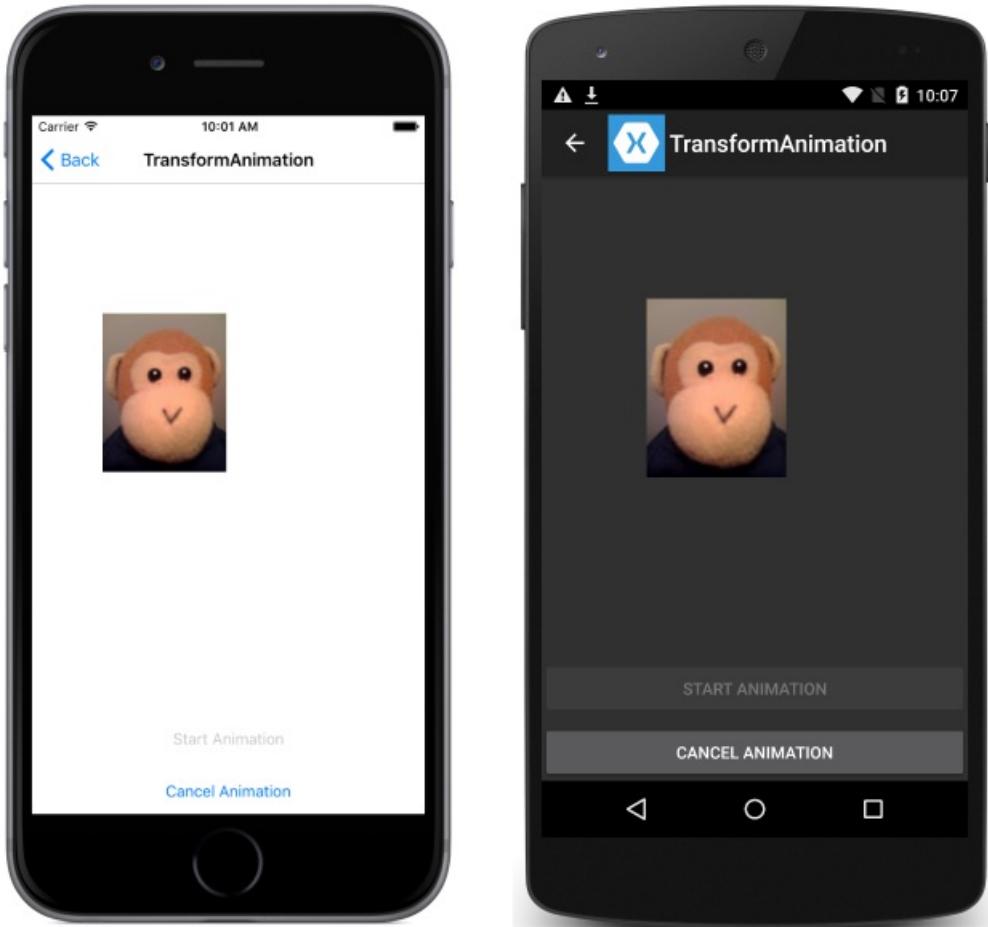
Translation

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.

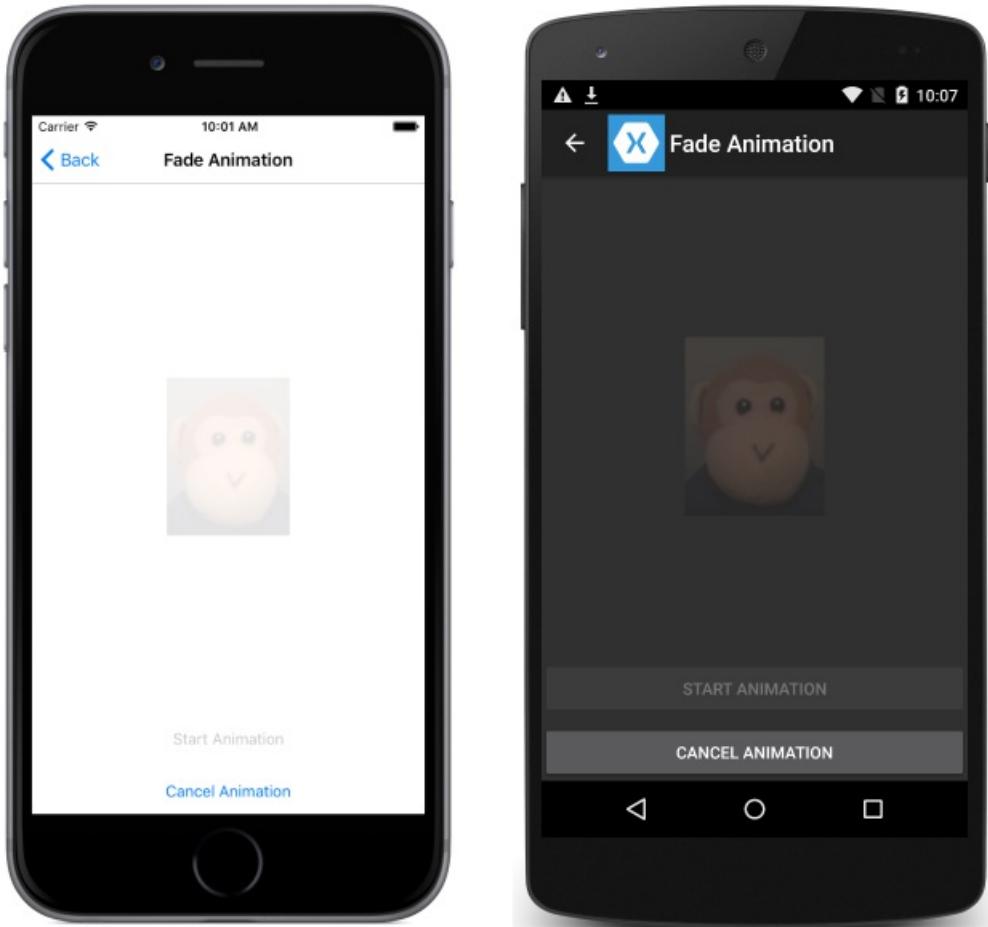
Fading

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 diagonally up and left
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.

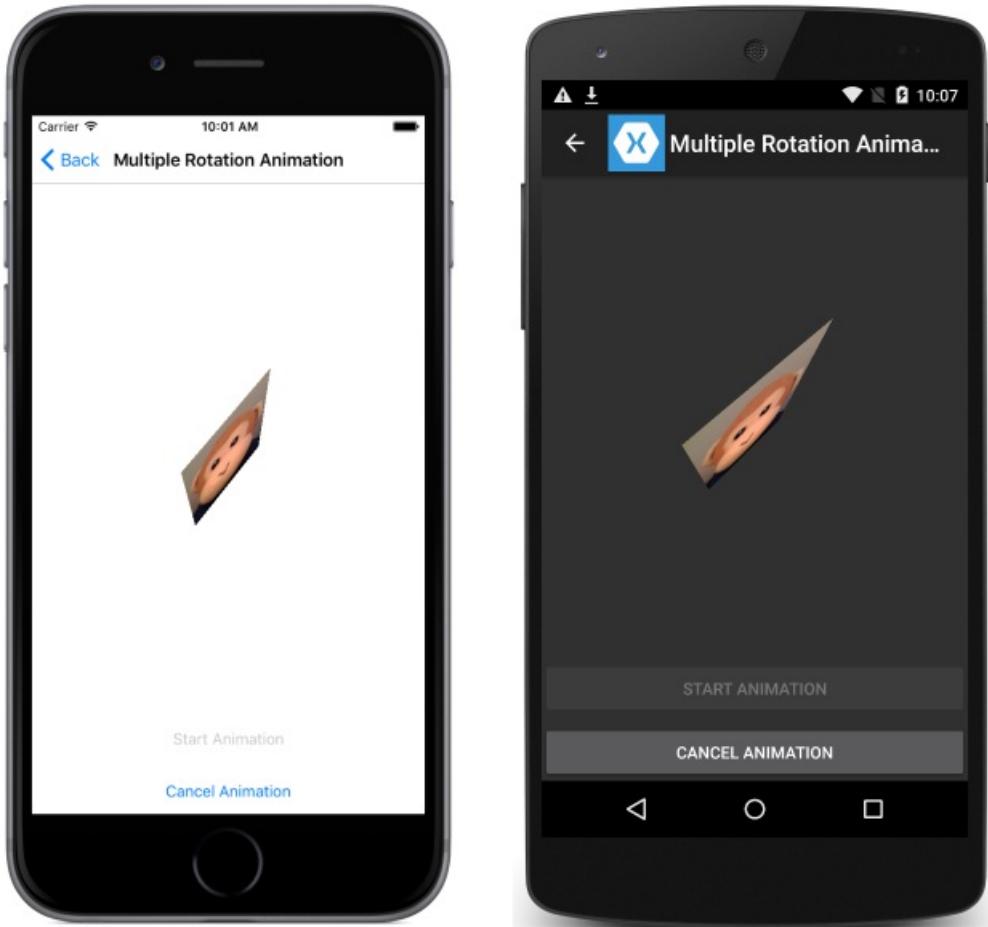
The `Task.WhenAll` method completes when all the methods in its collection have completed, as demonstrated in the following code example:

```
// 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 `CancelAnimations` extension method, as demonstrated in the following code example:

```
image.CancelAnimations();
```

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

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

NOTE

Xamarin.Forms 5.0 includes a type converter that converts a string representation of an easing function to the appropriate `Easing` enumeration member. This type converter is automatically invoked on any properties of type `Easing` that are set in XAML.

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:

```
double CustomEase (double t)
{
    return t == 0 || t == 1 ? t : (int)(5 * t) / 5.0;
}

await image.TranslateTo(0, 200, 2000, (Easing)CustomEase);
```

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> CustomEaseFunc = t => 9 * t * t * t - 13.5 * t * t + 5.5 * t;
await image.TranslateTo(0, 200, 2000, CustomEaseFunc);
```

The `CustomEaseFunc` 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

8/4/2022 • 10 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

In addition, the `Animation` class has an `IsEnabled` property that can be examined to determine if animations have been disabled by the operating system, such as when power saving mode is activated.

Create 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.

Cancel 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.

Create 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](#).

Create 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`.

Related links

- [Custom Animations \(sample\)](#)
- [Animation API](#)
- [AnimationExtensions API](#)

Xamarin.Forms Brushes

8/4/2022 • 2 minutes to read • [Edit Online](#)

A brush enables you to paint an area, such as the background of a control, using different approaches. Brush support in Xamarin.Forms is available in the `xamarin.Forms` namespace on iOS, Android, macOS, the Universal Windows Platform (UWP), and the Windows Presentation Foundation (WPF).

The `Brush` class is an abstract class that paints an area with its output. Classes that derive from `Brush` describe different ways of painting an area. The following list describes the different brush types available in Xamarin.Forms:

- `SolidColorBrush`, which paints an area with a solid color. For more information, see [Xamarin.Forms Brushes: Solid colors](#).
- `LinearGradientBrush`, which paints an area with a linear gradient. For more information, see [Xamarin.Forms Brushes: Linear gradients](#).
- `RadialGradientBrush`, which paints an area with a radial gradient. For more information, see [Xamarin.Forms Brushes: Radial gradients](#).

Instances of these brush types can be assigned to the `Stroke` and `Fill` properties of a `Shape`, and the `Background` property of a `VisualElement`.

NOTE

The `VisualElement.Background` property enables brushes to be used as the background in any control.

The `Brush` class also has an `IsNullOrEmpty` method that returns a `bool` that represents whether the brush is defined or not.

Xamarin.Forms Brushes: Solid colors

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `SolidColorBrush` class derives from the `Brush` class, and is used to paint an area with a solid color. There are a variety of approaches to specifying the color of a `SolidColorBrush`. For example, you can specify its color with a `Color` value or by using one of the predefined `SolidColorBrush` objects provided by the `Brush` class.

The `SolidColorBrush` class defines the `Color` property, of type `Color`, which represents the color of the brush. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

The `SolidColorBrush` class also has an `IsEmpty` method that returns a `bool` that represents whether the brush has been assigned a color.

Create a SolidColorBrush

There are three main techniques for creating a `SolidColorBrush`. You can create a `SolidColorBrush` from a `Color`, use a predefined brush, or create a `SolidColorBrush` using hexadecimal notation.

Use a predefined Color

Xamarin.Forms includes a type converter that creates a `SolidColorBrush` from a `Color` value. In XAML, this enables a `SolidColorBrush` to be created from a predefined `Color` value:

```
<Frame Background="DarkBlue"
       BorderColor="LightGray"
       HasShadow="True"
       CornerRadius="12"
       HeightRequest="120"
       WidthRequest="120" />
```

In this example, the background of the `Frame` is painted with a dark blue `SolidColorBrush`:



Alternatively, the `Color` value can be specified using property tag syntax:

```
<Frame BorderColor="LightGray"
       HasShadow="True"
       CornerRadius="12"
       HeightRequest="120"
       WidthRequest="120">
    <Frame.Background>
        <SolidColorBrush Color="DarkBlue" />
    </Frame.Background>
</Frame>
```

In this example, the background of the `Frame` is painted with a `SolidColorBrush` whose color is specified by setting the `SolidColorBrush.Color` property.

Use a predefined Brush

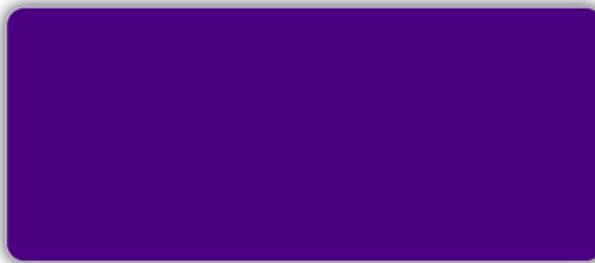
The `Brush` class defines a set of commonly used `SolidColorBrush` objects. The following example uses one of these predefined `SolidColorBrush` objects:

```
<Frame Background="{x:Static Brush.Indigo}"
       BorderColor="LightGray"
       HasShadow="True"
       CornerRadius="12"
       HeightRequest="120"
       WidthRequest="120" />
```

The equivalent C# code is:

```
Frame frame = new Frame
{
    Background = Brush.Indigo,
    BorderColor = Color.LightGray,
    // ...
};
```

In this example, the background of the `Frame` is painted with an indigo `SolidColorBrush`:



For a list of predefined `SolidColorBrush` objects provided by the `Brush` class, see [Solid color brushes](#).

Use hexadecimal notation

`SolidColorBrush` objects can also be created using hexadecimal notation. With this approach, a color is specified in terms of the amount of red, green, and blue to combine into a single color. The main format for specifying a color using hexadecimal notation is `#rrggbb`, where:

- `rr` is a two-digit hexadecimal number specifying the relative amount of red.
- `gg` is a two-digit hexadecimal number specifying the relative amount of green.
- `bb` is a two-digit hexadecimal number specifying the relative amount of blue.

In addition, a color can be specified as `#aarrggbb` where `aa` specifies the alpha value, or transparency, of the color. This approach enables you to create colors that are partially transparent.

The following example sets the color value of a `SolidColorBrush` using hexadecimal notation:

```
<Frame Background="#FF9988"
    BorderColor="LightGray"
    HasShadow="True"
    CornerRadius="12"
    HeightRequest="120"
    WidthRequest="120" />
```

In this example, the background of the `Frame` is painted with a salmon-colored `SolidColorBrush`:



For other ways of describing color, see [Colors in Xamarin.Forms](#).

Solid color brushes

For convenience, the `Brush` class provides a set of commonly used `SolidColorBrush` objects, such as `AliceBlue` and `YellowGreen`. The following image shows the color of each predefined brush, its name, and its hexadecimal value:

AliceBlue	#FFFF0FFF	DarkGreen	#FFB006400	Gray	#FFB08080	LightYellow	#FFFFFFE0	OrangeRed	#FFFF4500	StateGray	#FF708090
AntiqueWhite	#FFFFEBD7	DarkKhaki	#FFB0B76B	Green	#FF008000	Lime	#FF00FF00	Orchid	#FFDA70D6	Show	#FFFFFAFA
Aqua	#FF00FFFF	DarkMagenta	#FFB0080B	GreenYellow	#FFADFF2F	LimeGreen	#FF32CD32	PaleGoldenrod	#FFEE8AA	SpringGreen	#FF00FF7F
Aquamarine	#FF7FFFDF	DarkOliveGreen	#FF556B2F	Honeydew	#FFDFFF00	Linen	#FFFA0E06	PaleGreen	#FFFE4E1	SteelBlue	#FF4682B4
Azure	#FF00FFFF	DarkOrange	#FFF8C00	HotPink	#FFFF69B4	Magenta	#FFFF00FF	PaleTurquoise	#FFAFEEEE	Tan	#FFD2B48C
Beige	#FFFF5FDC	DarkOrchid	#FF932CC	IndianRed	#FC005CSC	Maroon	#FFB00000	PaleVioletRed	#FFD87093	Teal	#FF008080
Bisque	#FFFFE4C4	DarkRed	#FFB80000	Indigo	#FF4B0082	MediumAquamarine	#FF6CDCAA	PapayaWhip	#FFFEFD5	Thistle	#FFDB8FD8
Black	#FF000000	DarkSalmon	#FFE9967A	Ivory	#FFFFF0F0	MediumBlue	#FF0000CD	PeachPuff	#FFFFDA89	Tomato	#FFFF6347
BlanchedAlmond	#FFFBEEBC	DarkSeaGreen	#FF88BCF	Khaki	#FFFFF0F0	MediumOrchid	#FFBAA5D3	Peru	#FFC0853F	Transparent	#00FFFFFF
Blue	#FF0000FF	DarkSlateBlue	#F48308B	Lavender	#FFEE66FA	MediumPurple	#FF9370DB	Pink	#FFFFCCB	Turquoise	#FF40E0D0
BlueViolet	#FFBA28E2	DarkSlateGray	#FF214F4F	LavenderBlush	#FFFFFOE5	MediumSeaGreen	#FF3C8371	Plum	#FFD0A0D0	Violet	#FFEE82EE
Brown	#FFA52A2A	DarkTurquoise	#FF00CED1	LawnGreen	#FF7FCFC00	MediumSlateBlue	#FF7B68EE	PowderBlue	#FFB0E0E6	Wheat	#FF55DE83
BurlyWood	#FFDEB887	DarkViolet	#FF9400D3	LemonChiffon	#FFFFFACD	MediumSpringGreen	#FF00FA9A	Purple	#FF800080	White	#FFFFFFFFFF
CadetBlue	#FF5F9EA0	DeepPink	#FFF1493	LightBlue	#FFADD8E6	MediumTurquoise	#FF4B01CC	Red	#FFFF0000	WhiteSmoke	#FF5F5F5
Chartreuse	#FF7FFF00	DeepSkyBlue	#FF00BFFF	LightCoral	#FFFB080B0	MediumVioletRed	#FFC71585	RosyBrown	#FFBC8F8F	Yellow	#FFFFFF00
Chocolate	#FFD2691E	DimGray	#FF696969	LightCyan	#FFEDFFF	MidnightBlue	#FF191970	RoyalBlue	#FF4169E1	YellowGreen	#FF9ACD32
Coral	#FF7F7F50	DodgerBlue	#FF1E90FF	LightGoldenrodYellow	#FFFAFAD2	MintCream	#FF55FFFA	SaddleBrown	#FFB8A513		
CornflowerBlue	#FF6495ED	Firebrick	#FFB22222	LightGray	#FFD3D3D3	MistyRose	#FFFE4E1	Salmon	#FFFA8072		
Cornsilk	#FFFFF8DC	FloralWhite	#FFFFFA0	LightGreen	#FF90EE90	Moccasin	#FFFE4E85	SandyBrown	#FFFA4A60		
Crimson	#FFDC143C	ForestGreen	#FF28B22	LightPink	#FFFB6C1	NavajoWhite	#FFFDDEAD	SeaGreen	#FF2E8B57		
Cyan	#FF00FFFF	Fuchsia	#FFF00FFF	LightSalmon	#FFFA0A7A	Navy	#FF000080	Seashell	#FFFFF5EE		
DarkBlue	#FF00008B	Gainsboro	#FDCCDCD	LightSeaGreen	#FFB02BAA	OldLace	#FF00008B	Sienna	#FFA0522D		
DarkCyan	#FF00808B	GhostWhite	#FFF8BF	LightSkyBlue	#FB7CECFA	Olive	#FFB08000	Silver	#FFCC00C0		
DarkGoldenrod	#FFB8860B	Gold	#FFFFD700	LightSlateGray	#FF778899	OliveDrab	#FFB88E23	SkyBlue	#FF87CEEB		
DarkGray	#FFA9A9A9	Goldenrod	#FFDA520	LightSteelBlue	#FFB0C4DE	Orange	#FFFA500	SlateBlue	#FFB8A5CD		

Related links

- [BrushesDemos \(sample\)](#)
- [Colors in Xamarin.Forms](#)

Xamarin.Forms Brushes: Gradients

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `GradientBrush` class derives from the `Brush` class, and is an abstract class that describes a gradient, which is composed of gradient stops. A gradient brush paints an area with multiple colors that blend into each other along an axis. Classes that derive from `GradientBrush` describe different ways of interpreting gradient stops, and Xamarin.Forms provides the following gradient brushes:

- `LinearGradientBrush`, which paints an area with a linear gradient. For more information, see [Xamarin.Forms Brushes: Linear gradients](#).
- `RadialGradientBrush`, which paints an area with a radial gradient. For more information, see [Xamarin.Forms Brushes: Radial gradients](#).

The `GradientBrush` class defines the `GradientStops` property, of type `GradientStopsCollection`, which represents the brush's gradient stops, each of which specifies a color and an offset along the brush's gradient axis. A `GradientStopsCollection` is an `ObservableCollection` of `GradientStop` objects. The `GradientStops` property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

NOTE

The `GradientStops` property is the `ContentProperty` of the `GradientBrush` class, and so does not need to be explicitly set from XAML.

Gradient stops

Gradient stops are the building blocks of a gradient brush, and specify the colors in the gradient and their location along the gradient axis. Gradient stops are specified using `GradientStop` objects.

The `GradientStop` class defines the following properties:

- `Color`, of type `Color`, which represents the color of the gradient stop. The default value of this property is `Color.Default`.
- `Offset`, of type `float`, which represents the location of the gradient stop within the gradient vector. The default value of this property is 0, and valid values are in the range 0.0-1.0. The closer this value is to 0, the closer the color is to the start of the gradient. Similarly, the closer this value is to 1, the closer the color is to the end of the gradient.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

IMPORTANT

The coordinate system used by gradients is relative to a bounding box for the output area. 0 indicates 0 percent of the bounding box, and 1 indicates 100 percent of the bounding box. Therefore, (0.5,0.5) describes a point in the middle of the bounding box, and (1,1) describes a point at the bottom right of the bounding box.

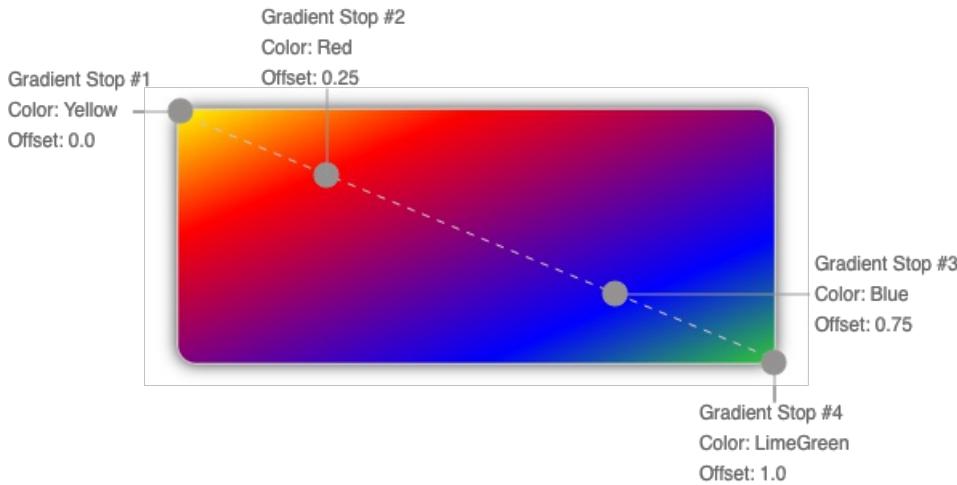
The following XAML example creates a diagonal `LinearGradientBrush` with four colors:

```

<LinearGradientBrush StartPoint="0,0"
                     EndPoint="1,1">
    <GradientStop Color="Yellow"
                  Offset="0.0" />
    <GradientStop Color="Red"
                  Offset="0.25" />
    <GradientStop Color="Blue"
                  Offset="0.75" />
    <GradientStop Color="LimeGreen"
                  Offset="1.0" />
</LinearGradientBrush>

```

The color of each point between gradient stops is interpolated as a combination of the color specified by the two bounding gradient stops. The following diagram shows the gradient stops from the previous example:



In this diagram, the circles mark the position of gradient stops, and the dashed line shows the gradient axis. The first gradient stop specifies the color yellow at an offset of 0.0. The second gradient stop specifies the color red at an offset of 0.25. The points between these two gradient stops gradually change from yellow to red as you move from left to right along the gradient axis. The third gradient stop specifies the color blue at an offset of 0.75. The points between the second and third gradient stops gradually change from red to blue. The fourth gradient stop specifies the color lime green at an offset of 1.0. The points between the third and fourth gradient stops gradually change from blue to lime green.

Related links

- [BrushesDemos \(sample\)](#)
- [Xamarin.Forms Brushes: Linear gradients](#)
- [Xamarin.Forms Brushes: Radial gradients](#)

Xamarin.Forms Brushes: Linear gradients

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

The `LinearGradientBrush` class derives from the `GradientBrush` class, and paints an area with a linear gradient, which blends two or more colors along a line known as the gradient axis. `GradientStop` objects are used to specify the colors in the gradient and their positions. For more information about `GradientStop` objects, see [Xamarin.Forms Brushes: Gradients](#).

The `LinearGradientBrush` class defines the following properties:

- `StartPoint`, of type `Point`, which represents the starting two-dimensional coordinates of the linear gradient. The default value of this property is (0,0).
- `EndPoint`, of type `Point`, which represents the ending two-dimensional coordinates of the linear gradient. The default value of this property is (1,1).

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `LinearGradientBrush` class also has an `IsEmpty` method that returns a `bool` that represents whether the brush has been assigned any `GradientStop` objects.

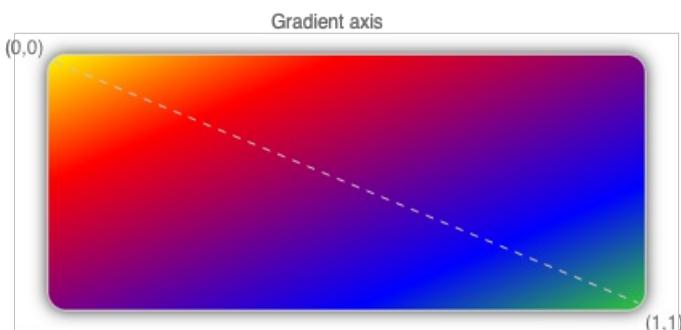
NOTE

Linear gradients can also be created with the `linear-gradient()` CSS function.

Create a LinearGradientBrush

A linear gradient brush's gradient stops are positioned along the gradient axis. The orientation and size of the gradient axis can be changed using the brush's `StartPoint` and `EndPoint` properties. By manipulating these properties, you can create horizontal, vertical, and diagonal gradients, reverse the gradient direction, condense the gradient spread, and more.

The `StartPoint` and `EndPoint` properties are relative to the area being painted. (0,0) represents the top-left corner of the area being painted, and (1,1) represents the bottom-right corner of the area being painted. The following diagram shows the gradient axis for a diagonal linear gradient brush:



In this diagram, the dashed line shows the gradient axis, which highlights the interpolation path of the gradient from the start point to the end point.

Create a horizontal linear gradient

To create a horizontal linear gradient, create a `LinearGradientBrush` object and set its `StartPoint` to (0,0) and its `EndPoint` to (1,0). Then, add two or more `GradientStop` objects to the `LinearGradientBrush.GradientStops` collection, that specify the colors in the gradient and their positions.

The following XAML example shows a horizontal `LinearGradientBrush` that's set as the `Background` of a `Frame`:

```
<Frame BorderColor="LightGray"
       HasShadow="True"
       CornerRadius="12"
       HeightRequest="120"
       WidthRequest="120">
  <Frame.Background>
    <!-- StartPoint defaults to (0,0) -->
    <LinearGradientBrush EndPoint="1,0">
      <GradientStop Color="Yellow"
                     Offset="0.1" />
      <GradientStop Color="Green"
                     Offset="1.0" />
    </LinearGradientBrush>
  </Frame.Background>
</Frame>
```

In this example, the background of the `Frame` is painted with a `LinearGradientBrush` that interpolates from yellow to green horizontally:



Create a vertical linear gradient

To create a vertical linear gradient, create a `LinearGradientBrush` object and set its `StartPoint` to (0,0) and its `EndPoint` to (0,1). Then, add two or more `GradientStop` objects to the `LinearGradientBrush.GradientStops` collection, that specify the colors in the gradient and their positions.

The following XAML example shows a vertical `LinearGradientBrush` that's set as the `Background` of a `Frame`:

```
<Frame BorderColor="LightGray"
       HasShadow="True"
       CornerRadius="12"
       HeightRequest="120"
       WidthRequest="120">
  <Frame.Background>
    <!-- StartPoint defaults to (0,0) -->
    <LinearGradientBrush EndPoint="0,1">
      <GradientStop Color="Yellow"
                     Offset="0.1" />
      <GradientStop Color="Green"
                     Offset="1.0" />
    </LinearGradientBrush>
  </Frame.Background>
</Frame>
```

In this example, the background of the `Frame` is painted with a `LinearGradientBrush` that interpolates from

yellow to green vertically:



Create a diagonal linear gradient

To create a diagonal linear gradient, create a `LinearGradientBrush` object and set its `StartPoint` to (0,0) and its `EndPoint` to (1,1). Then, add two or more `GradientStop` objects to the `LinearGradientBrush.GradientStops` collection, that specify the colors in the gradient and their positions.

The following XAML example shows a diagonal `LinearGradientBrush` that's set as the `Background` of a `Frame`:

```
<Frame BorderColor="LightGray"
       HasShadow="True"
       CornerRadius="12"
       HeightRequest="120"
       WidthRequest="120">
    <Frame.Background>
        <!-- StartPoint defaults to (0,0)
            Endpoint defaults to (1,1) -->
        <LinearGradientBrush>
            <GradientStop Color="Yellow"
                           Offset="0.1" />
            <GradientStop Color="Green"
                           Offset="1.0" />
        </LinearGradientBrush>
    </Frame.Background>
</Frame>
```

In this example, the background of the `Frame` is painted with a `LinearGradientBrush` that interpolates from yellow to green diagonally:



Related links

- [BrushesDemos \(sample\)](#)
- [Xamarin.Forms Brushes: Gradients](#)

Xamarin.Forms Brushes: Radial gradients

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

The `RadialGradientBrush` class derives from the `GradientBrush` class, and paints an area with a radial gradient, which blends two or more colors across a circle. `GradientStop` objects are used to specify the colors in the gradient and their positions. For more information about `GradientStop` objects, see [Xamarin.Forms Brushes: Gradients](#).

The `RadialGradientBrush` class defines the following properties:

- `Center`, of type `Point`, which represents the center point of the circle for the radial gradient. The default value of this property is (0.5,0.5).
- `Radius`, of type `double`, which represents the radius of the circle for the radial gradient. The default value of this property is 0.5.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `RadialGradientBrush` class also has an `IsEmpty` method that returns a `bool` that represents whether the brush has been assigned any `GradientStop` objects.

NOTE

Radial gradients can also be created with the `radial-gradient()` CSS function.

Create a RadialGradientBrush

A radial gradient brush's gradient stops are positioned along a gradient axis defined by a circle. The gradient axis radiates from the center of the circle to its circumference. The position and size of the circle can be changed using the brush's `Center` and `Radius` properties. The circle defines the end point of the gradient. Therefore, a gradient stop at 1.0 defines the color at the circle's circumference. A gradient stop at 0.0 defines the color at the center of the circle.

To create a radial gradient, create a `RadialGradientBrush` object and set its `Center` and `Radius` properties. Then, add two or more `GradientStop` objects to the `RadialGradientBrush.GradientStops` collection, that specify the colors in the gradient and their positions.

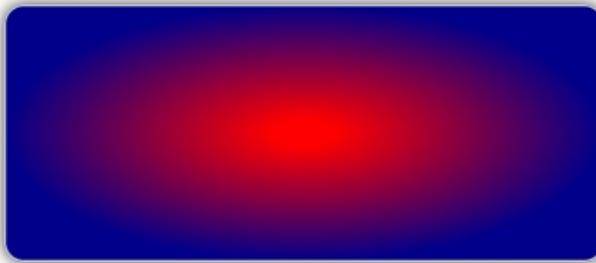
The following XAML example shows a `RadialGradientBrush` that's set as the `Background` of a `Frame`:

```

<Frame BorderColor="LightGray"
       HasShadow="True"
       CornerRadius="12"
       HeightRequest="120"
       WidthRequest="120">
    <Frame.Background>
        <!-- Center defaults to (0.5,0.5)
             Radius defaults to (0.5) -->
        <RadialGradientBrush>
            <GradientStop Color="Red"
                           Offset="0.1" />
            <GradientStop Color="DarkBlue"
                           Offset="1.0" />
        </RadialGradientBrush>
    </Frame.Background>
</Frame>

```

In this example, the background of the `Frame` is painted with a `RadialGradientBrush` that interpolates from red to dark blue. The center of the radial gradient is positioned in the center of the `Frame`:



The following XAML example moves the center of the radial gradient to the top-left corner of the `Frame`:

```

<!-- Radius defaults to (0.5) -->
<RadialGradientBrush Center="0.0,0.0">
    <GradientStop Color="Red"
                  Offset="0.1" />
    <GradientStop Color="DarkBlue"
                  Offset="1.0" />
</RadialGradientBrush>

```

In this example, the background of the `Frame` is painted with a `RadialGradientBrush` that interpolates from red to dark blue. The center of the radial gradient is positioned in the top-left of the `Frame`:



The following XAML example moves the center of the radial gradient to the bottom-right corner of the `Frame`:

```
<!-- Radius defaults to (0.5) -->
<RadialGradientBrush Center="1.0,1.0">
    <GradientStop Color="Red"
        Offset="0.1" />
    <GradientStop Color="DarkBlue"
        Offset="1.0" />
</RadialGradientBrush>
```

In this example, the background of the `Frame` is painted with a `RadialGradientBrush` that interpolates from red to dark blue. The center of the radial gradient is positioned in the bottom-right of the `Frame`:



Related links

- [BrushesDemos \(sample\)](#)
- [Xamarin.Forms Brushes: Gradients](#)

Colors in Xamarin.Forms

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

The `Color` structure lets you specify colors as Red-Green-Blue (RGB) values, Hue-Saturation-Luminosity (HSL) values, Hue-Saturation-Value (HSV) values, or with a color name. An Alpha channel is also available to indicate transparency.

`Color` objects can be created with the `Color` constructors, which can be used to specify a [gray shade](#), an [RGB value](#), or an [RGB value with transparency](#). In all cases, arguments are `double` values ranging from 0 to 1.

You can also use static methods to create `Color` objects:

- `Color.FromRgb` for `double` RGB values from 0 to 1.
- `Color.FromRgb` for integer RGB values from 0 to 255.
- `Color.FromRgba` for `double` RGB values with transparency.
- `Color.FromRgba` for integer RGB values with transparency.
- `Color.FromHsla` for `double` HSL values with transparency.
- `Color.FromHsv` for `double` HSV values from 0 to 1.
- `Color.FromHsv` for integer HSV values from 0 to 255.
- `Color.FromHsva` for `double` HSV values with transparency.
- `Color.FromHsva` for integer HSV values with transparency.
- `Color.FromUint` for a `uint` value calculated as $(B + 256 * (G + 256 * (R + 256 * A)))$.
- `Color.FromHex` for a `string` format of hexadecimal digits in the form "#AARRGGBB" or "#RRGGBB" or "#ARGB" or "#RGB", where each letter corresponds to a hexadecimal digit for the alpha, red, green, and blue channels.

Once created, a `Color` object is immutable. The characteristics of the color can be obtained from the following properties:

- `R`, which represents the red channel of the color.
- `G`, which represents the green channel of the color.
- `B`, which represents the blue channel of the color.
- `A`, which represents the alpha channel of the color.
- `Hue`, which represents the hue channel of the color.
- `Saturation`, which represents the saturation channel of the color.
- `Luminosity`, which represents the luminosity channel of the color.

These properties are all `double` values ranging from 0 to 1.

Named colors

The `Color` structure also defines 240 public static read-only fields for common colors, such as `AliceBlue`.

Color.Accent

The `Color.Accent` value results in a platform-specific (and sometimes user-selectable) color that is visible on either a dark or light background.

Color.Default

The `Color.Default` value defines a `Color` with all channels set to -1, and is intended to enforce the platform's color scheme. Consequently, it has a different meaning in different contexts on different platforms. By default the platform color schemes are:

- iOS: dark text on a light background.
- Android: dark text on a light background.
- Windows: dark text on a light background.

Color.Transparent

The `Color.Transparent` value defines a `Color` with all channels set to zero.

Modify a color

Several instance methods allow modifying an existing color to create a new color:

- `AddLuminosity` returns a `Color` by modifying the luminosity by the supplied delta.
- `MultiplyAlpha` returns a `Color` by modifying the alpha, multiplying it by the supplied alpha value.
- `ToHex` returns a hexadecimal `string` representation of a `Color`.
- `WithHue` returns a `Color`, replacing the hue with the value supplied.
- `WithLuminosity` returns a `Color`, replacing the luminosity with the value supplied.
- `WithSaturation` returns a `Color`, replacing the saturation with the value supplied.

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;
```

Examples

In XAML, colors are typically referenced using their named values, or with their Hex representations:

```
<Label Text="Sea color"
      TextColor="Aqua" />
<Label Text="RGB"
      TextColor="#00FF00" />
<Label Text="Alpha plus RGB"
      TextColor="#CC00FF00" />
<Label Text="Tiny RGB"
      TextColor="#0F0" />
<Label Text="Tiny Alpha plus RGB"
      TextColor="#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](#).

In C#, colors are typically referenced using their named values, or with their static methods:

```
Label red    = new Label { Text = "Red",    TextColor = Color.Red };
Label orange = new Label { Text = "Orange", TextColor = Color.FromHex("FF6A00") };
Label yellow = new Label { Text = "Yellow", TextColor = Color.FromHsla(0.167, 1.0, 0.5, 1.0) };
Label green   = new Label { Text = "Green",   TextColor = Color.FromRgb (38, 127, 0) };
Label blue    = new Label { Text = "Blue",    TextColor = Color.FromRgba(0, 38, 255, 255) };
Label indigo  = new Label { Text = "Indigo",  TextColor = Color.FromRgb (0, 72, 255) };
Label violet  = new Label { Text = "Violet", TextColor = Color.FromHsla(0.82, 1, 0.25, 1) };
```

The following example uses the `OnPlatform` markup extension to selectively set the color of an `ActivityIndicator`:

```
<ActivityIndicator Color="{OnPlatform iOS=Black, Default=Default}"
                  IsRunning="True" />
```

The equivalent C# code is:

```
ActivityIndicator activityIndicator = new ActivityIndicator
{
    Color = Device.RuntimePlatform == Device.iOS ? Color.Black : Color.Default,
    IsRunning = true
};
```

Related links

- [ColorsSample](#)
- [Bindable Picker \(sample\)](#)

Display Pop-ups

8/4/2022 • 4 minutes to read • [Edit Online](#)

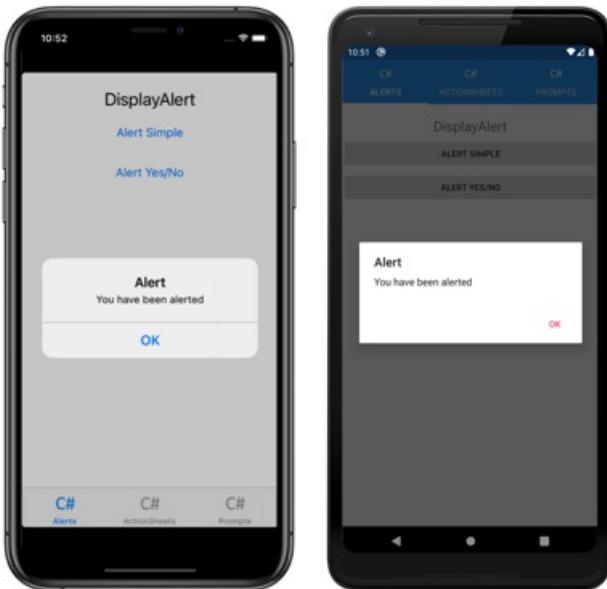
 [Download the sample](#)

Displaying an alert, asking a user to make a choice, or displaying a prompt is a common UI task. Xamarin.Forms has three methods on the `Page` class for interacting with the user via a pop-up: `DisplayAlert`, `DisplayActionSheet`, and `DisplayPromptAsync`. They are rendered with appropriate native controls on each platform.

Display 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:

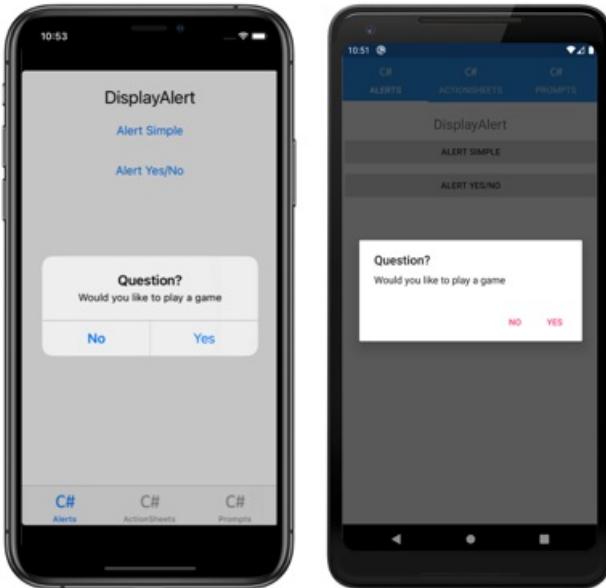
```
await DisplayAlert ("Alert", "You have been alerted", "OK");
```



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)
{
    bool answer = await DisplayAlert ("Question?", "Would you like to play a game", "Yes", "No");
    Debug.WriteLine ("Answer: " + answer);
}
```



The `DisplayAlert` method also has overloads that accept a `FlowDirection` argument that specifies the direction in which UI elements flow within the alert. For more information about flow direction, see [Right-to-left localization](#).

WARNING

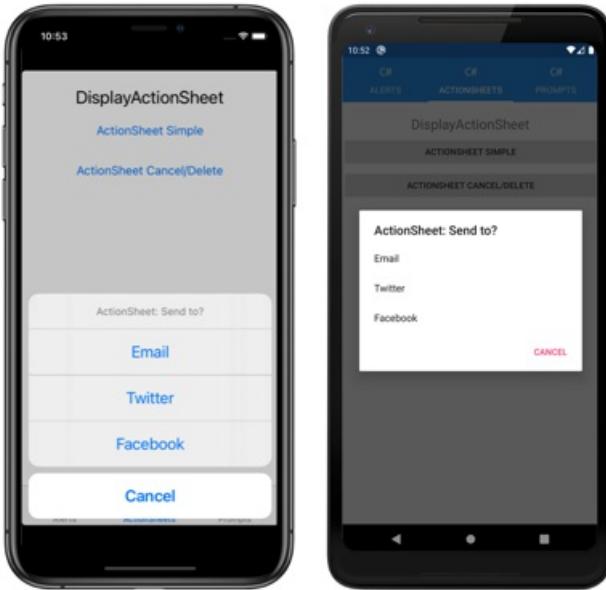
By default on UWP, when an alert is displayed any access keys that are defined on the page behind the alert can still be activated. For more information, see [VisualElement Access Keys on Windows](#).

Guide users through tasks

The `UIActionSheet` is a common UI element in iOS. The `Xamarin.Forms` `DisplayActionSheet` method lets you include this control in cross-platforms apps, rendering native alternatives in Android and UWP.

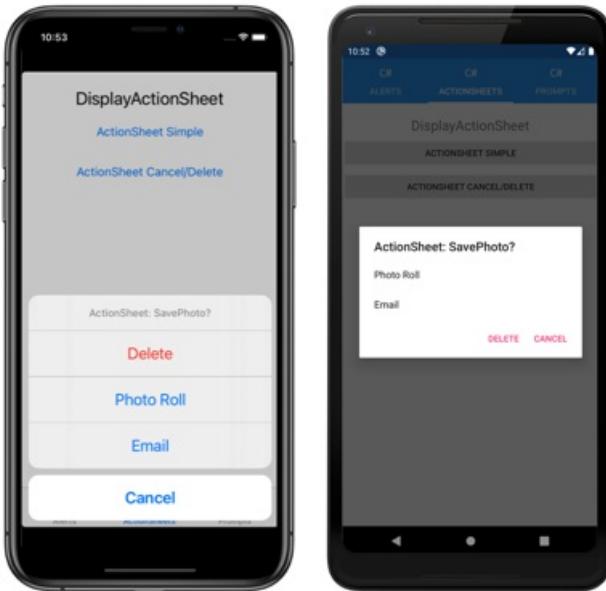
To display an action sheet, `await DisplayActionSheet` in any `Page`, passing the message and button labels as strings. The method returns the string label of the button that was clicked by the user. A simple example is shown here:

```
async void OnActionSheetSimpleClicked (object sender, EventArgs e)
{
    string action = await DisplayActionSheet ("ActionSheet: Send to?", "Cancel", null, "Email", "Twitter",
    "Facebook");
    Debug.WriteLine ("Action: " + action);
}
```



The `destroy` button is rendered differently to the other buttons on iOS, 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)
{
    string action = await DisplayActionSheet ("ActionSheet: SavePhoto?", "Cancel", "Delete", "Photo Roll",
    "Email");
    Debug.WriteLine ("Action: " + action);
}
```



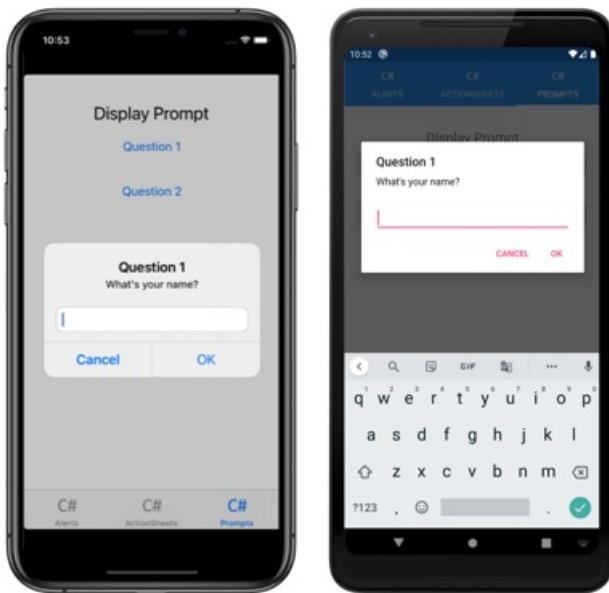
The `DisplayActionSheet` method also has an overload that accepts a `FlowDirection` argument that specifies the direction in which UI elements flow within the action sheet. For more information about flow direction, see [Right-to-left localization](#).

Display a prompt

To display a prompt, call the `DisplayPromptAsync` in any `Page`, passing a title and message as `string` arguments:

```
string result = await DisplayPromptAsync("Question 1", "What's your name?");
```

The prompt is displayed modally:



If the OK button is tapped, the entered response is returned as a `string`. If the Cancel button is tapped, `null` is returned.

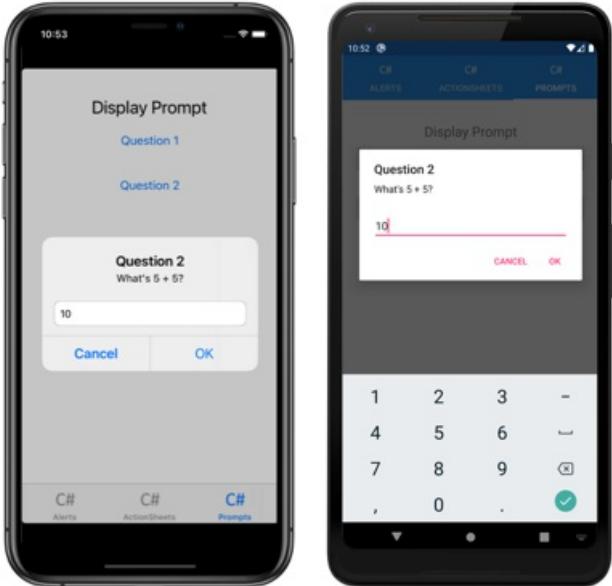
The full argument list for the `DisplayPromptAsync` method is:

- `title`, of type `string`, is the title to display in the prompt.
- `message`, of type `string`, is the message to display in the prompt.
- `accept`, of type `string`, is the text for the accept button. This is an optional argument, whose default value is `OK`.
- `cancel`, of type `string`, is the text for the cancel button. This is an optional argument, whose default value is `Cancel`.
- `placeholder`, of type `string`, is the placeholder text to display in the prompt. This is an optional argument, whose default value is `null`.
- `maxLength`, of type `int`, is the maximum length of the user response. This is an optional argument, whose default value is `-1`.
- `keyboard`, of type `Keyboard`, is the keyboard type to use for the user response. This is an optional argument, whose default value is `Keyboard.Default`.
- `initialValue`, of type `string`, is a pre-defined response that will be displayed, and which can be edited. This is an optional argument, whose default value is an empty `string`.

The following example shows setting some of the optional arguments:

```
string result = await DisplayPromptAsync("Question 2", "What's 5 + 5?", initialValue: "10", maxLength: 2, keyboard: Keyboard.Numeric);
```

This code displays a predefined response of 10, limits the number of characters that can be input to 2, and displays the numeric keyboard for user input:



WARNING

By default on UWP, when a prompt is displayed any access keys that are defined on the page behind the prompt can still be activated. For more information, see [VisualElement Access Keys on Windows](#).

Related links

- [PopupsSample](#)
- [Right-to-left localization](#)

Fonts in Xamarin.Forms

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

By default, Xamarin.Forms uses a system font defined by each platform. However, controls that display text define properties that you can use to change this font:

- `FontAttributes`, of type `FontAttributes`, which is an enumeration with three members: `None`, `Bold`, and `Italic`. The default value of this property is `None`.
- `FontSize`, of type `double`.
- `FontFamily`, of type `string`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

Set font attributes

Controls that display text can set the `FontAttributes` property to specify font attributes:

```
<Label Text="Italics"
      FontAttributes="Italic" />
<Label Text="Bold and italics"
      FontAttributes="Bold, Italic" />
```

The equivalent C# code is:

```
Label label1 = new Label
{
    Text = "Italics",
    FontAttributes = FontAttributes.Italic
};

Label label2 = new Label
{
    Text = "Bold and italics",
    FontAttributes = FontAttributes.Bold | FontAttributes.Italic
};
```

Set the font size

Controls that display text can set the `FontSize` property to specify the font size. The `FontSize` property can be set to a `double` value directly, or by a `NamedSize` enumeration value:

```
<Label Text="Font size 24"
      FontSize="24" />
<Label Text="Large font size"
      FontSize="Large" />
```

The equivalent C# code is:

```
Label label1 = new Label
{
    Text = "Font size 24",
    FontSize = 24
};

Label label2 = new Label
{
    Text = "Large font size",
    FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label))
};
```

Alternatively, the `Device.GetNamedSize` method has an override that specifies the second argument as an [Element](#):

```
Label myLabel = new Label
{
    Text = "Large font size",
};
myLabel.FontSize = Device.GetNamedSize(NamedSize.Large, myLabel);
```

NOTE

The `FontSize` value, when specified as a `double`, is measured in device-independent units. For more information, see [Units of Measurement](#).

For more information about named font sizes, see [Understand named font sizes](#).

Set the font family

Controls that display text can set the `FontFamily` property to a font family name, such as "Times Roman". However, this will only work if that font family is supported on the particular platform.

There are a number of techniques that can be used to attempt to derive the fonts that are available on a platform. However, the presence of a TTF (True Type Format) font file does not necessarily imply a font family, and TTFs are often included that are not intended for use in applications. In addition, the fonts installed on a platform can change with platform version. Therefore, the most reliable approach for specifying a font family is to use a custom font.

Custom fonts can be added to your Xamarin.Forms shared project and consumed by platform projects without any additional work. The process for accomplishing this is as follows:

1. Add the font to your Xamarin.Forms shared project as an embedded resource (**Build Action: EmbeddedResource**).
2. Register the font file with the assembly, in a file such as `AssemblyInfo.cs`, using the `ExportFont` attribute. An optional alias can also be specified.

The following example shows the Lobster-Regular font being registered with the assembly, along with an alias:

```
using Xamarin.Forms;

[assembly: ExportFont("Lobster-Regular.ttf", Alias = "Lobster")]
```

NOTE

The font can reside in any folder in the shared project, without having to specify the folder name when registering the font with the assembly.

On Windows, 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 font can then be consumed on each platform by referencing its name, without the file extension:

```
<!-- Use font name -->
<Label Text="Hello Xamarin.Forms"
      FontFamily="Lobster-Regular" />
```

Alternatively, it can be consumed on each platform by referencing its alias:

```
<!-- Use font alias -->
<Label Text="Hello Xamarin.Forms"
      FontFamily="Lobster" />
```

The equivalent C# code is:

```
// Use font name
Label label1 = new Label
{
    Text = "Hello Xamarin.Forms!",
    FontFamily = "Lobster-Regular"
};

// Use font alias
Label label2 = new Label
{
    Text = "Hello Xamarin.Forms!",
    FontFamily = "Lobster"
};
```

The following screenshots show the custom font:



IMPORTANT

For release builds on Windows, ensure the assembly containing the custom font is passed as an argument in the `Forms.Init` method call. For more information, see [Troubleshooting](#).

Set font properties per platform

The `OnPlatform` and `On` classes can be used in XAML to set font properties per platform. The example below sets different font families and sizes on each platform:

```

<Label Text="Different font properties on different platforms"
    FontSize="{OnPlatform iOS=20, Android=Medium, UWP=24}">
    <Label.FontFamily>
        <OnPlatform x:TypeArguments="x:String">
            <On Platform="iOS" Value="MarkerFelt-Thin" />
            <On Platform="Android" Value="Lobster-Regular" />
            <On Platform="UWP" Value="ArimaMadurai-Black" />
        </OnPlatform>
    </Label.FontFamily>
</Label>

```

The `Device.RuntimePlatform` property can be used in code to set font properties per platform

```

Label label = new Label
{
    Text = "Different font properties on different platforms"
};

label.FontSize = Device.RuntimePlatform == Device.iOS ? 20 :
    Device.RuntimePlatform == Device.Android ? Device.GetNamedSize(NamedSize.Medium, label) : 24;
label.FontFamily = Device.RuntimePlatform == Device.iOS ? "MarkerFelt-Thin" :
    Device.RuntimePlatform == Device.Android ? "Lobster-Regular" : "ArimaMadurai-Black";

```

For more information about providing platform-specific values, see [Provide platform-specific values](#). For information about the `OnPlatform` markup extension, see [OnPlatform markup extension](#).

Understand named font sizes

Xamarin.Forms defines fields in the `NamedSize` enumeration that represent specific font sizes. The following table shows the `NamedSize` members, and their default sizes on iOS, Android, and the Universal Windows Platform (UWP):

MEMBER	IOS	ANDROID	UWP
Default	17	14	14
Micro	12	10	15.667
Small	14	14	18.667
Medium	17	17	22.667
Large	22	22	32
Body	17	16	14
Header	17	14	46
Title	28	24	24
Subtitle	22	16	20
Caption	12	12	12

The size values are measured in device-independent units. For more information, see [Units of Measurement](#).

NOTE

On iOS and Android, named font sizes will autoscale based on operating system accessibility options. This behavior can be disabled on iOS with a platform-specific. For more information, see [Accessibility Scaling for Named Font Sizes on iOS](#).

Display font icons

Font icons can be displayed by Xamarin.Forms applications by specifying the font icon data in a `FontImageSource` object. This class, which derives from the `ImageSource` class, has the following properties:

- `Glyph` – the unicode character value of the font icon, specified as a `string`.
- `Size` – a `double` value that indicates the size, in device-independent units, of the rendered font icon. The default value is 30. In addition, this property can be set to a named font size.
- `FontFamily` – a `string` representing the font family to which the font icon belongs.
- `Color` – an optional `color` value to be used when displaying the font icon.

This data is used to create a PNG, which can be displayed by any view that can display an `ImageSource`. This approach permits font icons, such as emojis, to be displayed by multiple views, as opposed to limiting font icon display to a single text presenting view, such as a `Label`.

IMPORTANT

Font icons can only currently be specified by their unicode character representation.

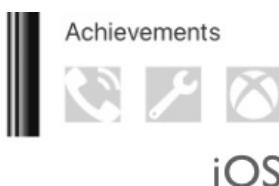
The following XAML example has a single font icon being displayed by an `Image` view:

```
<Image BackgroundColor="#D1D1D1">
    <Image.Source>
        <FontImageSource Glyph="" 
                        FontFamily="{OnPlatform iOS=Ionicons, Android=ionicons.ttf#}"
                        Size="44" />
    </Image.Source>
</Image>
```

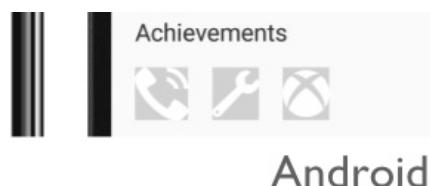
This code displays an XBox icon, from the Ionicons font family, in an `Image` view. Note that while the unicode character for this icon is `\uf30c`, it has to be escaped in XAML and so becomes ``. The equivalent C# code is:

```
Image image = new Image { BackgroundColor = Color.FromHex("#D1D1D1") };
image.Source = new FontImageSource
{
    Glyph = "\uf30c",
    FontFamily = Device.RuntimePlatform == Device.iOS ? "Ionicons" : "ionicons.ttf#",
    Size = 44
};
```

The following screenshots, from the [Bindable Layouts](#) sample, show several font icons being displayed by a bindable layout:



iOS



Android

Related links

- [FontsSample](#)
- [Text \(sample\)](#)
- [Bindable Layouts \(sample\)](#)
- [Provide platform-specific values](#)
- [OnPlatform markup extension](#)
- [Bindable Layouts](#)

SkiaSharp Graphics in Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

This guide assumes that you are familiar with Xamarin.Forms programming.

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, edit its **Info.plist** file 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 \(sample\)](#)

Xamarin.Forms splash screen

8/4/2022 • 2 minutes to read • [Edit Online](#)

Applications often have a startup delay while the application completes its initialization process. Developers may want to offer a branded experience, typically called a splash screen, while the application is starting. This article explains how to create splash screens for Xamarin.Forms applications.

Xamarin.Forms is initialized on each platform after the native startup sequence has completed. Xamarin.Forms is initialized:

- In the `OnCreate` method of the `MainActivity` class on Android.
- In the `FinishedLaunching` method of the `AppDelegate` class on iOS.
- In the `OnLaunched` method of the `App` class on UWP.

The splash screen should be shown as soon as possible when the application is launched, but Xamarin.Forms is not initialized until late in the startup sequence, which means that the splash screen must be implemented outside of Xamarin.Forms on each platform. The following sections explain how to create a splash screens on each platform.

Xamarin.Forms Android splash screen

Creating a splash screen on Android requires creating a splash `Activity` as the `MainLauncher` with a special theme. As soon as the splash `Activity` is started, it launches the main `Activity` with the normal application theme.

For more information about splash screens on Xamarin.Android, see [Xamarin.Android splash screen](#).

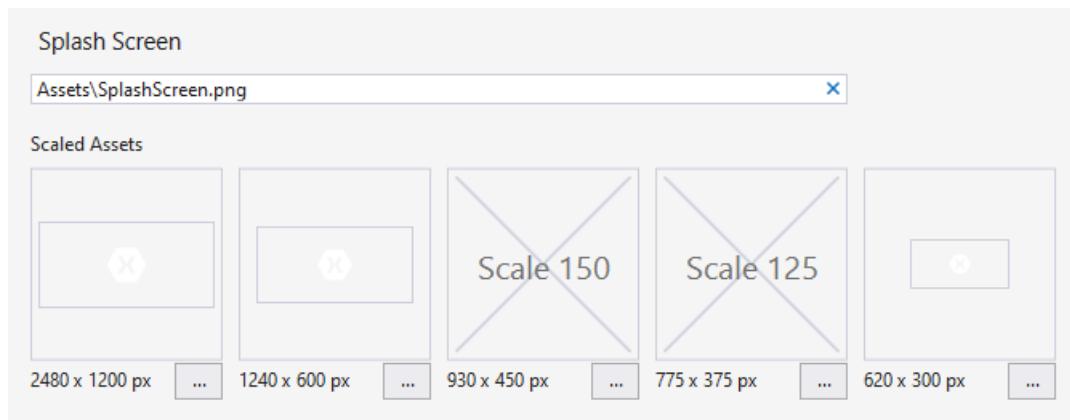
Xamarin.Forms iOS splash screen

A splash screen on iOS is referred to as a Launch Screen. Creating a Launch Screen on iOS requires creating a Storyboard that defines the UI of the launch screen, and then setting the Storyboard as the Launch Screen in the `Info.plist`.

For more information about Launch Screens on Xamarin.iOS, see [Xamarin.iOS Launch Screen](#).

Xamarin.Forms UWP splash screen

On UWP, the `Package.appxmanifest` contains a **Visual Assets** tab with a **Splash Screen** submenu. The splash screen graphics can be specified in this menu:



Related links

- [Xamarin.Android splash screen](#)
- [Xamarin.iOS Launch Screen](#)

Styling Xamarin.Forms Apps

8/4/2022 • 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

8/4/2022 • 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 setting 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.

Style Classes

Xamarin.Forms style classes enable multiple styles to be applied to a control, without resorting to style inheritance.

Introduction to Xamarin.Forms Styles

8/4/2022 • 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"
    IconImageSource="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#:

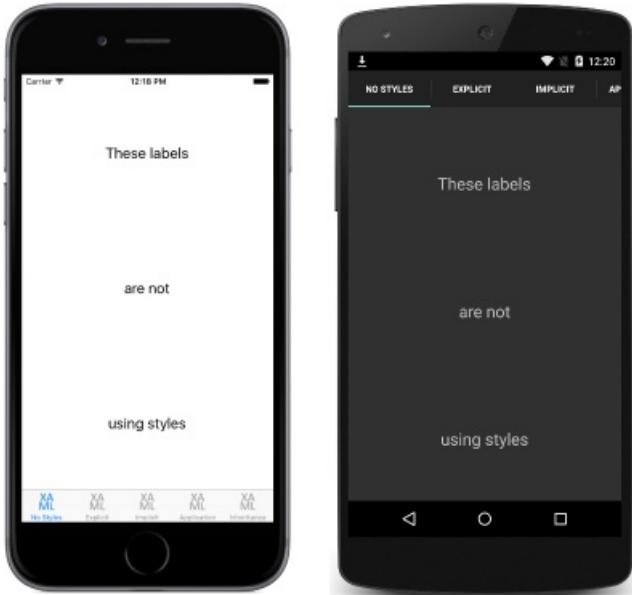
```

public class NoStylesPageCS : ContentPage
{
    public NoStylesPageCS ()
    {
        Title = "No Styles";
        IconImageSource = "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.

Create 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 introduced the `AutomationId` property, which has superseded the `StyleId` property.

Related links

- [XAML Markup Extensions](#)
- [Style](#)
- [Setter](#)

Explicit Styles in Xamarin.Forms

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

An *explicit style* is one that is selectively applied to controls by setting their `Style` properties.

Create an explicit style in XAML

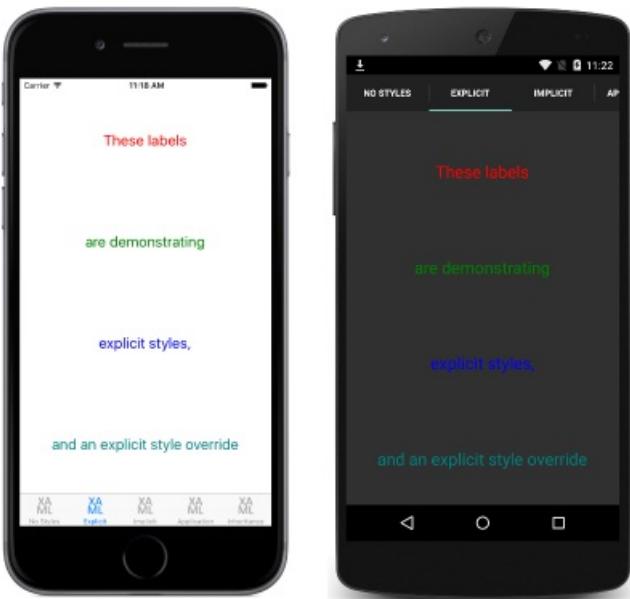
To declare a `Style` at the page level, a `ResourceDictionary` must be added to the page and then one or more `Style` declarations can be included in the `ResourceDictionary`. A `Style` is made *explicit* by giving its declaration an `x:Key` attribute, which gives it a descriptive key in the `ResourceDictionary`. *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"
    IconImageSource="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.

Create 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"
    IconImageSource="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](#).

Create 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.

Related links

- [XAML Markup Extensions](#)
- [Basic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

Implicit Styles in Xamarin.Forms

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

An *implicit style* is one that's used by all controls of the same `TargetType`, without requiring each control to reference the style.

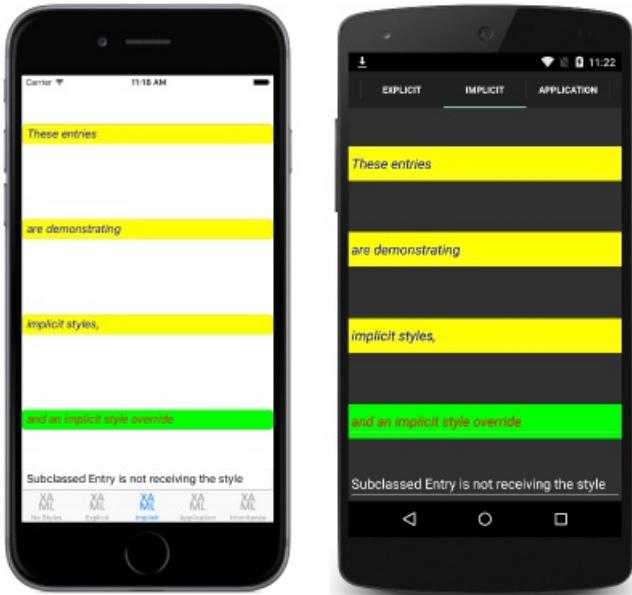
Create an implicit style in XAML

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" IconImageSource="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.

Create 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" IconImageSource="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](#).

Create 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`.

Apply a style to derived types

The `Style.ApplyToDerivedTypes` property enables a style to be applied to controls that are derived from the base type referenced by the `TargetType` property. Therefore, setting this property to `true` enables a single style to target multiple types, provided that the types derive from the base type specified in the `TargetType` property.

The following example shows an implicit style that sets the background color of `Button` instances to red:

```

<Style TargetType="Button"
       ApplyToDerivedTypes="True">
    <Setter Property="BackgroundColor"
           Value="Red" />
</Style>

```

Placing this style in a page-level `ResourceDictionary` will result in it being applied to all `Button` instances on the page, and also to any controls that derive from `Button`. However, if the `ApplyToDerivedTypes` property remained unset, the style would only be applied to `Button` instances.

The equivalent C# code is:

```
var buttonStyle = new Style(typeof(Button))
{
    ApplyToDerivedTypes = true,
    Setters =
    {
        new Setter
        {
            Property = VisualElement.BackgroundColorProperty,
            Value = Color.Red
        }
    }
};

Resources = new ResourceDictionary { buttonStyle };
```

Related links

- [XAML Markup Extensions](#)
- [Basic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

Global Styles in Xamarin.Forms

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

Create 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:

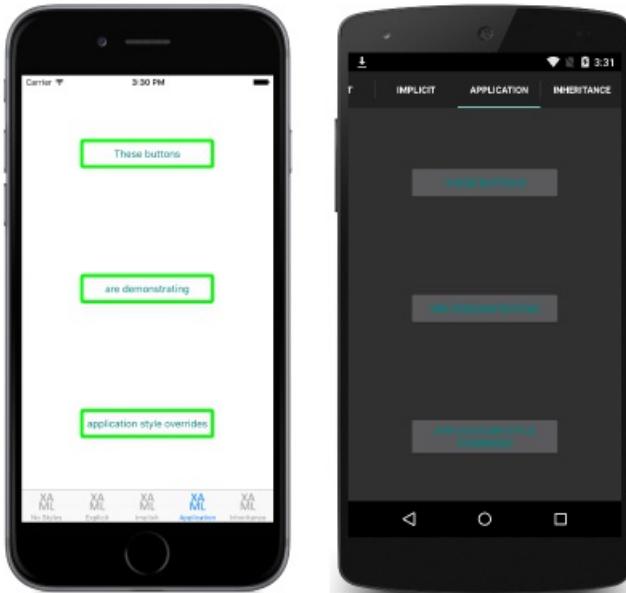
```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.App">
  <Application.Resources>
    <ResourceDictionary>
      <Style x:Key="buttonStyle" TargetType="Button">
        <Setter Property="HorizontalOptions" Value="Center" />
        <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" IconImageSource="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:



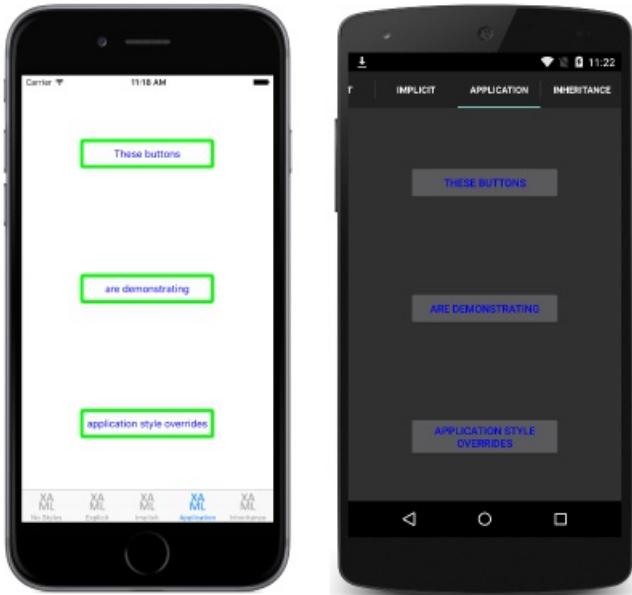
For information about creating styles in a page's [ResourceDictionary](#), see [Explicit Styles](#) and [Implicit Styles](#).

Override 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" IconImageSource="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:



Create 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.

Related links

- [XAML Markup Extensions](#)
- [Basic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

Style Inheritance in Xamarin.Forms

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

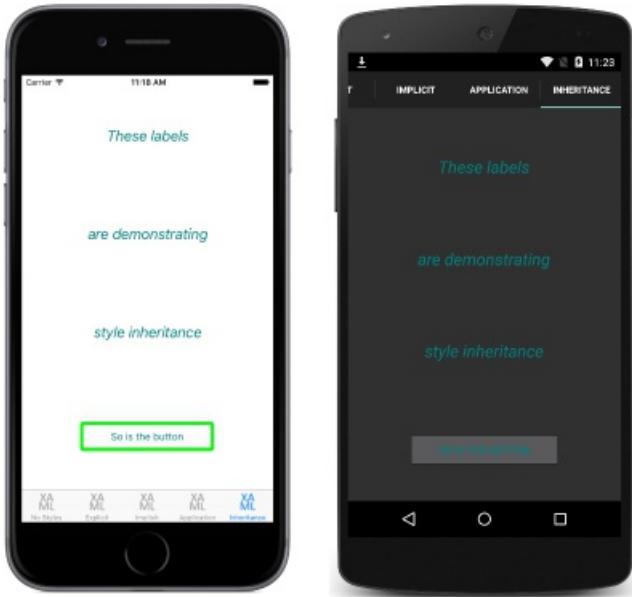
Styles that inherit from a base style can include `Setter` instances for new properties, or use them to override styles from the base style. In addition, styles that inherit from a base style must target the same type, or a type that derives from the type targeted by the base style. For example, if a 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" IconImageSource="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" IconImageSource="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.

Related links

- [XAML Markup Extensions](#)
- [Basic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

Dynamic Styles in Xamarin.Forms

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

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"
IconImageSource="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:

```

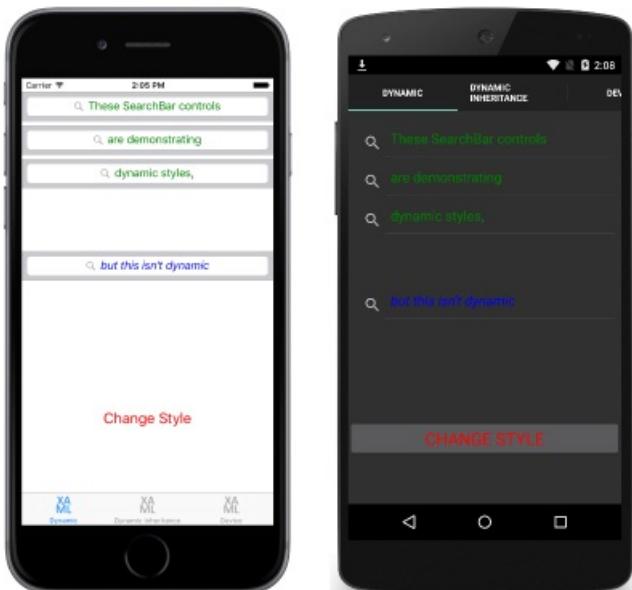
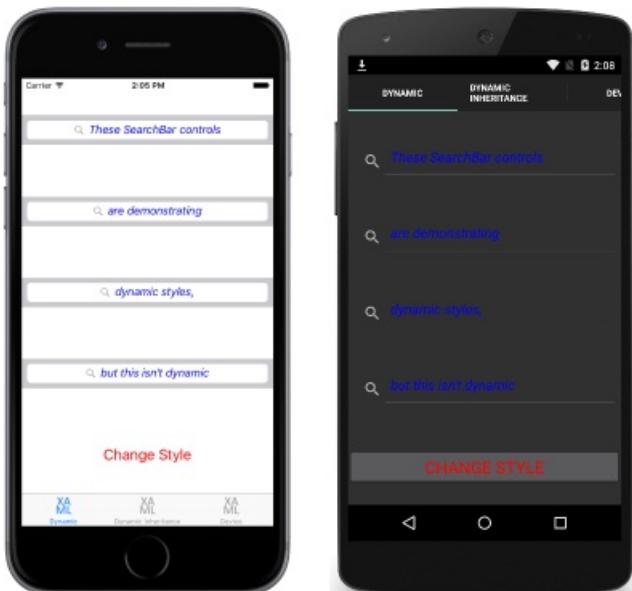
public partial class DynamicStylesPage : ContentPage
{
    bool originalStyle = true;

    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 `SearchBar` 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:

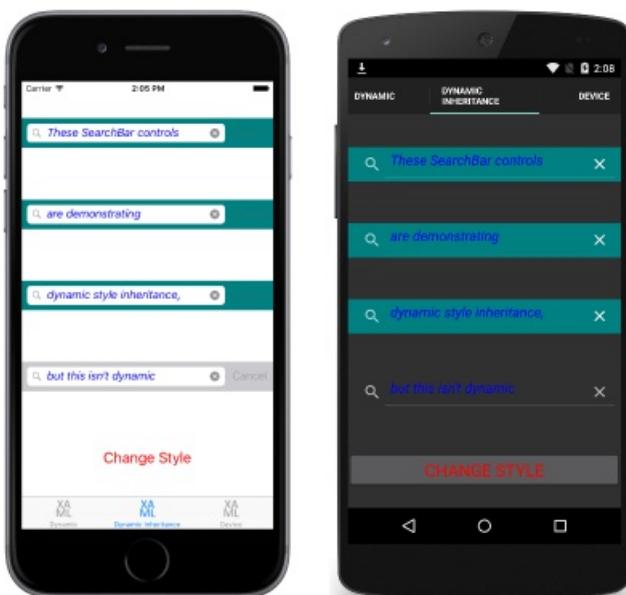
```

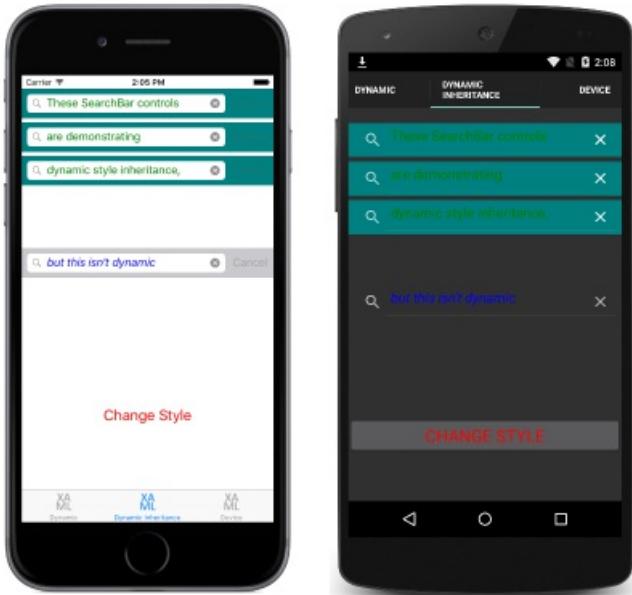
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.DynamicStylesInheritancePage"
    Title="Dynamic Inheritance" IconImageSource="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" 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 `SearchBar` 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.

Related links

- [XAML Markup Extensions](#)
- [Dynamic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

Related video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Device Styles in Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms includes six dynamic styles, known as device styles, in the `DeviceStyles` 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"
    IconImageSource="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";
        IconImageSource = "csharp.png";
        Padding = new Thickness (0, 20, 0, 0);

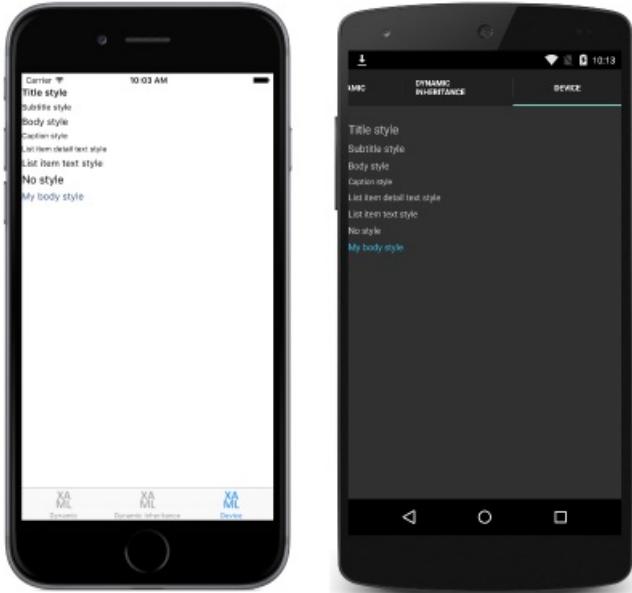
        Content = new StackLayout {
            Children = {
                new Label { Text = "Title style", Style = Device.Styles.TitleStyle },
                new Label { Text = "Subtitle style", Style = Device.Styles.SubtitleStyle },
                new Label { Text = "Body style", Style = Device.Styles.BodyStyle },
                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.

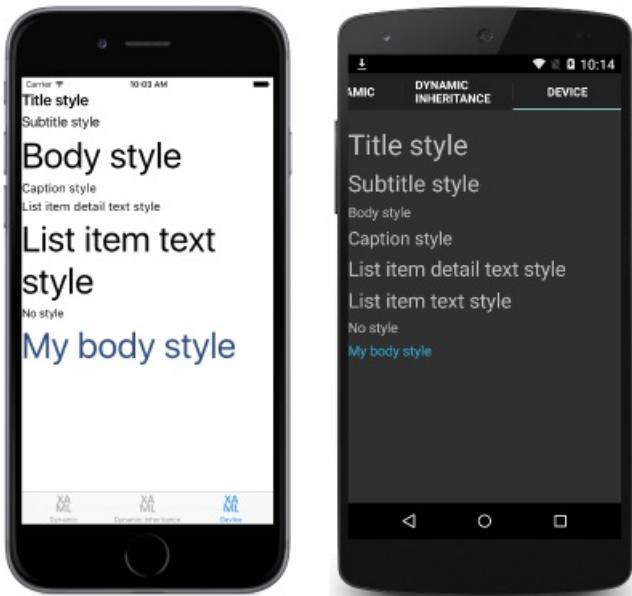
Accessibility

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:



Related links

- [Text Styles](#)
- [XAML Markup Extensions](#)
- [Dynamic Styles \(sample\)](#)
- [Working with Styles \(sample\)](#)
- [Device.Styles](#)
- [ResourceDictionary](#)
- [Style](#)
- [Setter](#)

Xamarin.Forms Style Classes

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms style classes enable multiple styles to be applied to a control, without resorting to style inheritance.

Create style classes

A style class can be created by setting the `Class` property on a `Style` to a `string` that represents the class name. The advantage this offers, over defining an explicit style using the `x:Key` attribute, is that multiple style classes can be applied to a `VisualElement`.

IMPORTANT

Multiple styles can share the same class name, provided they target different types. This enables multiple style classes, that are identically named, to target different types.

The following example shows three `BoxView` style classes, and a `VisualElement` style class:

```

<ContentPage ...>
    <ContentPage.Resources>
        <Style TargetType="BoxView"
            Class="Separator">
            <Setter Property="BackgroundColor"
                Value="#CCCCCC" />
            <Setter Property="HeightRequest"
                Value="1" />
        </Style>

        <Style TargetType="BoxView"
            Class="Rounded">
            <Setter Property="BackgroundColor"
                Value="#1FAECE" />
            <Setter Property="HorizontalOptions"
                Value="Start" />
            <Setter Property="CornerRadius"
                Value="10" />
        </Style>

        <Style TargetType="BoxView"
            Class="Circle">
            <Setter Property="BackgroundColor"
                Value="#1FAECE" />
            <Setter Property="WidthRequest"
                Value="100" />
            <Setter Property="HeightRequest"
                Value="100" />
            <Setter Property="HorizontalOptions"
                Value="Start" />
            <Setter Property="CornerRadius"
                Value="50" />
        </Style>

        <Style TargetType="VisualElement"
            Class="Rotated"
            ApplyToDerivedTypes="true">
            <Setter Property="Rotation"
                Value="45" />
        </Style>
    </ContentPage.Resources>
</ContentPage>

```

The `Separator`, `Rounded`, and `Circle` style classes each set `BoxView` properties to specific values.

The `Rotated` style class has a `TargetType` of `VisualElement`, which means it can only be applied to `VisualElement` instances. However, its `ApplyToDerivedTypes` property is set to `true`, which ensures that it can be applied to any controls that derive from `VisualElement`, such as `BoxView`. For more information about applying a style to a derived type, see [Apply a style to derived types](#).

The equivalent C# code is:

```

var separatorBoxViewStyle = new Style(typeof(BoxView))
{
    Class = "Separator",
    Setters =
    {
        new Setter
        {
            Property = VisualElement.BackgroundColorProperty,
            Value = Color.FromHex("#CCCCCC")
        },
        new Setter
        {
            Property = VisualElement.HeightRequestProperty,

```

```

        Value = 1
    }
}
};

var roundedBoxViewStyle = new Style(typeof(BoxView))
{
    Class = "Rounded",
    Setters =
    {
        new Setter
        {
            Property = VisualElement.BackgroundColorProperty,
            Value = Color.FromHex("#1FAECE")
        },
        new Setter
        {
            Property = View.HorizontalOptionsProperty,
            Value = LayoutOptions.Start
        },
        new Setter
        {
            Property = BoxView.CornerRadiusProperty,
            Value = 10
        }
    }
};

var circleBoxViewStyle = new Style(typeof(BoxView))
{
    Class = "Circle",
    Setters =
    {
        new Setter
        {
            Property = VisualElement.BackgroundColorProperty,
            Value = Color.FromHex("#1FAECE")
        },
        new Setter
        {
            Property = VisualElement.WidthRequestProperty,
            Value = 100
        },
        new Setter
        {
            Property = VisualElement.HeightRequestProperty,
            Value = 100
        },
        new Setter
        {
            Property = View.HorizontalOptionsProperty,
            Value = LayoutOptions.Start
        },
        new Setter
        {
            Property = BoxView.CornerRadiusProperty,
            Value = 50
        }
    }
};

var rotatedVisualElementStyle = new Style(typeof(VisualElement))
{
    Class = "Rotated",
    ApplyToDerivedTypes = true,
    Setters =
    {
        new Setter
        {

```

```

        Property = VisualElement.RotationProperty,
        Value = 45
    }
}

};

Resources = new ResourceDictionary
{
    separatorBoxViewStyle,
    roundedBoxViewStyle,
    circleBoxViewStyle,
    rotatedVisualElementStyle
};

```

Consume style classes

Style classes can be consumed by setting the `styleClass` property of the control, which is of type `IList<string>`, to a list of style class names. The style classes will be applied, provided that the type of the control matches the `TargetType` of the style classes.

The following example shows three `BoxView` instances, each set to different style classes:

```

<ContentPage ...>
    <ContentPage.Resources>
        ...
    </ContentPage.Resources>
    <StackLayout Margin="20">
        <BoxView StyleClass="Separator" />
        <BoxView WidthRequest="100"
            HeightRequest="100"
            HorizontalOptions="Center"
            StyleClass="Rounded, Rotated" />
        <BoxView HorizontalOptions="Center"
            StyleClass="Circle" />
    </StackLayout>
</ContentPage>

```

In this example, the first `BoxView` is styled to be a line separator, while the third `BoxView` is circular. The second `BoxView` has two style classes applied to it, which give it rounded corners and rotate it 45 degrees:



IMPORTANT

Multiple style classes can be applied to a control because the `StyleClass` property is of type `IList<string>`. When this occurs, style classes are applied in ascending list order. Therefore, when multiple style classes set identical properties, the property in the style class that's in the highest list position will take precedence.

The equivalent C# code is:

```
...
Content = new StackLayout
{
    Children =
    {
        new BoxView { StyleClass = new [] { "Separator" } },
        new BoxView { WidthRequest = 100, HeightRequest = 100, HorizontalOptions = LayoutOptions.Center,
StyleClass = new [] { "Rounded", "Rotated" } },
        new BoxView { HorizontalOptions = LayoutOptions.Center, StyleClass = new [] { "Circle" } }
    }
};
```

Related links

- [Basic Styles \(sample\)](#)

Styling Xamarin.Forms apps using Cascading Style Sheets (CSS)

8/4/2022 • 14 minutes to read • [Edit Online](#)



[Download the sample](#)

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;
}

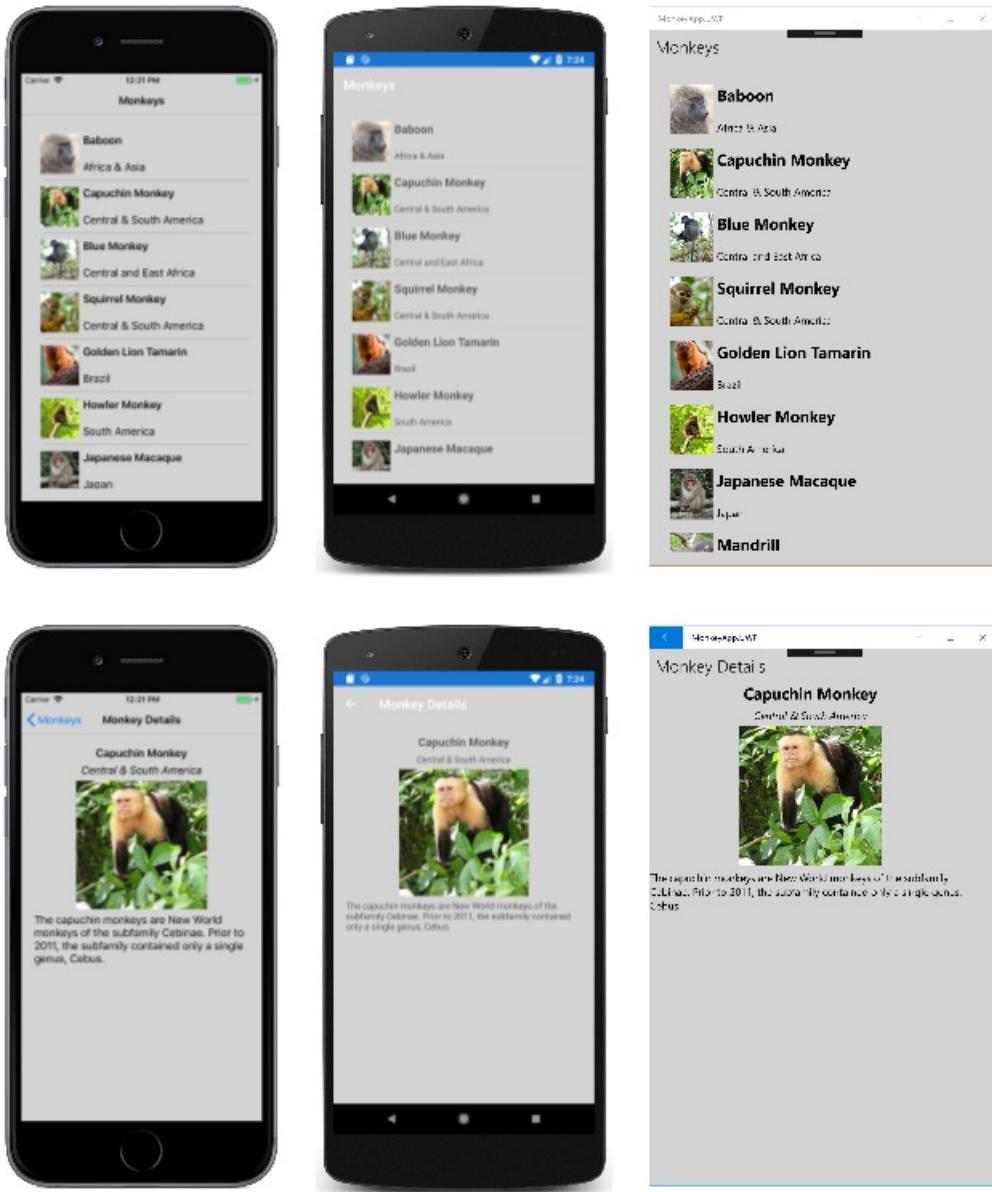
```

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.

NOTE

It's not currently possible to change a style sheet at runtime and have the new style sheet applied.

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 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](#).

Child stylesheets always override parent stylesheets if they set the same properties. Therefore, the following precedence rules are followed when applying styles that set the same properties:

- A style defined in the application resources will be overwritten by a style defined in the page resources, if they set the same properties.
- A style defined in page resources will be overwritten by a style defined in the control resources, if they set the same properties.
- A style defined in the application resources will be overwritten by a style defined in the control resources, if they set the same properties.

IMPORTANT

CSS variables are unsupported.

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 subclasses 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
.class	.header	Selects all elements with the <code>StyleClass</code> property containing 'header'. Note that this selector is case sensitive.
#id	#email	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.
*	*	Selects all elements.
element	label	Selects all elements of type <code>Label</code> , but not subclasses. Note that this selector is case insensitive.
^base	^contentpage	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.
element,element	label,button	Selects all <code>Button</code> elements and all <code>Label</code> elements. Note that this selector is case insensitive.
element element	stacklayout label	Selects all <code>Label</code> elements inside a <code>StackLayout</code> . Note that this selector is case insensitive.
element>element	stacklayout>label	Selects all <code>Label</code> elements with <code>StackLayout</code> as a direct parent. Note that this selector is case insensitive.
element+element	label+entry	Selects all <code>Entry</code> elements directly after a <code>Label</code> . Note that this selector is case insensitive.
element~element	label~entry	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>
<code>align-self</code>	<code>VisualElement</code>	<code>auto</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-self: flex-end;</code>
<code>background-color</code>	<code>VisualElement</code>	<code>color</code> <code>initial</code>	<code>background-color: springgreen;</code>
<code>background-image</code>	<code>Page</code>	<code>string</code> <code>initial</code>	<code>background-image: bg.png;</code>
<code>border-color</code>	<code>Button</code> , <code>Frame</code> , <code>ImageButton</code>	<code>color</code> <code>initial</code>	<code>border-color: #9acd32;</code>
<code>border-radius</code>	<code>BoxView</code> , <code>Button</code> , <code>Frame</code> , <code>ImageButton</code>	<code>double</code> <code>initial</code>	<code>border-radius: 10;</code>
<code>border-width</code>	<code>Button</code> , <code>ImageButton</code>	<code>double</code> <code>initial</code>	<code>border-width: .5;</code>

PROPERTY	APPLIES TO	VALUES	EXAMPLE
color	ActivityIndicator , BoxView , Button , CheckBox , 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;
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;

PROPERTY	APPLIES TO	VALUES	EXAMPLE
justify-content	FlexLayout	start center end spacebetween spacearound spaceevenly flex-start flex-end space-between space-around initial	justify-content: flex-end;
letter-spacing	Button , DatePicker , Editor , Entry , Label , Picker , SearchBar , SearchHandler , Span , TimePicker	double initial	letter-spacing: 2.5;
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;
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;

PROPERTY	APPLIES TO	VALUES	EXAMPLE
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;
text-transform	Button , Editor , Entry , Label , SearchBar , SearchHandler	none default uppercase lowercase initial	text-transform: uppercase;
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;

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`.

Targeting `Span` elements has a known issue preventing spans from being the target of CSS styles by both element and name (using the `#` symbol). The `Span` element derives from `GestureElement`, which does not have the `StyleClass` property so spans do not support CSS class targeting. For more information, see [Not able to apply CSS styling to Span control](#).

Xamarin.Forms specific properties

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
<code>-xf-bar-background-color</code>	<code>NavigationPage</code> , <code>TabbedPage</code>	<code>color</code> <code>initial</code>	<code>-xf-bar-background-color: teal;</code>
<code>-xf-bar-text-color</code>	<code>NavigationPage</code> , <code>TabbedPage</code>	<code>color</code> <code>initial</code>	<code>-xf-bar-text-color: gray</code>
<code>-xf-horizontal-scrollbar-visibility</code>	<code>ScrollView</code>	<code>default</code> <code>always</code> <code>never</code> <code>initial</code>	<code>-xf-horizontal-scrollbar-visibility: never;</code>
<code>-xf-max-length</code>	<code>Entry</code> , <code>Editor</code> , <code>SearchBar</code>	<code>int</code> <code>initial</code>	<code>-xf-max-length: 20;</code>
<code>-xf-max-track-color</code>	<code>Slider</code>	<code>color</code> <code>initial</code>	<code>-xf-max-track-color: red;</code>
<code>-xf-min-track-color</code>	<code>Slider</code>	<code>color</code> <code>initial</code>	<code>-xf-min-track-color: yellow;</code>
<code>-xf-orientation</code>	<code>ScrollView</code> , <code>StackLayout</code>	<code>horizontal</code> <code>vertical</code> <code>both</code> <code>initial</code> . <code>both</code> is only supported on a <code>ScrollView</code> .	<code>-xf-orientation: horizontal;</code>
<code>-xf-placeholder</code>	<code>Entry</code> , <code>Editor</code> , <code>SearchBar</code>	<code>quoted text</code> <code>initial</code>	<code>-xf-placeholder: Enter name;</code>
<code>-xf-placeholder-color</code>	<code>Entry</code> , <code>Editor</code> , <code>SearchBar</code>	<code>color</code> <code>initial</code>	<code>-xf-placeholder-color: green;</code>
<code>-xf-spacing</code>	<code>StackLayout</code>	<code>double</code> <code>initial</code>	<code>-xf-spacing: 8;</code>
<code>-xf-thumb-color</code>	<code>Slider</code> , <code>Switch</code>	<code>color</code> <code>initial</code>	<code>-xf-thumb-color: limegreen;</code>
<code>-xf-vertical-scrollbar-visibility</code>	<code>ScrollView</code>	<code>default</code> <code>always</code> <code>never</code> <code>initial</code>	<code>-xf-vertical-scrollbar-visibility: always;</code>
<code>-xf-vertical-text-alignment</code>	<code>Label</code>	<code>start</code> <code>center</code> <code>end</code> <code>initial</code>	<code>-xf-vertical-text-alignment: end;</code>

PROPERTY	APPLIES TO	VALUES	EXAMPLE
-xf-visual	VisualElement	string initial	-xf-visual: material;

Xamarin.Forms Shell specific properties

The following Xamarin.Forms Shell specific CSS properties are also supported (in the **Values** column, types are *italic*, while string literals are `gray`):

PROPERTY	APPLIES TO	VALUES	EXAMPLE
-xf-flyout-background	Shell	color initial	-xf-flyout-background: red;
-xf-shell-background	Element	color initial	-xf-shell-background: green;
-xf-shell-disabled	Element	color initial	-xf-shell-disabled: blue;
-xf-shell-foreground	Element	color initial	-xf-shell-foreground: yellow;
-xf-shell-tabbar-background	Element	color initial	-xf-shell-tabbar-background: white;
-xf-shell-tabbar-disabled	Element	color initial	-xf-shell-tabbar-disabled: black;
-xf-shell-tabbar-foreground	Element	color initial	-xf-shell-tabbar-foreground: gray;
-xf-shell-tabbar-title	Element	color initial	-xf-shell-tabbar-title: lightgray;
-xf-shell-tabbar-unselected	Element	color initial	-xf-shell-tabbar-unselected: cyan;
-xf-shell-title	Element	color initial	-xf-shell-title: teal;
-xf-shell-unselected	Element	color initial	-xf-shell-unselected: limegreen;

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` , `#rrggbba` , `#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.

Functions

Linear and radial gradients can be specified using the `linear-gradient()` and `radial-gradient()` CSS functions, respectively. The result of these functions should be assigned to the `background` property of a control.

CSS in Xamarin.Forms with Xamarin.University

[Xamarin.Forms 3.0 CSS video](#)

Related Links

- [MonkeyAppCSS \(sample\)](#)
- [Resource Dictionaries](#)
- [Styling Xamarin.Forms Apps using XAML Styles](#)

Theming a Xamarin.Forms application

8/4/2022 • 2 minutes to read • [Edit Online](#)

Theme an application

Theming can be implemented in Xamarin.Forms applications by creating a [ResourceDictionary](#) for each theme, and then loading the resources with the [DynamicResource](#) markup extension.

Respond to system theme changes

Devices typically include light and dark themes, which each refer to a broad set of appearance preferences that can be set at the operating system level. Applications should respect these system themes, and respond immediately when the system theme changes.

Theme a Xamarin.Forms Application

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms applications can respond to style changes dynamically at runtime by using the `DynamicResource` markup extension. This 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` markup extension performs a single dictionary lookup, the `DynamicResource` markup extension maintains a link to the dictionary key. Therefore, if the value associated with the key is replaced, the change is applied to the `VisualElement`. This enables runtime theming to be implemented in Xamarin.Forms applications.

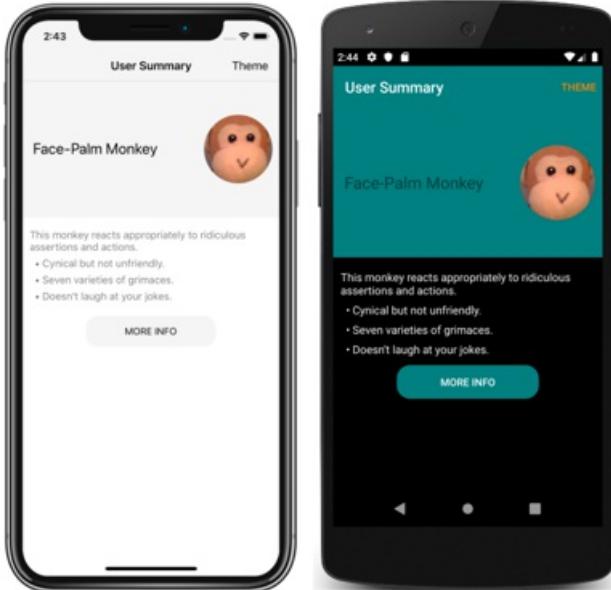
The process for implementing runtime theming in a Xamarin.Forms application is as follows:

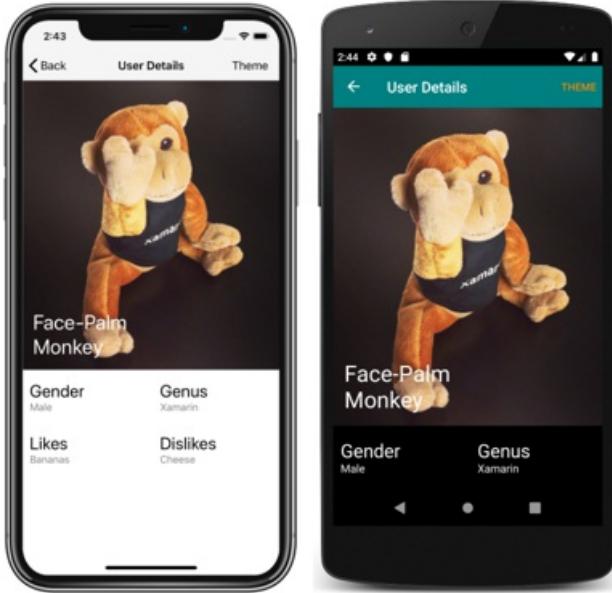
1. Define the resources for each theme in a `ResourceDictionary`.
2. Consume theme resources in the application, using the `DynamicResource` markup extension.
3. Set a default theme in the application's `App.xaml` file.
4. Add code to load a theme at runtime.

IMPORTANT

Use the `StaticResource` markup extension if you don't need to change the app theme at runtime.

The following screenshots show themed pages, with the iOS application using a light theme and the Android application using a dark theme:





NOTE

Changing a theme at runtime requires the use of XAML styles, and is not currently possible using CSS.

Define themes

A theme is defined as a collection of resource objects stored in a [ResourceDictionary](#).

The following example shows the [LightTheme](#) from the sample application:

```
<ResourceDictionary xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ThemingDemo.LightTheme">
    <Color x:Key="PageBackgroundColor">White</Color>
    <Color x:Key="NavigationBarColor">WhiteSmoke</Color>
    <Color x:Key="PrimaryColor">WhiteSmoke</Color>
    <Color x:Key="SecondaryColor">Black</Color>
    <Color x:Key="PrimaryTextColor">Black</Color>
    <Color x:Key="SecondaryTextColor">White</Color>
    <Color x:Key="TertiaryTextColor">Gray</Color>
    <Color x:Key="TransparentColor">Transparent</Color>
</ResourceDictionary>
```

The following example shows the [DarkTheme](#) from the sample application:

```
<ResourceDictionary xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ThemingDemo.DarkTheme">
    <Color x:Key="PageBackgroundColor">Black</Color>
    <Color x:Key="NavigationBarColor">Teal</Color>
    <Color x:Key="PrimaryColor">Teal</Color>
    <Color x:Key="SecondaryColor">White</Color>
    <Color x:Key="PrimaryTextColor">White</Color>
    <Color x:Key="SecondaryTextColor">White</Color>
    <Color x:Key="TertiaryTextColor">WhiteSmoke</Color>
    <Color x:Key="TransparentColor">Transparent</Color>
</ResourceDictionary>
```

Each [ResourceDictionary](#) contains [color](#) resources that define their respective themes, with each [ResourceDictionary](#) using identical key values. For more information about resource dictionaries, see [Resource](#)

IMPORTANT

A code behind file is required for each `ResourceDictionary`, which calls the `InitializeComponent` method. This is necessary so that a CLR object representing the chosen theme can be created at runtime.

Set a default theme

An application requires a default theme, so that controls have values for the resources they consume. A default theme can be set by merging the theme's `ResourceDictionary` into the application-level `ResourceDictionary` that's defined in `App.xaml`:

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ThemingDemo.App">
    <Application.Resources>
        <ResourceDictionary Source="Themes/LightTheme.xaml" />
    </Application.Resources>
</Application>
```

For more information about merging resource dictionaries, see [Merged resource dictionaries](#).

Consume theme resources

When an application wants to consume a resource that's stored in a `ResourceDictionary` that represents a theme, it should do so with the `DynamicResource` markup extension. This ensures that if a different theme is selected at runtime, the values from the new theme will be applied.

The following example shows three styles from the sample application that can be applied to `Label` objects:

```

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

    <Application.Resources>

        <Style x:Key="LargeLabelStyle"
               TargetType="Label">
            <Setter Property="TextColor"
                   Value="{DynamicResource SecondaryTextColor}" />
            <Setter Property="FontSize"
                   Value="30" />
        </Style>

        <Style x:Key="MediumLabelStyle"
               TargetType="Label">
            <Setter Property="TextColor"
                   Value="{DynamicResource PrimaryTextColor}" />
            <Setter Property="FontSize"
                   Value="25" />
        </Style>

        <Style x:Key="SmallLabelStyle"
               TargetType="Label">
            <Setter Property="TextColor"
                   Value="{DynamicResource TertiaryTextColor}" />
            <Setter Property="FontSize"
                   Value="15" />
        </Style>
    </Application.Resources>
</Application>

```

These styles are defined in the application-level resource dictionary, so that they can be consumed by multiple pages. Each style consumes theme resources with the `DynamicResource` markup extension.

These styles are then consumed by pages:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:ThemingDemo"
    x:Class="ThemingDemo.UserSummaryPage"
    Title="User Summary"
    BackgroundColor="{DynamicResource PageBackgroundColor}">
    ...
    <ScrollView>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="200" />
                <RowDefinition Height="120" />
                <RowDefinition Height="70" />
            </Grid.RowDefinitions>
            <Grid BackgroundColor="{DynamicResource PrimaryColor}">
                <Label Text="Face-Palm Monkey"
                    VerticalOptions="Center"
                    Margin="15"
                    Style="{StaticResource MediumLabelStyle}" />
                ...
            </Grid>
            <StackLayout Grid.Row="1"
                Margin="10">
                <Label Text="This monkey reacts appropriately to ridiculous assertions and actions."
                    Style="{StaticResource SmallLabelStyle}" />
                <Label Text=" &#x2022; Cynical but not unfriendly."
                    Style="{StaticResource SmallLabelStyle}" />
                <Label Text=" &#x2022; Seven varieties of grimaces."
                    Style="{StaticResource SmallLabelStyle}" />
                <Label Text=" &#x2022; Doesn't laugh at your jokes."
                    Style="{StaticResource SmallLabelStyle}" />
            </StackLayout>
            ...
        </Grid>
    </ScrollView>
</ContentPage>

```

When a theme resource is consumed directly, it should be consumed with the `DynamicResource` markup extension. However, when a style that uses the `DynamicResource` markup extension is consumed, it should be consumed with the `StaticResource` markup extension.

For more information about styling, see [Styling Xamarin.Forms Apps using XAML Styles](#). For more information about the `DynamicResource` markup extension, see [Dynamic Styles in Xamarin.Forms](#).

Load a theme at runtime

When a theme is selected at runtime, the application should:

1. Remove the current theme from the application. This is achieved by clearing the `MergedDictionaries` property of the application-level `ResourceDictionary`.
2. Load the selected theme. This is achieved by adding an instance of the selected theme to the `MergedDictionaries` property of the application-level `ResourceDictionary`.

Any `visualElement` objects that set properties with the `DynamicResource` markup extension will then apply the new theme values. This occurs because the `DynamicResource` markup extension maintains a link to dictionary keys. Therefore, when the values associated with keys are replaced, the changes are applied to the `VisualElement` objects.

In the sample application, a theme is selected via a modal page that contains a `Picker`. The following code shows the `OnPickerSelectionChanged` method, which is executed when the selected theme changes:

```
void OnPickerSelectionChanged(object sender, EventArgs e)
{
    Picker picker = sender as Picker;
    Theme theme = (Theme)picker.SelectedItem;

    ICollection<ResourceDictionary> mergedDictionaries = Application.Current.Resources.MergedDictionaries;
    if (mergedDictionaries != null)
    {
        mergedDictionaries.Clear();

        switch (theme)
        {
            case Theme.Dark:
                mergedDictionaries.Add(new DarkTheme());
                break;
            case Theme.Light:
            default:
                mergedDictionaries.Add(new LightTheme());
                break;
        }
    }
}
```

Related links

- [Theming \(sample\)](#)
- [Respond to system theme changes](#)
- [Resource Dictionaries](#)
- [Dynamic Styles in Xamarin.Forms](#)
- [Styling Xamarin.Forms Apps using XAML Styles](#)

Respond to system theme changes in Xamarin.Forms applications

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

Devices typically include light and dark themes, which each refer to a broad set of appearance preferences that can be set at the operating system level. Applications should respect these system themes, and respond immediately when the system theme changes.

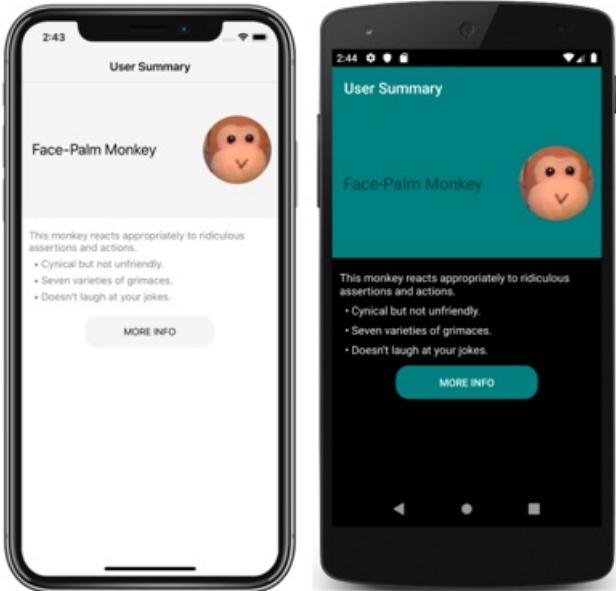
The system theme may change for a variety of reasons, depending on the device configuration. This includes the system theme being explicitly changed by the user, it changing due to the time of day, and it changing due to environmental factors such as low light.

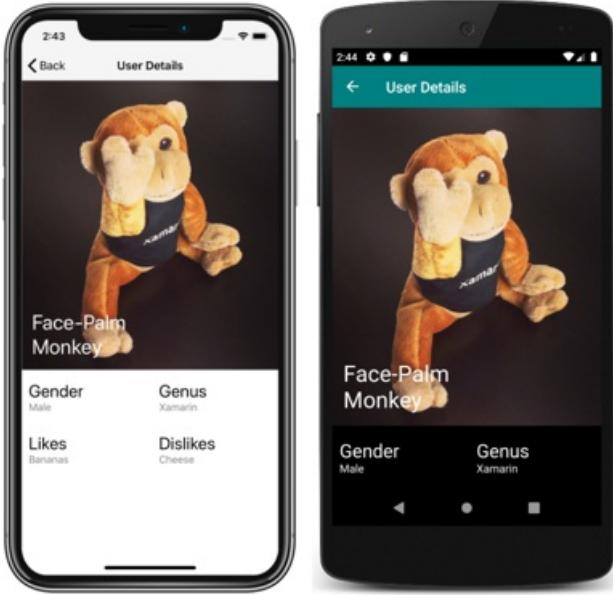
Xamarin.Forms applications can respond to system theme changes by consuming resources with the `AppThemeBinding` markup extension, and the `SetAppThemeColor` and `SetOnAppTheme<T>` extension methods.

The following requirements must be met for Xamarin.Forms to respond to a system theme change:

- Xamarin.Forms 4.6.0.967 or greater.
- iOS 13 or greater.
- Android 10 (API 29) or greater.
- UWP build 14393 or greater.
- macOS 10.14 or greater.

The following screenshots show themed pages, for light and dark system themes on iOS and Android:





Define and consume theme resources

Resources for light and dark themes can be consumed with the `AppThemeBinding` markup extension, and the `SetAppThemeColor` and `SetOnAppTheme<T>` extension methods. With these approaches, resources are automatically applied based on the value of the current system theme. In addition, objects that consume these resources are automatically updated if the system theme changes while an app is running.

AppThemeBinding markup extension

The `AppThemeBinding` markup extension enables you to consume a resource, such as an image or color, based on the current system theme:

```
<ContentPage ...>
    <StackLayout Margin="20">
        <Label Text="This text is green in light mode, and red in dark mode."
              TextColor="{AppThemeBinding Light=Green, Dark=Red}" />
        <Image Source="{AppThemeBinding Light=lightlogo.png, Dark=darklogo.png}" />
    </StackLayout>
</ContentPage>
```

In this example, the text color of the first `Label` is set to green when the device is using its light theme, and is set to red when the device is using its dark theme. Similarly, the `Image` displays a different image file based upon the current system theme.

In addition, resources defined in a `ResourceDictionary` can be consumed with the `StaticResource` markup extension:

```

<ContentPage ...>
    <ContentPage.Resources>

        <!-- Light colors -->
        <Color x:Key="LightPrimaryColor">WhiteSmoke</Color>
        <Color x:Key="LightSecondaryColor">Black</Color>

        <!-- Dark colors -->
        <Color x:Key="DarkPrimaryColor">Teal</Color>
        <Color x:Key="DarkSecondaryColor">White</Color>

        <Style x:Key="ButtonStyle"
            TargetType="Button">
            <Setter Property="BackgroundColor"
                Value="{AppThemeBinding Light={StaticResource LightPrimaryColor}, Dark={StaticResource
DarkPrimaryColor}}" />
            <Setter Property="TextColor"
                Value="{AppThemeBinding Light={StaticResource LightSecondaryColor}, Dark={StaticResource
DarkSecondaryColor}}" />
        </Style>

    </ContentPage.Resources>

    <Grid BackgroundColor="{AppThemeBinding Light={StaticResource LightPrimaryColor}, Dark={StaticResource
DarkPrimaryColor}}">
        <Button Text="MORE INFO"
            Style="{StaticResource ButtonStyle}" />
    </Grid>
</ContentPage>

```

In this example, the background color of the `Grid` and the `Button` style changes based on whether the device is using its light theme or dark theme.

For more information about the `AppThemeBinding` markup extension, see [AppThemeBinding markup extension](#).

Extension methods

Xamarin.Forms includes `SetAppThemeColor` and `SetOnAppTheme<T>` extension methods that enable `VisualElement` objects to respond to system theme changes.

The `SetAppThemeColor` method enables `Color` objects to be specified that will be set on a target property based on the current system theme:

```

Label label = new Label();
label.SetAppThemeColor(Label.TextColorProperty, Color.Green, Color.Red);

```

In this example, the text color of the `Label` is set to green when the device is using its light theme, and is set to red when the device is using its dark theme.

The `SetOnAppTheme<T>` method enables objects of type `T` to be specified that will be set on a target property based on the current system theme:

```

Image image = new Image();
image.SetOnAppTheme<FileImageSource>(Image.SourceProperty, "lightlogo.png", "darklogo.png");

```

In this example, the `Image` displays `lightlogo.png` when the device is using its light theme, and `darklogo.png` when the device is using its dark theme.

Detect the current system theme

The current system theme can be detected by getting the value of the `Application.RequestedTheme` property:

```
OSAppTheme currentTheme = Application.Current.RequestedTheme;
```

The `RequestedTheme` property returns an `OSAppTheme` enumeration member. The `OSAppTheme` enumeration defines the following members:

- `Unspecified`, which indicates that the device is using an unspecified theme.
- `Light`, which indicates that the device is using its light theme.
- `Dark`, which indicates that the device is using its dark theme.

Set the current user theme

The theme used by the application can be set with the `Application.UserAppTheme` property, which is of type `OSAppTheme`, regardless of which system theme is currently operational:

```
Application.Current.UserAppTheme = OSAppTheme.Dark;
```

In this example, the application is set to use the theme defined for the system dark mode, regardless of which system theme is currently operational.

NOTE

Set the `UserAppTheme` property to `OSAppTheme.Unspecified` to default to the operational system theme.

React to theme changes

The system theme on a device may change for a variety of reasons, depending on how the device is configured. Xamarin.Forms apps can be notified when the system theme changes by handling the `Application.RequestedThemeChanged` event:

```
Application.Current.RequestedThemeChanged += (s, a) =>
{
    // Respond to the theme change
};
```

The `AppThemeChangedEventArgs` object, which accompanies the `RequestedThemeChanged` event, has a single property named `RequestedTheme`, of type `OSAppTheme`. This property can be examined to detect the requested system theme.

IMPORTANT

To respond to theme changes on Android you must include the `ConfigChanges.UiMode` flag in the `Activity` attribute of your `MainActivity` class.

Related links

- [SystemThemes \(sample\)](#)
- [AppThemeBinding markup extension](#)
- [Resource Dictionaries](#)

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

Xamarin.Forms Visual

8/4/2022 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms Material Visual

Xamarin.Forms Material Visual can be used to create Xamarin.Forms applications that look identical, or largely identical, on iOS and Android.

Create a Xamarin.Forms Visual Renderer

Xamarin.Forms Visual enables renderers to be selectively applied to `VisualElement` objects, without having to subclass Xamarin.Forms views.

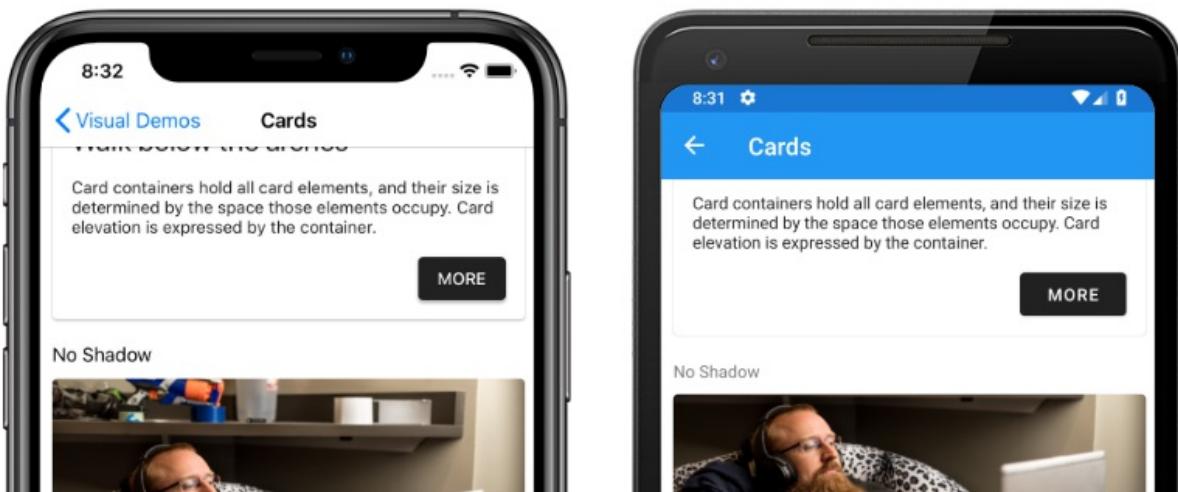
Xamarin.Forms Material Visual

8/4/2022 • 4 minutes to read • [Edit Online](#)

 [Download the sample](#)

Material Design is an opinionated design system created by Google, that prescribes the size, color, spacing, and other aspects of how views and layouts should look and behave.

Xamarin.Forms Material Visual can be used to apply Material Design rules to Xamarin.Forms applications, creating applications that look largely identical on iOS and Android. When Material Visual is enabled, supported views adopt the same design cross-platform, creating a unified look and feel.



The process for enabling Xamarin.Forms Material Visual in your application is:

1. Add the [Xamarin.Forms.Visual.Material](#) NuGet package to your iOS and Android platform projects. This NuGet package delivers optimized Material Design renderers on iOS and Android. On iOS, the package provides the transitive dependency to [Xamarin.iOS.MaterialComponents](#), which is a C# binding to Google's [Material Components for iOS](#). On Android, the package provides build targets to ensure that your TargetFramework is correctly set up.
2. Initialize Material Visual in each platform project. For more information, see [Initialize Material Visual](#).
3. Create Material Visual controls by setting the `Visual` property to `Material` on any pages that should adopt the Material Design rules. For more information, see [Consume Material renderers](#).
4. [optional] Customize Material controls. For more information, see [Customize Material controls](#).

IMPORTANT

On Android, Material Visual requires a minimum version of 5.0 (API 21) or greater, and a TargetFramework of version 9.0 (API 28). In addition, your platform project requires Android support libraries 28.0.0 or greater, and its theme needs to inherit from a Material Components theme or continue to inherit from an AppCompat theme. For more information, see [Getting started with Material Components for Android](#).

Material Visual currently supports the following controls:

- [ActivityIndicator](#)
- [Button](#)
- [CheckBox](#)

- `DatePicker`
- `Editor`
- `Entry`
- `Frame`
- `Picker`
- `ProgressBar`
- `Slider`
- `Stepper`
- `TimePicker`

Material controls are realized by Material renderers, which apply the Material Design rules. Functionally, Material renderers are no different to the default renderers. For more information, see [Customize Material Visual](#).

Initialize Material Visual

After installing the `Xamarin.Forms.Visual.Material` NuGet package, the Material renderers must be initialized in each platform project.

On iOS, this should occur in `AppDelegate.cs` by invoking the `Xamarin.Forms.FormsMaterial.Init` method *after* the `Xamarin.Forms.Forms.Init` method:

```
global::Xamarin.Forms.Forms.Init();
global::Xamarin.Forms.FormsMaterial.Init();
```

On Android, this should occur in `MainActivity.cs` by invoking the `Xamarin.Forms.FormsMaterial.Init` method *after* the `Xamarin.Forms.Forms.Init` method:

```
global::Xamarin.Forms.Forms.Init(this, savedInstanceState);
global::Xamarin.Forms.FormsMaterial.Init(this, savedInstanceState);
```

Apply Material Visual

Applications can enable Material Visual by setting the `VisualElement.Visual` property on a page, layout, or view, to `Material`:

```
<ContentPage Visual="Material"
    ...
    ...
/>
```

The equivalent C# code is:

```
ContentPage contentPage = new ContentPage();
contentPage.Visual = VisualMarker.Material;
```

Setting the `VisualElement.Visual` property to `Material` directs your application to use the Material Visual renderers instead of the default renderers. The `Visual` property can be set to any type that implements `IVisual`, with the `VisualMarker` class providing the following `IVisual` properties:

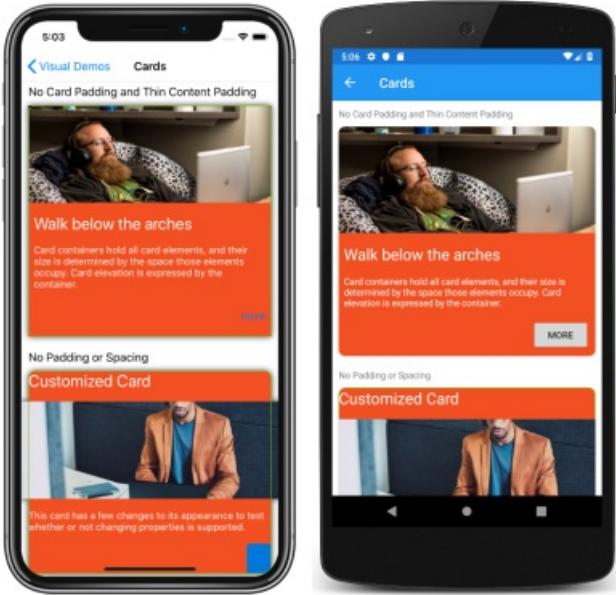
- `Default` – indicates that the view should render using the default renderer.
- `MatchParent` – indicates that the view should use the same renderer as its direct parent.

- **Material** – indicates that the view should render using a Material renderer.

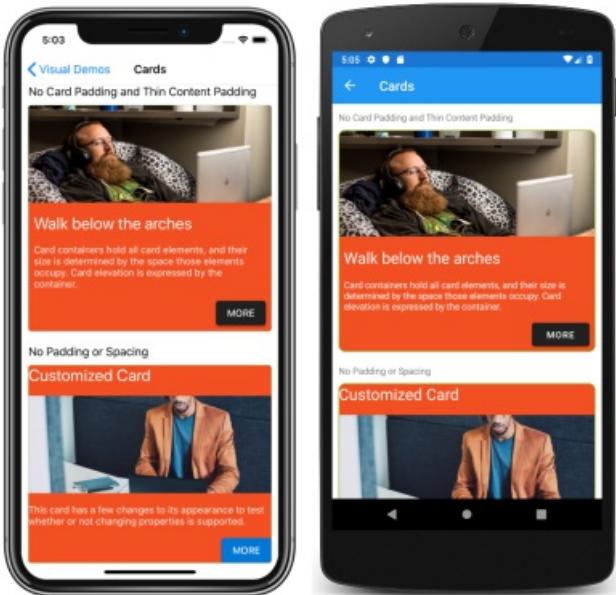
IMPORTANT

The `Visual` property is defined in the `VisualElement` class, with views inheriting the `Visual` property value from their parents. Therefore, setting the `Visual` property on a `ContentPage` ensures that any supported views in the page will use that Visual. In addition, the `Visual` property can be overridden on a view.

The following screenshots show a user interface rendered using the default renderers:



The following screenshots show the same user interface rendered using the Material renderers:



The main visible differences between the default renderers and Material renderers, shown here, are that the Material renderers capitalize `Button` text, and round the corners of `Frame` borders. However, Material renderers use native controls, and therefore there may still be user interface differences between platforms for areas such as fonts, shadows, colors, and elevation.

NOTE

Material Design components adhere closely to Google's guidelines. As a result, Material Design renderers are biased towards that sizing and behavior. When you require greater control of styles or behavior, you can still create your own [Effect](#), [Behavior](#), or [Custom Renderer](#) to achieve the detail you require.

Customize Material Visual

The Material Visual NuGet package is a collection of renderers that realize the Xamarin.Forms controls. Customizing Material Visual controls is identical to customizing default controls.

Effects are the recommended technique when the goal is to customize an existing control. If a Material Visual renderer exists, it is less work to customize the control with an effect than it is to subclass the renderer. For more information about effects see [Xamarin.Forms effects](#).

Custom renderers are the recommended technique when a Material renderer does not exist. The following renderer classes are included with Material Visual:

- [MaterialButtonRenderer](#)
- [MaterialCheckBoxRenderer](#)
- [MaterialEntryRenderer](#)
- [MaterialFrameRenderer](#)
- [MaterialProgressBarRenderer](#)
- [MaterialDatePickerRenderer](#)
- [MaterialTimePickerRenderer](#)
- [MaterialPickerRenderer](#)
- [MaterialActivityIndicatorRenderer](#)
- [MaterialEditorRenderer](#)
- [MaterialSliderRenderer](#)
- [MaterialStepperRenderer](#)

Subclassing a Material renderer is almost identical to non-Material renderers. However, when exporting a renderer that subclasses a Material renderer, you must provide a third argument to the `ExportRenderer` attribute that specifies the `VisualMarker.MaterialVisual` type:

```
using Xamarin.Forms.Material.Android;

[assembly: ExportRenderer(typeof(ProgressBar), typeof(CustomMaterialProgressBarRenderer), new[] {
    typeof(VisualMarker.MaterialVisual) })]
namespace MyApp.Android
{
    public class CustomMaterialProgressBarRenderer : MaterialProgressBarRenderer
    {
        //...
    }
}
```

In this example, the `ExportRendererAttribute` specifies that the `CustomMaterialProgressBarRenderer` class will be used to render the `ProgressBar` view, with the `IVisual` type registered as the third argument.

NOTE

A renderer that specifies an `IVisual` type, as part of its `ExportRendererAttribute`, will be used to render opted in views, rather than the default renderer. At renderer selection time, the `Visual` property of the view is inspected and included in the renderer selection process.

For more information about custom renderers, see [Custom Renderers](#).

Related links

- [Material Visual \(sample\)](#)
- [Create a Xamarin.Forms Visual Renderer](#)
- [Xamarin.Forms Effects](#)
- [Custom Renderers](#)

Create a Xamarin.Forms Visual Renderer

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms Visual enables renderers to be created and selectively applied to `VisualElement` objects, without having to subclass Xamarin.Forms views. A renderer that specifies an `IVisual` type, as part of its `ExportRendererAttribute`, will be used to render opted in views, rather than the default renderer. At renderer selection time, the `Visual` property of the view is inspected and included in the renderer selection process.

IMPORTANT

Currently the `visual` property cannot be changed after the view has been rendered, but this will change in a future release.

The process for creating and consuming a Xamarin.Forms Visual renderer is:

1. Create platform renderers for the required view. For more information, see [Create renderers](#).
2. Create a type that derives from `IVisual`. For more information, see [Create an IVisual type](#).
3. Register the `IVisual` type as part of the `ExportRendererAttribute` that decorates the renderers. For more information, see [Register the IVisual type](#).
4. Consume the Visual renderer by setting the `Visual` property on the view to the `IVisual` name. For more information, see [Consume the Visual renderer](#).
5. [optional] Register a name for the `IVisual` type. For more information, see [Register a name for the IVisual type](#).

Create platform renderers

For information about creating a renderer class, see [Custom Renderers](#). However, note that a Xamarin.Forms Visual renderer is applied to a view without having to subclass the view.

The renderer classes outlined here implement a custom `Button` that displays its text with a shadow.

iOS

The following code example shows the button renderer for iOS:

```
public class CustomButtonRenderer : ButtonRenderer
{
    protected override void OnElementChanged(ElementChangedEventArgs<Button> e)
    {
        base.OnElementChanged(e);

        if (e.OldElement != null)
        {
            // Cleanup
        }

        if (e.NewElement != null)
        {
            Control.TitleShadowOffset = new CoreGraphics.CGSize(1, 1);
            Control.SetTitleShadowColor(Color.Black.ToUIColor(), UIKit.UIControlState.Normal);
        }
    }
}
```

Android

The following code example shows the button renderer for Android:

```
public class CustomButtonRenderer : Xamarin.Forms.Platform.Android.AppCompat.ButtonRenderer
{
    public CustomButtonRenderer(Context context) : base(context)
    {

    }

    protected override void OnElementChanged(ElementChangedEventArgs<Button> e)
    {
        base.OnElementChanged(e);

        if (e.OldElement != null)
        {
            // Cleanup
        }

        if (e.NewElement != null)
        {
            Control.SetShadowLayer(5, 3, 3, Color.Black.ToAndroid());
        }
    }
}
```

Create an IVisual type

In your cross-platform library, create a type that derives from `IVisual`:

```
public class CustomVisual : IVisual
{
}
```

The `CustomVisual` type can then be registered against the renderer classes, permitting `Button` objects to opt into using the renderers.

Register the IVisual type

In the platform projects, add the `ExportRendererAttribute` at the assembly level:

```
[assembly: ExportRenderer(typeof(Xamarin.Forms.Button), typeof(CustomButtonRenderer), new[] {
    typeof(CustomVisual) })]
namespace VisualDemos.iOS
{
    public class CustomButtonRenderer : ButtonRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Button> e)
        {
            // ...
        }
    }
}
```

In this example for the iOS platform project, the `ExportRendererAttribute` specifies that the `CustomButtonRenderer` class will be used to render consuming `Button` objects, with the `IVisual` type registered as the third argument. A renderer that specifies an `IVisual` type, as part of its `ExportRendererAttribute`, will be used to render opted in views, rather than the default renderer.

Consume the Visual renderer

A `Button` object can opt into using the renderer classes by setting its `Visual` property to `Custom`:

```
<Button Visual="Custom"
        Text="CUSTOM BUTTON"
        BackgroundColor="{StaticResource PrimaryColor}"
        TextColor="{StaticResource SecondaryTextColor}"
        HorizontalOptions="FillAndExpand" />
```

NOTE

In XAML, a type converter removes the need to include the "Visual" suffix in the `Visual` property value. However, the full type name can also be specified.

The equivalent C# code is:

```
Button button = new Button { Text = "CUSTOM BUTTON", ... };
button.Visual = new CustomVisual();
```

At renderer selection time, the `Visual` property of the `Button` is inspected and included in the renderer selection process. If a renderer isn't located, the Xamarin.Forms default renderer will be used.

The following screenshots show the rendered `Button`, which displays its text with a shadow:



Register a name for the IVisual type

The `visualAttribute` can be used to optionally register a different name for the `IVisual` type. This approach can be used to resolve naming conflicts between different Visual libraries, or in situations where you just want to refer to a Visual by a different name than its type name.

The `visualAttribute` should be defined at the assembly level in either the cross-platform library, or in the platform project:

```
[assembly: Visual("MyVisual", typeof(CustomVisual))]
```

The `IVisual` type can then be consumed through its registered name:

```
<Button Visual="MyVisual"
       ... />
```

NOTE

When consuming a Visual through its registered name, any "Visual" suffix must be included.

Related links

- [Material Visual \(sample\)](#)
- [Xamarin.Forms Material Visual](#)
- [Custom Renderers](#)

Xamarin.Forms Visual State Manager

8/4/2022 • 18 minutes to read • [Edit Online](#)



[Download the sample](#)

Use the Visual State Manager to make changes to XAML elements based on visual states set from code.

The Visual State Manager (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 the following visual states:

- "Normal"
- "Disabled"
- "Focused"
- "Selected"

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.

Common states

The Visual State Manager allows you to include markup 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>

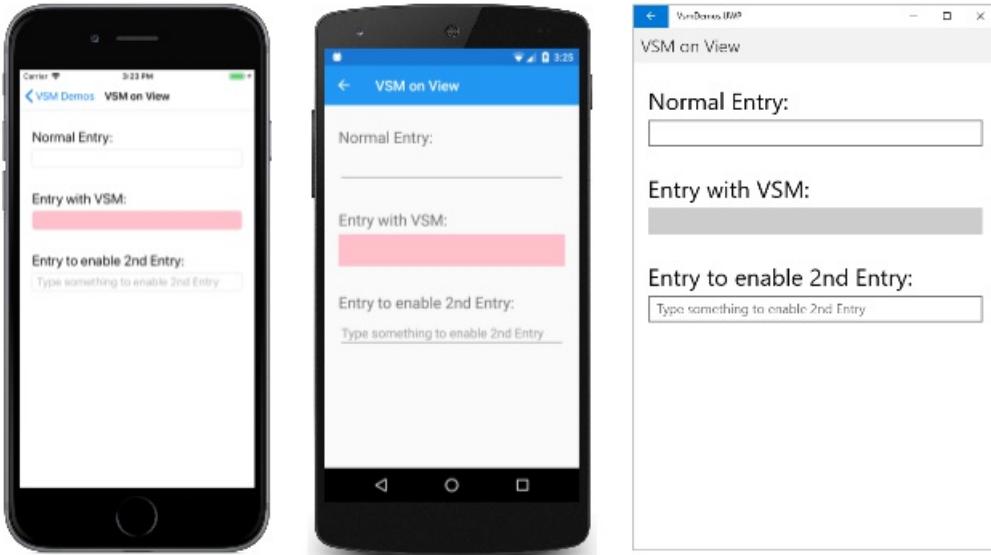
            <Style TargetType="Label">
                <Setter Property="Margin" Value="20, 30, 20, 0" />
                <Setter Property="FontSize" Value="Large" />
            </Style>
        </StackLayout.Resources>

        <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>

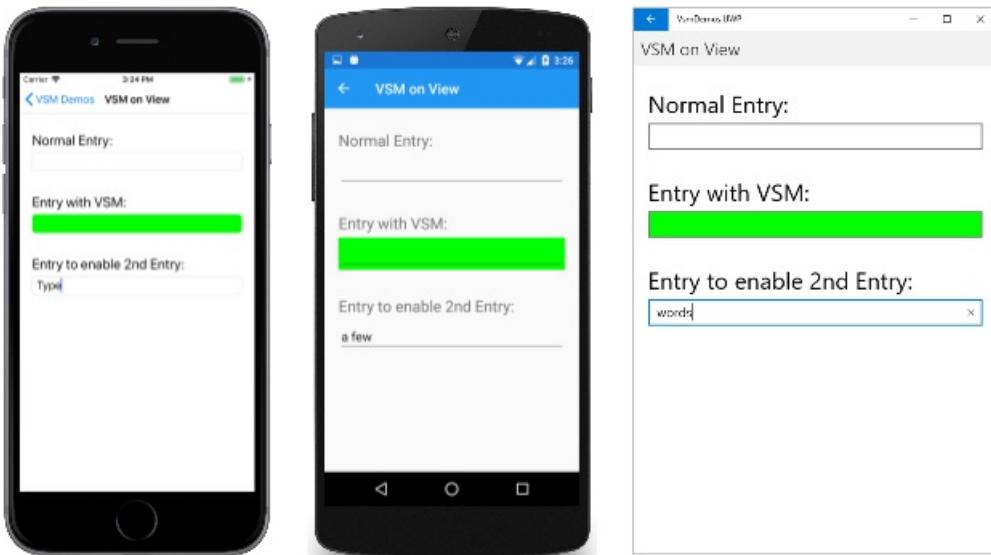
```

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):

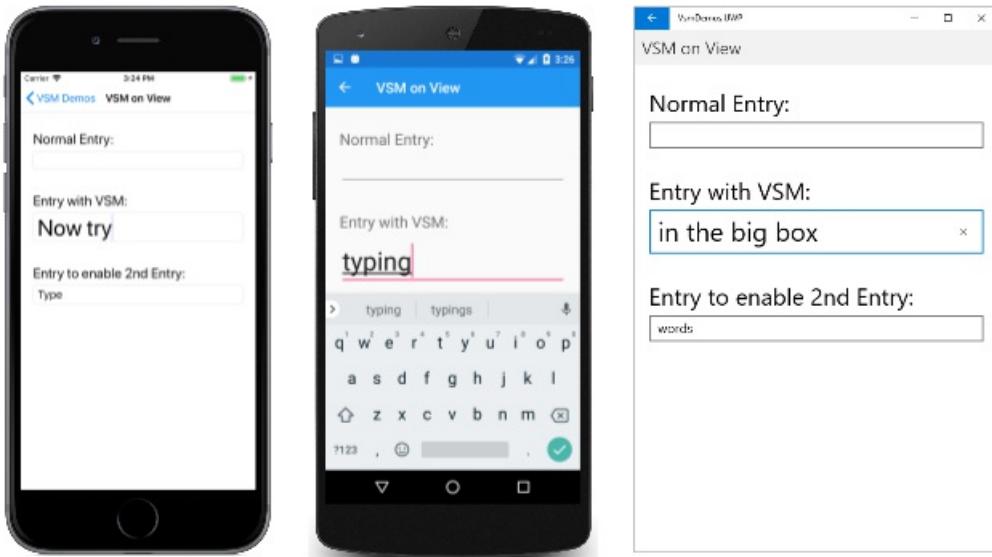


The current visual state is "Disabled" so the background of the second `Entry` is pink on the iOS and Android screens. The UWP implementation of `Entry` does not allow setting the background color when the `Entry` is disabled.

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 `VisualStateManager` 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.

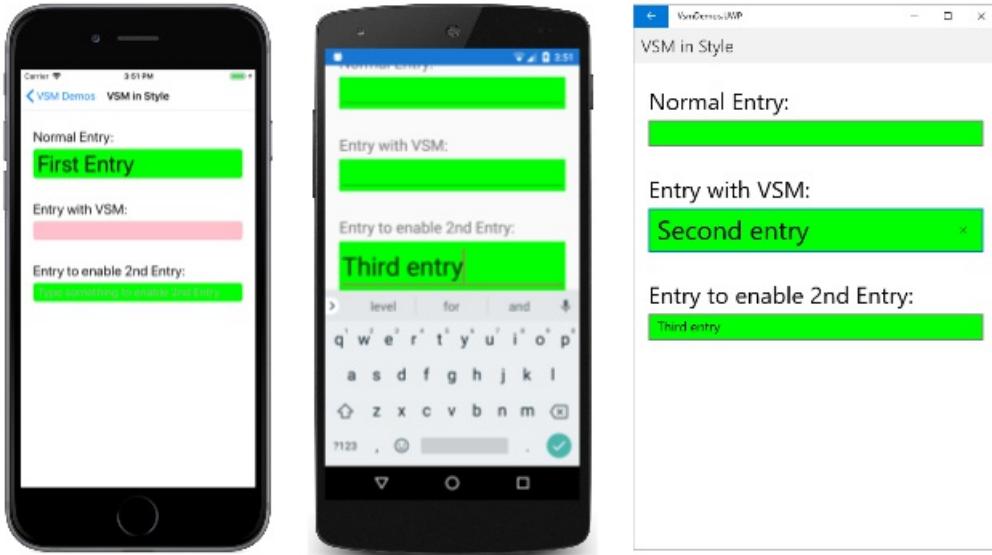
Here's the **VSM in Style** page showing the complete VSM markup:

```

<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:



Visual states in Xamarin.Forms

The following table lists the visual states that are defined in Xamarin.Forms:

CLASS	STATES	MORE INFORMATION
Button	Pressed	Button visual states
CheckBox	IsChecked	CheckBox visual states
CarouselView	DefaultItem , CurrentItem , PreviousItem , NextItem	CarouselView visual states
ImageButton	Pressed	ImageButton visual states
RadioButton	Checked , Unchecked	RadioButton visual states
Switch	On , Off	Switch visual states
VisualElement	Normal , Disabled , Focused , Selected	Common states

Each of these states can be accessed through the visual state group named `CommonStates`.

In addition, the `CollectionView` implements the `Selected` state. For more information, see [Change selected item color](#).

Set state on multiple elements

In the previous examples, visual states were attached to and operated on single elements. However, it's also possible to create visual states that are attached to a single element, but that set properties on other elements within the same scope. This avoids having to repeat visual states on each element the states operate on.

The `Setter` type has a `TargetName` property, of type `string`, that represents the target element that the `Setter` for a visual state will manipulate. When the `TargetName` property is defined, the `Setter` sets the `Property` of the element defined in `TargetName` to `Value`:

```
<Setter TargetName="label"
        Property="Label.TextColor"
        Value="Red" />
```

In this example, a `Label` named `label` will have its `TextColor` property set to `Red`. When setting the `TargetName` property you must specify the full path to the property in `Property`. Therefore, to set the `TextColor` property on a `Label`, `Property` is specified as `Label.TextColor`.

NOTE

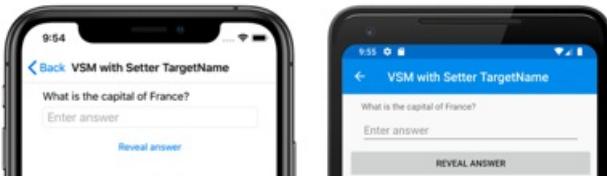
Any property referenced by a `Setter` object must be backed by a bindable property.

The **VSM with Setter TargetName** page in the [VsmDemos](#) sample shows how to set state on multiple elements, from a single visual state group. The XAML file consists of a `StackLayout` containing a `Label` element, an `Entry`, and a `Button`:

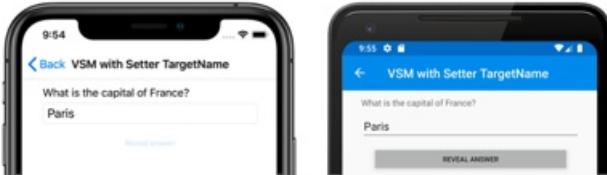
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="VsmDemos.VsmSetterTargetNamePage"
    Title="VSM with Setter TargetName">
    <StackLayout Margin="10">
        <Label Text="What is the capital of France?" />
        <Entry x:Name="entry"
            Placeholder="Enter answer" />
        <Button Text="Reveal answer">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup x:Name="CommonStates">
                    <VisualState x:Name="Normal" />
                    <VisualState x:Name="Pressed">
                        <VisualState.Setters>
                            <Setter Property="Scale"
                                Value="0.8" />
                            <Setter TargetName="entry"
                                Property="Entry.Text"
                                Value="Paris" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </Button>
    </StackLayout>
</ContentPage>
```

VSM markup is attached to the `StackLayout`. There are two mutually-exclusive states, named "Normal" and "Pressed", with each state containing `VisualState` tags.

The "Normal" state is active when the `Button` isn't pressed, and a response to the question can be entered:



The "Pressed" state becomes active when the `Button` is pressed:



The "Pressed" `visualState` specifies that when the `Button` is pressed, its `Scale` property will be changed from the default value of 1 to 0.8. In addition, the `Entry` named `entry` will have its `Text` property set to Paris. Therefore, the result is that when the `Button` is pressed it's rescaled to be slightly smaller, and the `Entry` displays Paris. Then, when the `Button` is released it's rescaled to its default value of 1 ,and the `Entry` displays any previously entered text.

IMPORTANT

Property paths are currently unsupported in `Setter` elements that specify the `TargetName` property.

Define your own visual states

Every class that derives from `VisualElement` supports the common states "Normal", "Focused", and "Disabled". In addition, the `CollectionView` class supports the "Selected" state. 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 a `StackLayout` containing two `Label` elements, an `Entry`, and a `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 x:Name="stackLayout"
        Padding="10, 10">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup Name="ValidityStates">
                <VisualState Name="Valid">
                    <VisualState.Setters>
                        <Setter TargetName="helpLabel"
                            Property="Label.TextColor"
                            Value="Transparent" />
                        <Setter TargetName="entry"
                            Property="Entry.BackgroundColor"
                            Value="Lime" />
                    </VisualState.Setters>
                </VisualState>
                <VisualState Name="Invalid">
                    <VisualState.Setters>
                        <Setter TargetName="entry"
                            Property="Entry.BackgroundColor"
                            Value="Pink" />
                        <Setter TargetName="submitButton"
                            Property="Button.IsEnabled"
                            Value="False" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        <Label Text="Enter a U.S. phone number:"
            FontSize="Large" />
        <Entry x:Name="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" />
        <Button x:Name="submitButton"
            Text="Submit"
            FontSize="Large"
            Margin="0, 20"
            VerticalOptions="Center"
            HorizontalOptions="Center" />
    </StackLayout>
</ContentPage>

```

VSM markup is attached to the `StackLayout` (named `stackLayout`). There are two mutually-exclusive states, named "Valid" and "Invalid", with each state containing `VisualState` tags.

If the `Entry` does not contain a valid phone number, then the current state is "Invalid", and so the `Entry` has a pink background, the second `Label` is visible, and the `Button` is disabled:



When a valid phone number is entered, then the current state becomes "Valid". The `Entry` gets a lime background, the second `Label` 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 `stackLayout`:

```
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(stackLayout, 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 only needs to take account of the object on the page that defines the visual states, and to call `VisualStateManager.GoToState` for this object. This is because both visual states target multiple objects on the page.

You might wonder: If the code-behind file must reference the object on the page that defines the visual states, why can't the code-behind file simply access this and other 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.

Visual state triggers

Visual states support state triggers, which are a specialized group of triggers that define the conditions under which a `VisualState` should be applied.

State triggers are added to the `StateTriggers` collection of a `VisualState`. This collection can contain a single state trigger, or multiple state triggers. A `VisualState` will be applied when any state triggers in the collection are active.

When using state triggers to control visual states, Xamarin.Forms uses the following precedence rules to determine which trigger (and corresponding `VisualState`) will be active:

1. Any trigger that derives from `StateTriggerBase`.
2. An `AdaptiveTrigger` activated due to the `MinWindowWidth` condition being met.
3. An `AdaptiveTrigger` activated due to the `MinWindowHeight` condition being met.

If multiple triggers are simultaneously active (for example, two custom triggers) then the first trigger declared in the markup takes precedence.

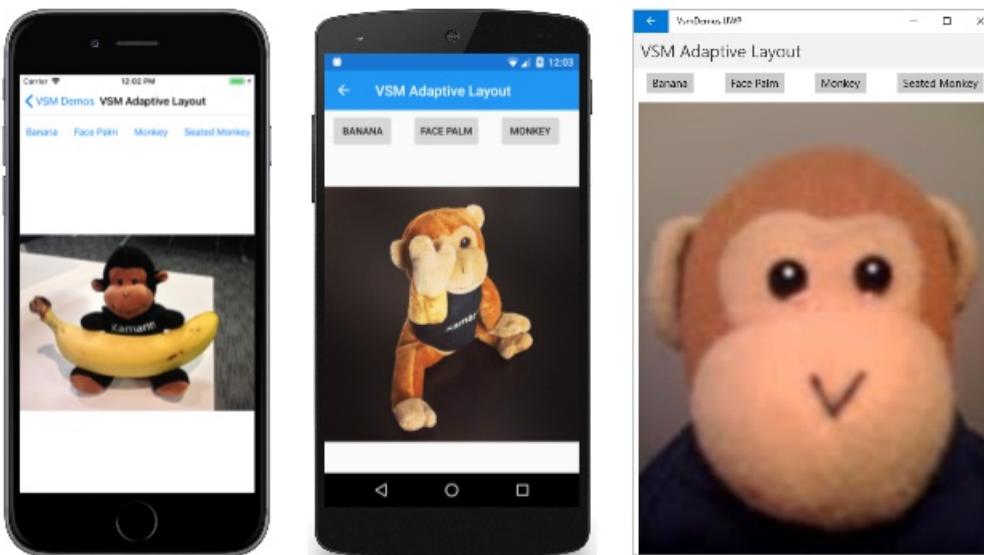
For more information about state triggers, see [State triggers](#).

Use the Visual State Manager for adaptive layout

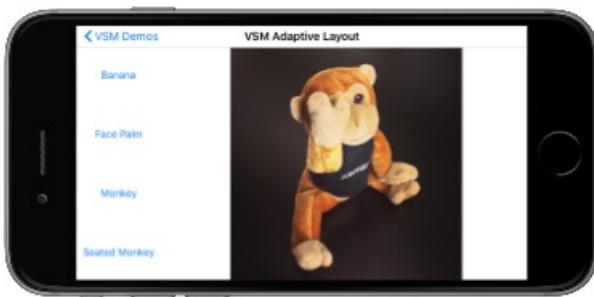
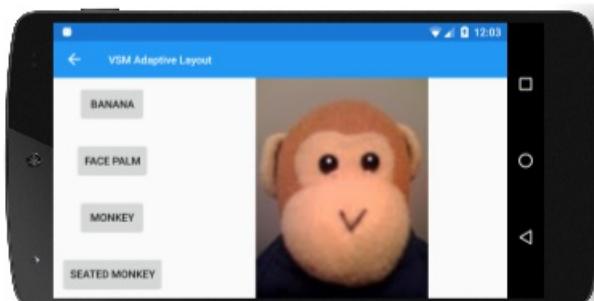
A Xamarin.Forms application running on a phone can usually be viewed in a portrait or landscape aspect ratio, and a Xamarin.Forms program running on the desktop can be resized to assume many different sizes and aspect ratios. A well-designed application might display its content differently for these 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="menuScrollView">
```

```

<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 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" />
                                <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 portrait 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 video](#)

Related links

- [VsmDemos](#)
- [State triggers](#)

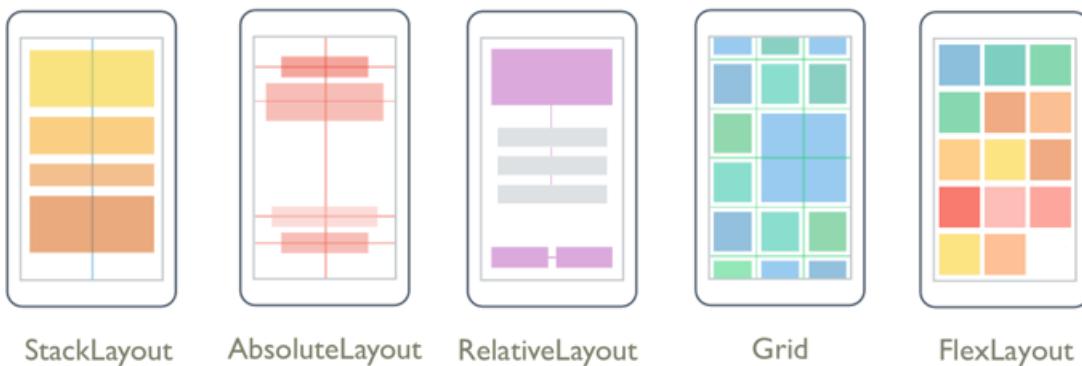
Choose a Xamarin.Forms Layout

8/4/2022 • 8 minutes to read • [Edit Online](#)

 [Download the sample](#)

Xamarin.Forms layout classes allow you to arrange and group UI controls in your application. Choosing a layout class requires knowledge of how the layout positions its child elements, and how the layout sizes its child elements. In addition, it may be necessary to nest layouts to create your desired layout.

The following image shows typical layouts that can be achieved with the main Xamarin.Forms layout classes:



StackLayout

A `StackLayout` organizes elements in a one-dimensional stack, either horizontally or vertically. The `Orientation` property specifies the direction of the elements, and the default orientation is `Vertical`. `StackLayout` is typically used to arrange a subsection of the UI on a page.

The following XAML shows how to create a vertical `StackLayout` containing three `Label` objects:

```
<StackLayout Margin="20,35,20,25">
    <Label Text="The StackLayout has its Margin property set, to control the rendering position of the StackLayout." />
    <Label Text="The Padding property can be set to specify the distance between the StackLayout and its children." />
    <Label Text="The Spacing property can be set to specify the distance between views in the StackLayout." />
</StackLayout>
```

In a `StackLayout`, if an element's size is not explicitly set, it expands to fill the available width, or height if the `Orientation` property is set to `Horizontal`.

A `StackLayout` is often used as a parent layout, which contains other child layouts. However, a `StackLayout` should not be used to reproduce a `Grid` layout by using a combination of `stackLayout` objects. The following code 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">
    <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>

```

This is wasteful because unnecessary layout calculations are performed. Instead, the desired layout can be better achieved by using a [Grid](#).

TIP

When using a [StackLayout](#), ensure that only one child element 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.

For more information, see [Xamarin.Forms StackLayout](#).

Grid

A [Grid](#) is used for displaying elements in rows and columns, which can have proportional or absolute sizes. A grid's rows and columns are specified with the [RowDefinitions](#) and [ColumnDefinitions](#) properties.

To position elements in specific [Grid](#) cells, use the [Grid.Column](#) and [Grid.Row](#) attached properties. To make elements span across multiple rows and columns, use the [Grid.RowSpan](#) and [Grid.ColumnSpan](#) attached properties.

NOTE

A [Grid](#) layout should not be confused with tables, and is not intended to present tabular data. Unlike HTML tables, a [Grid](#) is intended for laying out content. For displaying tabular data, consider using a [ListView](#), [CollectionView](#), or [TableView](#).

The following XAML shows how to create a [Grid](#) with two rows and two columns:

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="50" />
        <RowDefinition Height="50" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Label Text="Column 0, Row 0"
        WidthRequest="200" />
    <Label Grid.Column="1"
        Text="Column 1, Row 0" />
    <Label Grid.Row="1"
        Text="Column 0, Row 1" />
    <Label Grid.Column="1"
        Grid.Row="1"
        Text="Column 1, Row 1" />
</Grid>

```

In this example, sizing works as follows:

- Each row has an explicit height of 50 device-independent units.
- The width of the first column is set to `Auto`, and is therefore as wide as required for its children. In this case, it's 200 device-independent units wide to accommodate the width of the first `Label`.

Space can be distributed within a column or row by using auto sizing, which lets columns and rows size to fit their content. This is achieved by setting the height of a `RowDefinition`, or the width of a `ColumnDefinition`, to `Auto`. Proportional sizing can also be used to distribute available space among the rows and columns of the grid by weighted proportions. This is achieved by setting the height of a `RowDefinition`, or the width of a `ColumnDefinition`, to a value that uses the `*` operator.

Caution

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.

For more information, see [Xamarin.Forms Grid](#).

FlexLayout

A `FlexLayout` is similar to a `StackLayout` in that it displays child elements either horizontally or vertically in a stack. However, a `FlexLayout` can also wrap its children if there are too many to fit in a single row or column, and also enables more granular control of the size, orientation, and alignment of its child elements.

The following XAML shows how to create a `FlexLayout` that displays its views in a single column:

```

<FlexLayout Direction="Column"
    AlignItems="Center"
    JustifyContent="SpaceEvenly">
    <Label Text="FlexLayout in Action" />
    <Button Text="Button" />
    <Label Text="Another Label" />
</FlexLayout>

```

In this example, layout works as follows:

- The `Direction` property is set to `Column`, which causes the children of the `FlexLayout` to be arranged in a

single column of items.

- The `AlignItems` property is set to `Center`, which causes each item to be horizontally centered.
- The `JustifyContent` property is set to `SpaceEvenly`, which allocates all leftover vertical space equally between all the items, and above the first item, and below the last item.

For more information, see [Xamarin.Forms FlexLayout](#).

RelativeLayout

A `RelativeLayout` is used to position and size elements relative to properties of the layout or sibling elements.

By default, an element is positioned in the upper left corner of the layout. A `RelativeLayout` can be used to create UIs that scale proportionally across device sizes.

Within a `RelativeLayout`, positions and sizes are specified as constraints. Constraints 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. In addition, constants can be negative.

NOTE

A `RelativeLayout` supports positioning elements outside of its own bounds.

The following XAML shows how to arrange elements in a `RelativeLayout`:

```
<RelativeLayout>
    <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=.85}"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0}" />
    <BoxView x:Name="pole"
        Color="Gray"
        WidthRequest="15"
        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>
```

In this example, layout works as follows:

- The blue `BoxView` is given an explicit size of 50x50 device-independent units. It's placed in the upper left corner of the layout, which is the default position.
- The red `BoxView` is given an explicit size of 50x50 device-independent units. It's placed in the upper right corner of the layout.
- The gray `BoxView` is given an explicit width of 15 device-independent units, and its height is set to be 75% of the height of its parent.
- The green `BoxView` isn't given an explicit size. Its position is set relative to the `BoxView` named `pole`.

WARNING

Avoid using a `RelativeLayout` whenever possible. It will result in the CPU having to perform significantly more work.

For more information, see [Xamarin.Forms RelativeLayout](#).

AbsoluteLayout

An `AbsoluteLayout` is used to position and size elements using explicit values, or values relative to the size of the layout. The position is specified by the upper-left corner of the child relative to the upper-left corner of the `AbsoluteLayout`.

An `AbsoluteLayout` should be regarded as a special-purpose layout to be used only when you can impose a size on children, or when the element's size doesn't affect the positioning of other children. A standard use of this layout is to create an overlay, which covers the page with other controls, perhaps to protect the user from interacting with the normal controls on the page.

IMPORTANT

The `HorizontalOptions` and `VerticalOptions` properties have no effect on children of an `AbsoluteLayout`.

Within an `AbsoluteLayout`, the `AbsoluteLayout.LayoutBounds` attached property is used to specify the horizontal position, vertical position, width and height of an element. In addition, the `AbsoluteLayout.LayoutFlags` attached property specifies how the layout bounds will be interpreted.

The following XAML shows how to arrange elements in an `AbsoluteLayout`:

```
<AbsoluteLayout Margin="40">
    <BoxView Color="Red"
        AbsoluteLayout.LayoutFlags="PositionProportional"
        AbsoluteLayout.LayoutBounds="0.5, 0, 100, 100"
        Rotation="30" />
    <BoxView Color="Green"
        AbsoluteLayout.LayoutFlags="PositionProportional"
        AbsoluteLayout.LayoutBounds="0.5, 0, 100, 100"
        Rotation="60" />
    <BoxView Color="Blue"
        AbsoluteLayout.LayoutFlags="PositionProportional"
        AbsoluteLayout.LayoutBounds="0.5, 0, 100, 100" />
</AbsoluteLayout>
```

In this example, layout works as follows:

- Each `BoxView` is given an explicit size of 100x100, and is displayed in the same position, horizontally centered.
- The red `BoxView` is rotated 30 degrees, and the green `BoxView` is rotated 60 degrees.

- On each `BoxView`, the `AbsoluteLayout.LayoutFlags` attached property is set to `PositionProportional`, indicating that the position is proportional to the remaining space after width and height are accounted for.

Caution

Avoid using the `AbsoluteLayout.AutoSize` property whenever possible, as it will cause the layout engine to perform additional layout calculations.

For more information, see [Xamarin.Forms AbsoluteLayout](#).

Input transparency

Each visual 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 layout class, its value transfers to child elements. Therefore, setting the `InputTransparent` property to `true` on a layout class will result in all elements within the layout not receiving input.

Layout performance

To obtain the best possible layout performance, follow the guidelines at [Optimize layout performance](#).

In addition, page rendering performance can also be improved by using layout compression, which removes specified layouts from the visual tree. For more information, see [Layout compression](#).

Related links

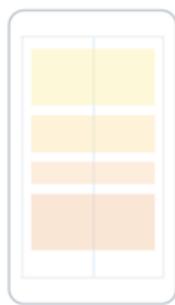
- [Layout \(sample\)](#)
- [Xamarin.Forms Layouts \(video\)](#)
- [Xamarin.Forms StackLayout](#)
- [Xamarin.Forms Grid](#)
- [Xamarin.Forms FlexLayout](#)
- [Xamarin.Forms AbsoluteLayout](#)
- [Xamarin.Forms RelativeLayout](#)
- [Optimize layout performance](#)
- [Layout compression](#)

Xamarin.Forms AbsoluteLayout

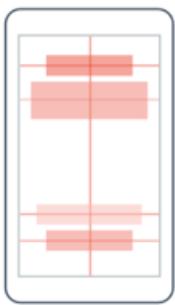
8/4/2022 • 8 minutes to read • [Edit Online](#)



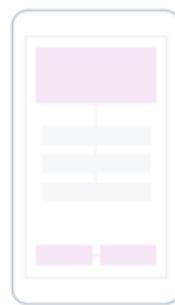
[Download the sample](#)



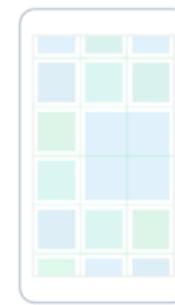
StackLayout



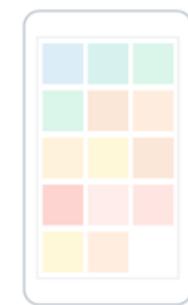
AbsoluteLayout



RelativeLayout



Grid



FlexLayout

An `AbsoluteLayout` is used to position and size children using explicit values. The position is specified by the upper-left corner of the child relative to the upper-left corner of the `AbsoluteLayout`, in device-independent units. `AbsoluteLayout` also implements a proportional positioning and sizing feature. In addition, unlike some other layout classes, `AbsoluteLayout` is able to position children so that they overlap.

An `AbsoluteLayout` should be regarded as a special-purpose layout to be used only when you can impose a size on children, or when the element's size doesn't affect the positioning of other children.

The `AbsoluteLayout` class defines the following properties:

- `LayoutBounds`, of type `Rectangle`, which is an attached property that represents the position and size of a child. The default value of this property is `(0,0,AutoSize,AutoSize)`.
- `LayoutFlags`, of type `AbsoluteLayoutFlags`, which is an attached property that indicates whether properties of the layout bounds used to position and size the child are interpreted proportionally. The default value of this property is `AbsoluteLayoutFlags.None`.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings and styled. For more information about attached properties, see [Xamarin.Forms Attached Properties](#).

The `AbsoluteLayout` class derives from the `Layout<T>` class, which defines a `Children` property of type `IList<T>`. The `Children` property is the `ContentProperty` of the `Layout<T>` class, and therefore does not need to be explicitly set from XAML.

TIP

To obtain the best possible layout performance, follow the guidelines at [Optimize layout performance](#).

Position and size children

The position and size of children in an `AbsoluteLayout` is defined by setting the `AbsoluteLayout.LayoutBounds` attached property of each child, using absolute values or proportional values. Absolute and proportional values can be mixed for children when the position should scale, but the size should stay fixed, or vice versa. For information about absolute values, see [Absolute positioning and sizing](#). For information about proportional values, see [Proportional positioning and sizing](#).

The `AbsoluteLayout.LayoutBounds` attached property can be set using two formats, regardless of whether absolute or proportional values are used:

- `x, y`. With this format, the `x` and `y` values indicate the position of the upper-left corner of the child relative to its parent. The child is unconstrained and sizes itself.
- `x, y, width, height`. With this format, the `x` and `y` values indicate the position of the upper-left corner of the child relative to its parent, while the `width` and `height` values indicate the child's size.

To specify that a child sizes itself horizontally or vertically, or both, set the `width` and/or `height` values to the `AbsoluteLayout.AutoSize` property. However, overuse of this property can harm application performance, as it causes the layout engine to perform additional layout calculations.

IMPORTANT

The `HorizontalOptions` and `VerticalOptions` properties have no effect on children of an `AbsoluteLayout`.

Absolute positioning and sizing

By default, an `AbsoluteLayout` positions and sizes children using absolute values, specified in device-independent units, which explicitly define where children should be placed in the layout. This is achieved by adding children to the `Children` collection of an `AbsoluteLayout` and setting the `AbsoluteLayout.LayoutParams` attached property on each child to absolute position and/or size values.

WARNING

Using absolute values for positioning and sizing children can be problematic, because different devices have different screen sizes and resolutions. Therefore, the coordinates for the center of the screen on one device may be offset on other devices.

The following XAML shows an `AbsoluteLayout` whose children are positioned using absolute values:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="AbsoluteLayoutDemos.Views.StylishHeaderDemoPage"
    Title="Stylish header demo">
    <AbsoluteLayout Margin="20">
        <BoxView Color="Silver"
            AbsoluteLayout.LayoutBounds="0, 10, 200, 5" />
        <BoxView Color="Silver"
            AbsoluteLayout.LayoutBounds="0, 20, 200, 5" />
        <BoxView Color="Silver"
            AbsoluteLayout.LayoutBounds="10, 0, 5, 65" />
        <BoxView Color="Silver"
            AbsoluteLayout.LayoutBounds="20, 0, 5, 65" />
        <Label Text="Stylish Header"
            FontSize="24"
            AbsoluteLayout.LayoutBounds="30, 25" />
    </AbsoluteLayout>
</ContentPage>
```

In this example, the position of each `BoxView` object is defined using the first two absolute values that are specified in the `AbsoluteLayout.LayoutParams` attached property. The size of each `BoxView` is defined using the third and forth values. The position of the `Label` object is defined using the two absolute values that are specified in the `AbsoluteLayout.LayoutParams` attached property. Size values are not specified for the `Label`, and so it's unconstrained and sizes itself. In all cases, the absolute values represent device-independent units.

The following screenshot shows the resulting layout:



The equivalent C# code is shown below:

```
public class StylishHeaderDemoPageCS : ContentPage
{
    public StylishHeaderDemoPageCS()
    {
        AbsoluteLayout absoluteLayout = new AbsoluteLayout
        {
            Margin = new Thickness(20)
        };

        absoluteLayout.Children.Add(new BoxView
        {
            Color = Color.Silver,
        }, new Rectangle(0, 10, 200, 5));
        absoluteLayout.Children.Add(new BoxView
        {
            Color = Color.Silver
        }, new Rectangle(0, 20, 200, 5));
        absoluteLayout.Children.Add(new BoxView
        {
            Color = Color.Silver
        }, new Rectangle(10, 0, 5, 65));
        absoluteLayout.Children.Add(new BoxView
        {
            Color = Color.Silver
        }, new Rectangle(20, 0, 5, 65));

        absoluteLayout.Children.Add(new Label
        {
            Text = "Stylish Header",
            FontSize = 24
        }, new Point(30,25));

        Title = "Stylish header demo";
        Content = absoluteLayout;
    }
}
```

In this example, the position and size of each `BoxView` is defined using a `Rectangle` object. The position of the `Label` is defined using a `Point` object.

In C#, it's also possible to set the position and size of a child of an `AbsoluteLayout` after it has been added to the `Children` collection, using the `AbsoluteLayout.SetLayoutBounds` method. The first argument to this method is the child, and the second is a `Rectangle` object.

NOTE

An `AbsoluteLayout` that uses absolute values can position and size children so that they don't fit within the bounds of the layout.

Proportional positioning and sizing

An `AbsoluteLayout` can position and size children using proportional values. This is achieved by adding children to the `Children` collection of the `AbsoluteLayout`, and by setting the `AbsoluteLayout.LayoutBounds` attached property on each child to proportional position and/or size values in the range 0-1. Position and size values are made proportional by setting the `AbsoluteLayout.LayoutFlags` attached property on each child.

The `AbsoluteLayout.LayoutFlags` attached property, of type `AbsoluteLayoutFlags`, allows you to set a flag that indicates that the layout bounds position and size values for a child are proportional to the size of the `AbsoluteLayout`. When laying out a child, `AbsoluteLayout` scales the position and size values appropriately, to any device size.

The `AbsoluteLayoutFlags` enumeration defines the following members:

- `None`, indicates that values will be interpreted as absolute. This is the default value of the `AbsoluteLayout.LayoutFlags` attached property.
- `XProportional`, indicates that the `x` value will be interpreted as proportional, while treating all other values as absolute.
- `YProportional`, indicates that the `y` value will be interpreted as proportional, while treating all other values as absolute.
- `WidthProportional`, indicates that the `width` value will be interpreted as proportional, while treating all other values as absolute.
- `HeightProportional`, indicates that the `height` value will be interpreted as proportional, while treating all other values as absolute.
- `PositionProportional`, indicates that the `x` and `y` values will be interpreted as proportional, while the size values are interpreted as absolute.
- `SizeProportional`, indicates that the `width` and `height` values will be interpreted as proportional, while the position values are interpreted as absolute.
- `All`, indicates that all values will be interpreted as proportional.

TIP

The `AbsoluteLayoutFlags` enumeration is a `Flags` enumeration, which means that enumeration members can be combined. This is accomplished in XAML with a comma-separated list, and in C# with the bitwise OR operator.

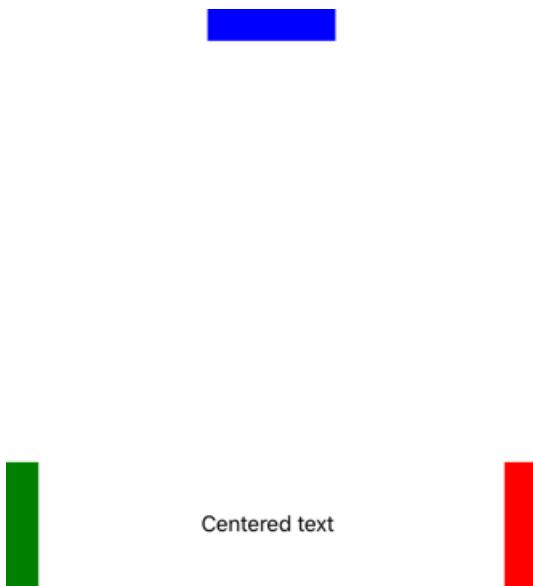
For example, if you use the `SizeProportional` flag and set the width of a child to 0.25 and the height to 0.1, the child will be one-quarter of the width of the `AbsoluteLayout` and one-tenth the height. The `PositionProportional` flag is similar. A position of (0,0) puts the child in the upper-left corner, while a position of (1,1) puts the child in the lower-right corner, and a position of (0.5,0.5) centers the child within the `AbsoluteLayout`.

The following XAML shows an `AbsoluteLayout` whose children are positioned using proportional values:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="AbsoluteLayoutDemos.Views.ProportionalDemoPage"
    Title="Proportional demo">
    <AbsoluteLayout>
        <BoxView Color="Blue"
            AbsoluteLayout.LayoutBounds="0.5,0,100,25"
            AbsoluteLayout.LayoutFlags="PositionProportional" />
        <BoxView Color="Green"
            AbsoluteLayout.LayoutBounds="0,0.5,25,100"
            AbsoluteLayout.LayoutFlags="PositionProportional" />
        <BoxView Color="Red"
            AbsoluteLayout.LayoutBounds="1,0.5,25,100"
            AbsoluteLayout.LayoutFlags="PositionProportional" />
        <BoxView Color="Black"
            AbsoluteLayout.LayoutBounds="0.5,1,100,25"
            AbsoluteLayout.LayoutFlags="PositionProportional" />
        <Label Text="Centered text"
            AbsoluteLayout.LayoutBounds="0.5,0.5,110,25"
            AbsoluteLayout.LayoutFlags="PositionProportional" />
    </AbsoluteLayout>
</ContentPage>
```

In this example, each child is positioned using proportional values but sized using absolute values. This is accomplished by setting the `AbsoluteLayout.LayoutFlags` attached property of each child to `PositionProportional`. The first two values that are specified in the `AbsoluteLayout.LayoutBounds` attached property, for each child, define the position using proportional values. The size of each child is defined with the third and forth absolute values, using device-independent units.

The following screenshot shows the resulting layout:



The equivalent C# code is shown below:

```
public class ProportionalDemoPageCS : ContentPage
{
    public ProportionalDemoPageCS()
    {
        BoxView blue = new BoxView { Color = Color.Blue };
        AbsoluteLayout.SetLayoutBounds(blue, new Rectangle(0.5, 0, 100, 25));
        AbsoluteLayout.SetLayoutFlags(blue, AbsoluteLayoutFlags.PositionProportional);

        BoxView green = new BoxView { Color = Color.Green };
        AbsoluteLayout.SetLayoutBounds(green, new Rectangle(0, 0.5, 25, 100));
        AbsoluteLayout.SetLayoutFlags(green, AbsoluteLayoutFlags.PositionProportional);

        BoxView red = new BoxView { Color = Color.Red };
        AbsoluteLayout.SetLayoutBounds(red, new Rectangle(1, 0.5, 25, 100));
        AbsoluteLayout.SetLayoutFlags(red, AbsoluteLayoutFlags.PositionProportional);

        BoxView black = new BoxView { Color = Color.Black };
        AbsoluteLayout.SetLayoutBounds(black, new Rectangle(0.5, 1, 100, 25));
        AbsoluteLayout.SetLayoutFlags(black, AbsoluteLayoutFlags.PositionProportional);

        Label label = new Label { Text = "Centered text" };
        AbsoluteLayout.SetLayoutBounds(label, new Rectangle(0.5, 0.5, 110, 25));
        AbsoluteLayout.SetLayoutFlags(label, AbsoluteLayoutFlags.PositionProportional);

        Title = "Proportional demo";
        Content = new AbsoluteLayout
        {
            Children = { blue, green, red, black, label }
        };
    }
}
```

In this example, the position and size of each child is set with the `AbsoluteLayout.SetLayoutBounds` method. The first argument to the method is the child, and the second is a `Rectangle` object. The position of each child is set with proportional values, while the size of each child is set with absolute values, using device-independent units.

NOTE

An `AbsoluteLayout` that uses proportional values can position and size children so that they don't fit within the bounds of the layout by using values outside the 0-1 range.

Related links

- [AbsoluteLayout demos \(sample\)](#)
- [Xamarin.Forms Attached Properties](#)
- [Choose a Xamarin.Forms Layout](#)
- [Improve Xamarin.Forms App Performance](#)

The Xamarin.Forms FlexLayout

8/4/2022 • 23 minutes to read • [Edit Online](#)



[Download the sample](#)

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">

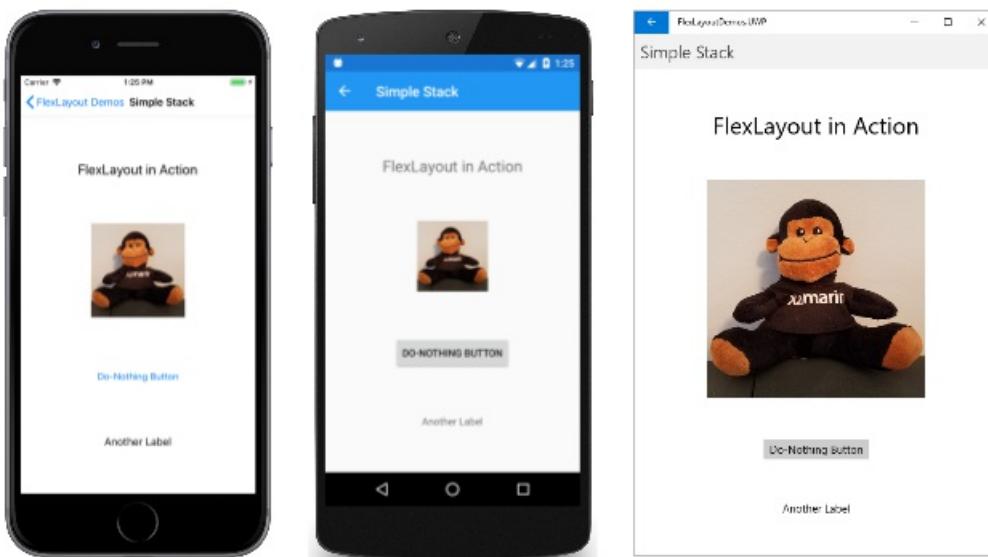
        <Label Text="FlexLayout in Action"
            FontSize="Large" />

        <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()
    {
        using (WebClient webClient = new WebClient())
        {
            try
            {
                // Download the list of stock photos
                Uri uri = new Uri("https://raw.githubusercontent.com/xamarin/docs-
archive/master/Images/stock/small/stock.json");
                byte[] data = await webClient.DownloadDataTaskAsync(uri);

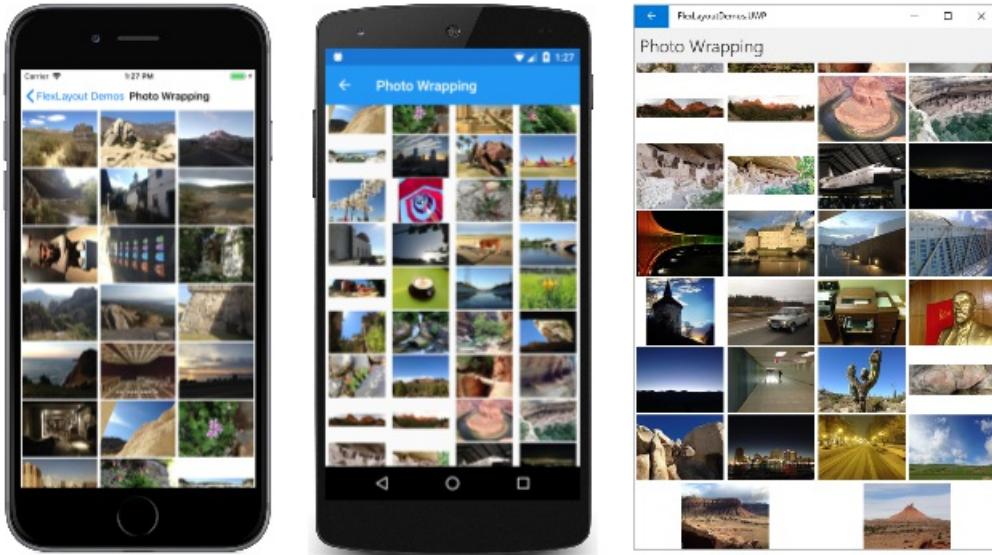
                // Convert to a Stream object
                using (Stream stream = new MemoryStream(data))
                {
                    // Deserialize the JSON into an ImageList object
                    var jsonSerializer = new DataContractJsonSerializer(typeof(ImageList));
                    ImageList 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))
                        };
                        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" />

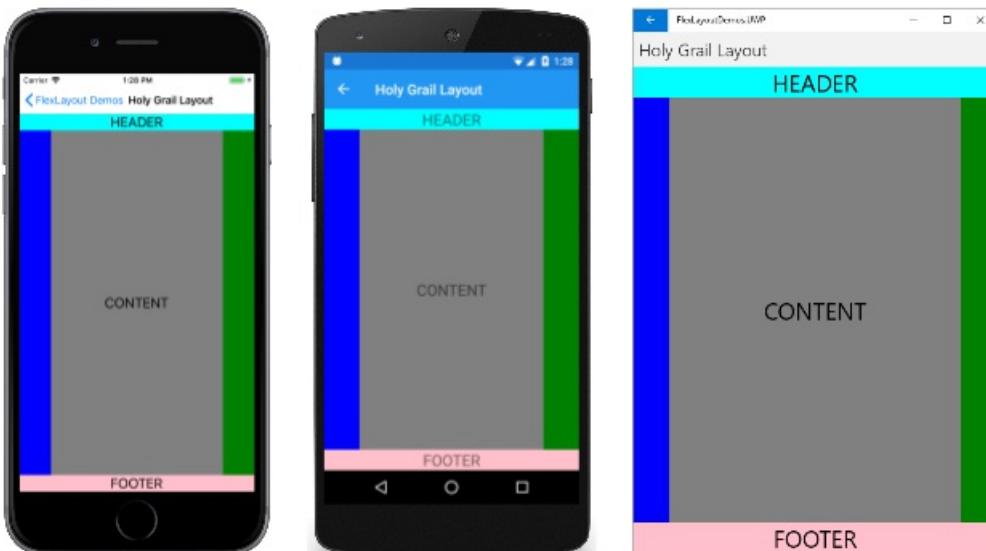
            <!-- Aside items -->
            <BoxView FlexLayout.Basis="50"
                Color="Green" />

        </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:

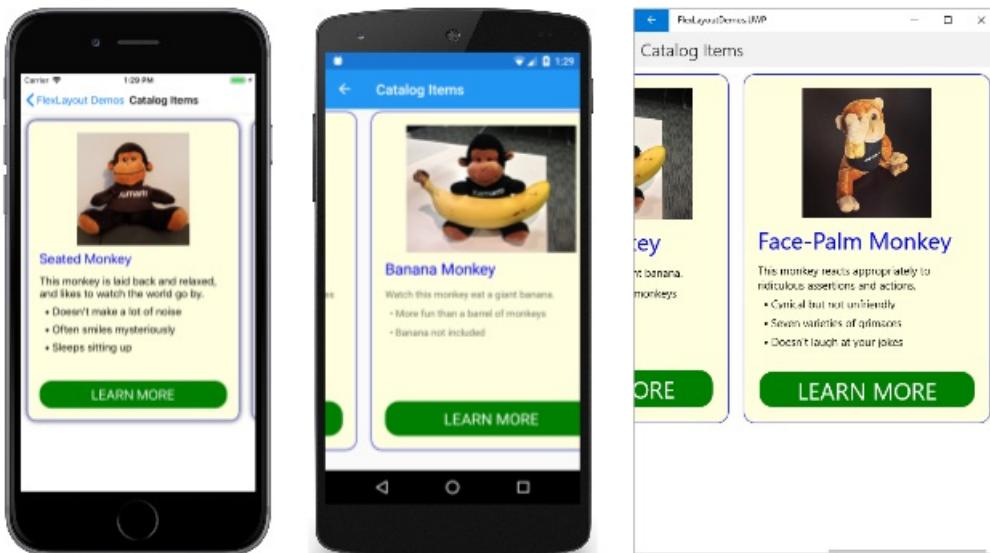
- The `Order` attached bindable property is set on the first `BoxView`. This property is an integer with a default value of 0. You can use this property to change the layout order. Generally developers prefer the content of the page to appear in markup prior to the navigation items and aside items. Setting 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" />
            </FlexLayout>
        </Frame>
    </FlexLayout>
</ScrollView>
```

```

        <Label Text=" &#x2022; Banana not included" />
        <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.

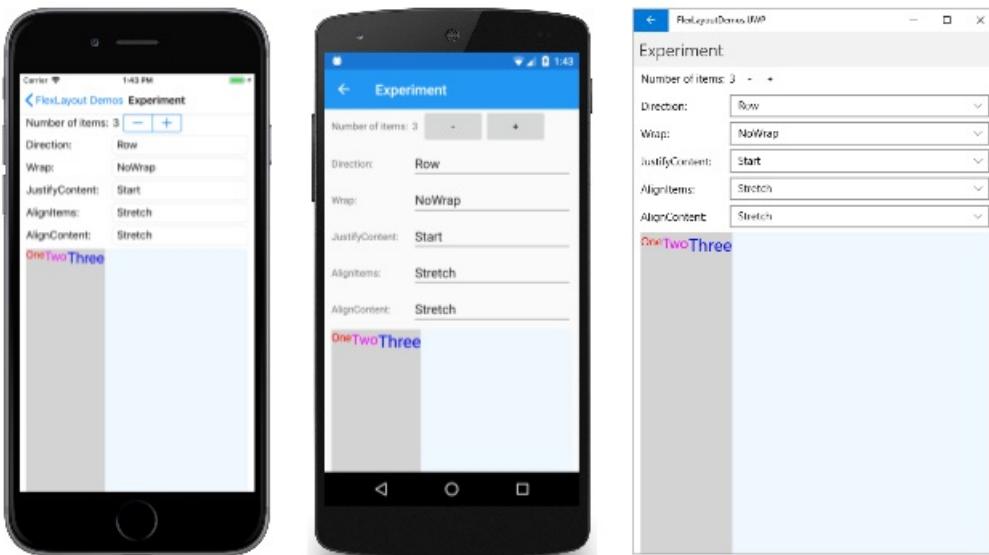
The bindable properties in detail

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.

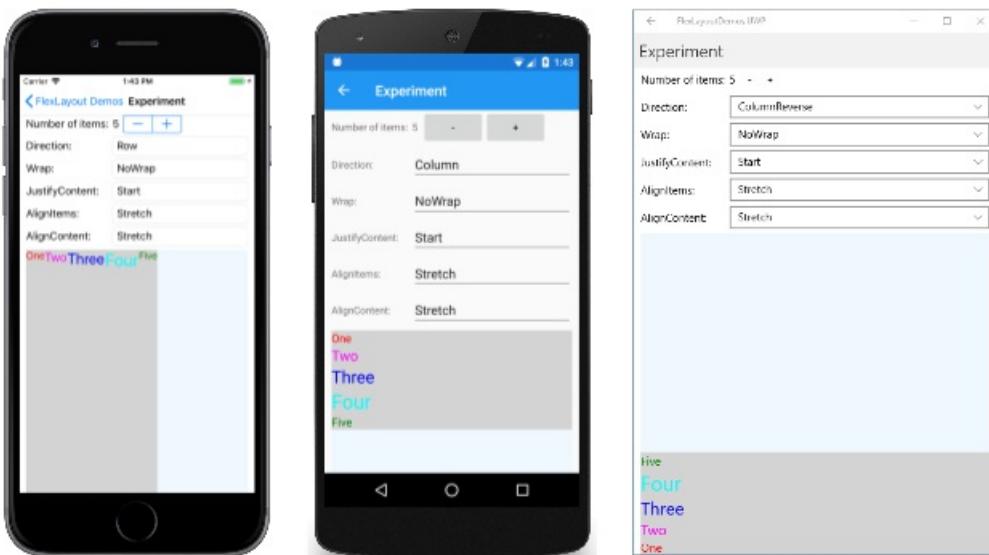
The `Direction` property

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:



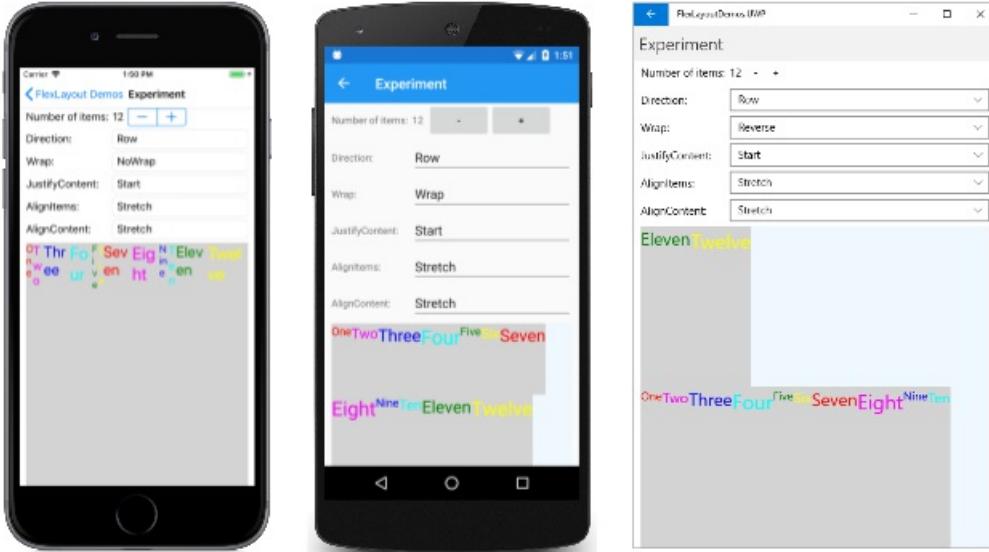
Notice that for the `Reverse` options, the items start at the right or bottom.

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:



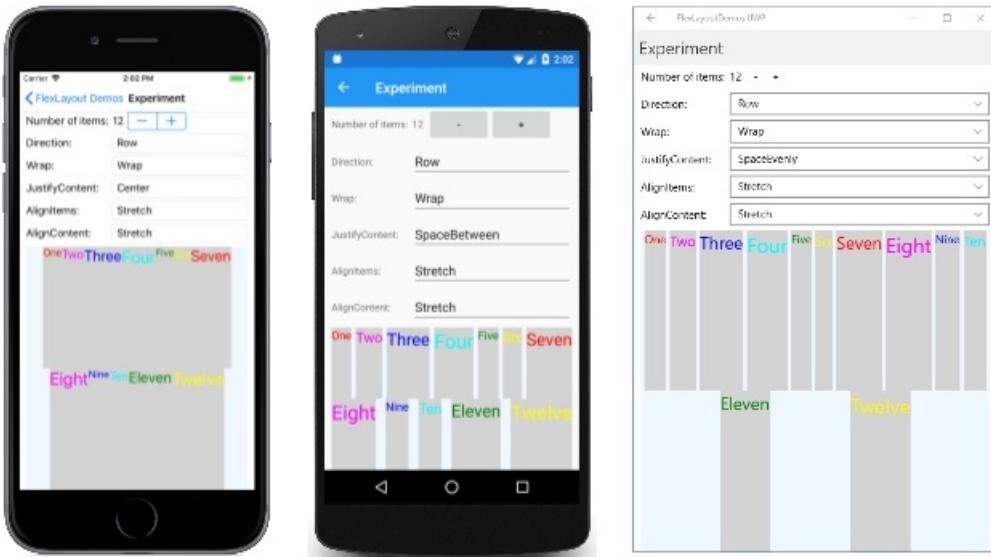
When the `Wrap` property is set to `NoWrap` and the main axis is constrained (as in this program), and the main axis is not wide or tall enough to fit all the children, the `FlexLayout` attempts to make the items smaller, as the iOS screenshot demonstrates. You can control the shrinkage of the items 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:



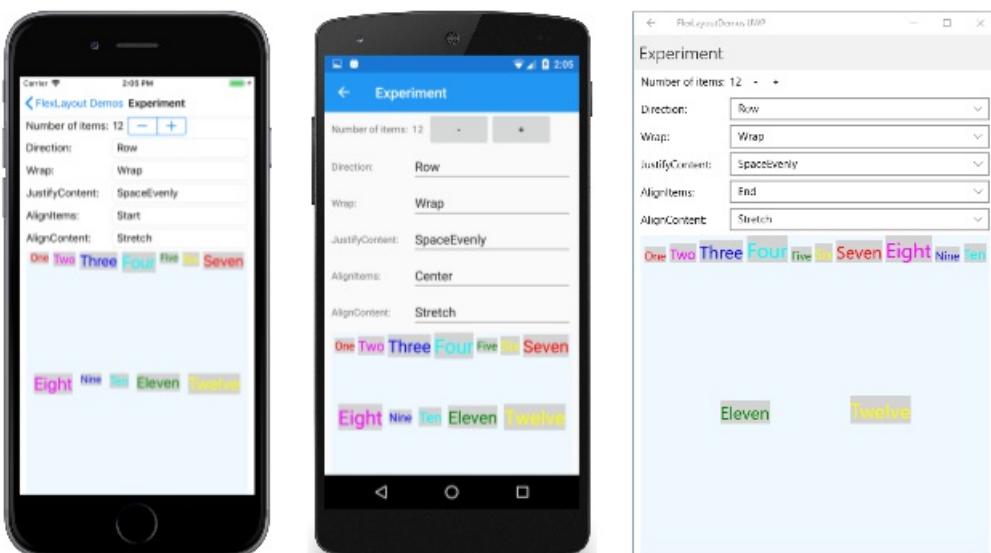
In all three screenshots, the `Wrap` property is set to `Wrap`. The `Start` default is shown in the previous Android screenshot. The iOS screenshot here shows the `Center` option: all the items are moved to the center. The three other options beginning with the word `Space` allocate the extra space not 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.

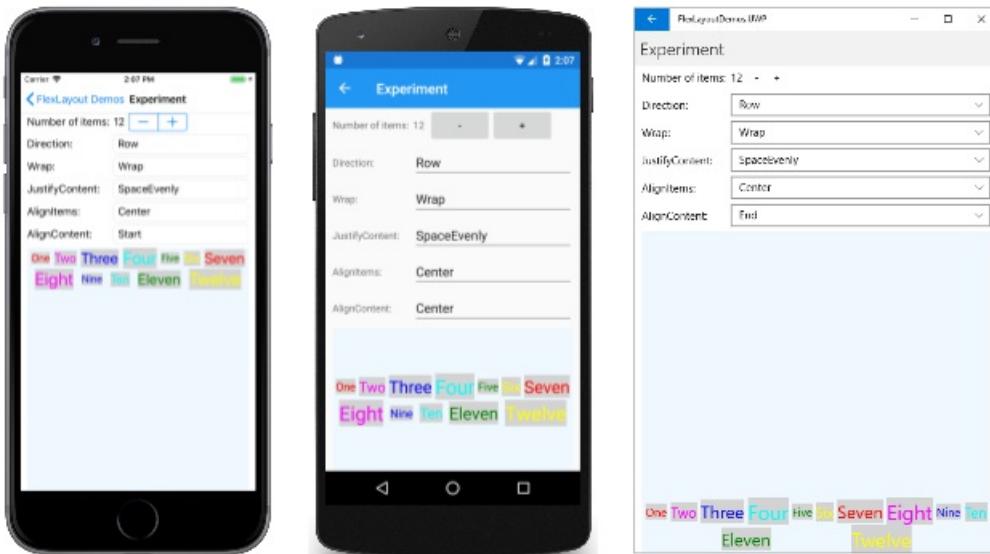
For any individual item, the `AlignItems` setting can be overridden with the `AlignSelf` attached bindable property.

The AlignContent property

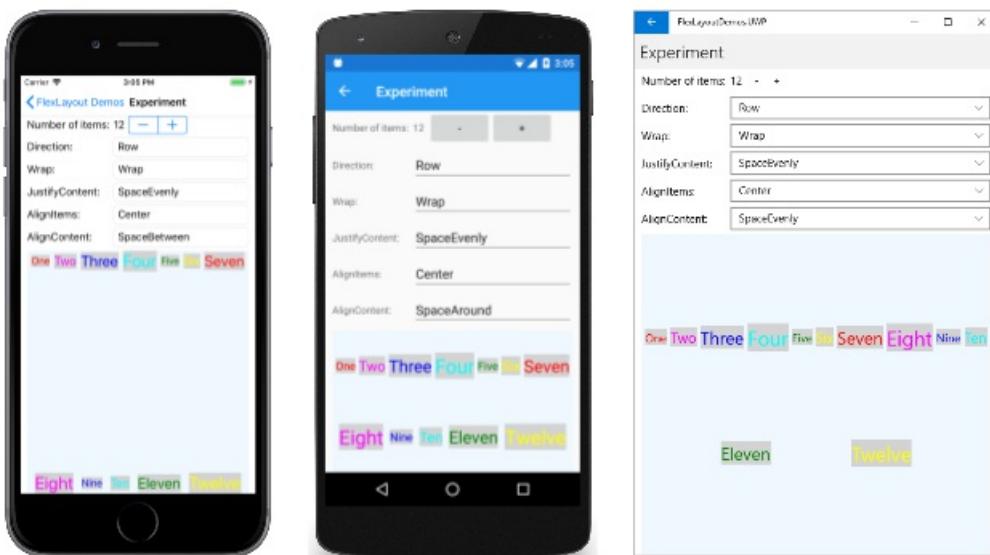
The `AlignContent` property is of type `FlexAlignContent`, an enumeration with seven members:

- `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:

```
FlexLayout.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 ... FlexLayout.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.

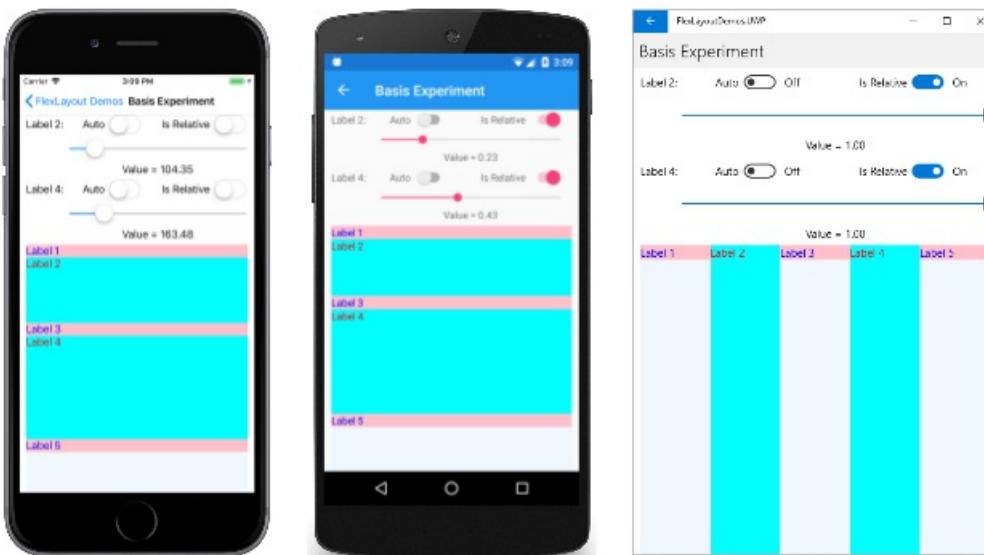
In XAML, you can use a number for a size in device-independent units:

```
<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.

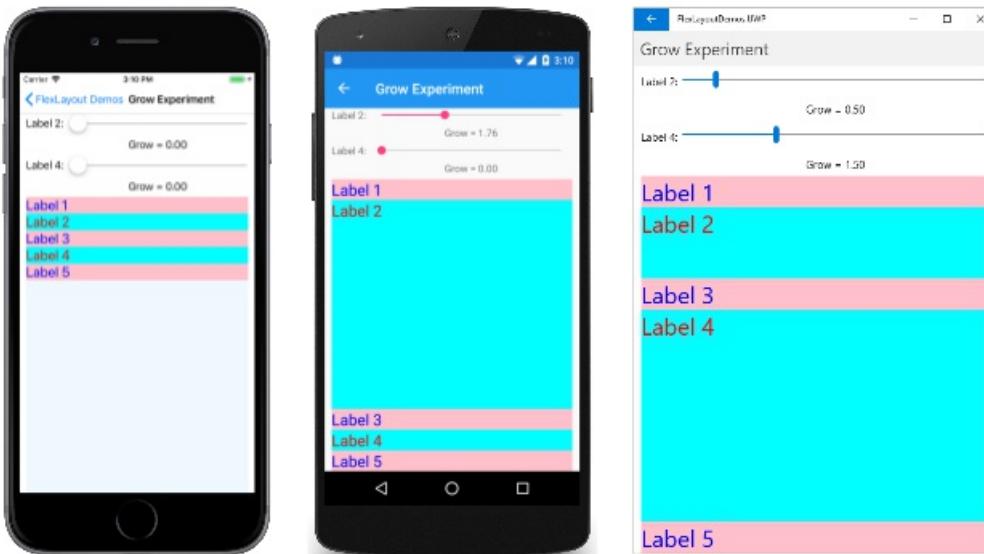
The Grow Property

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:

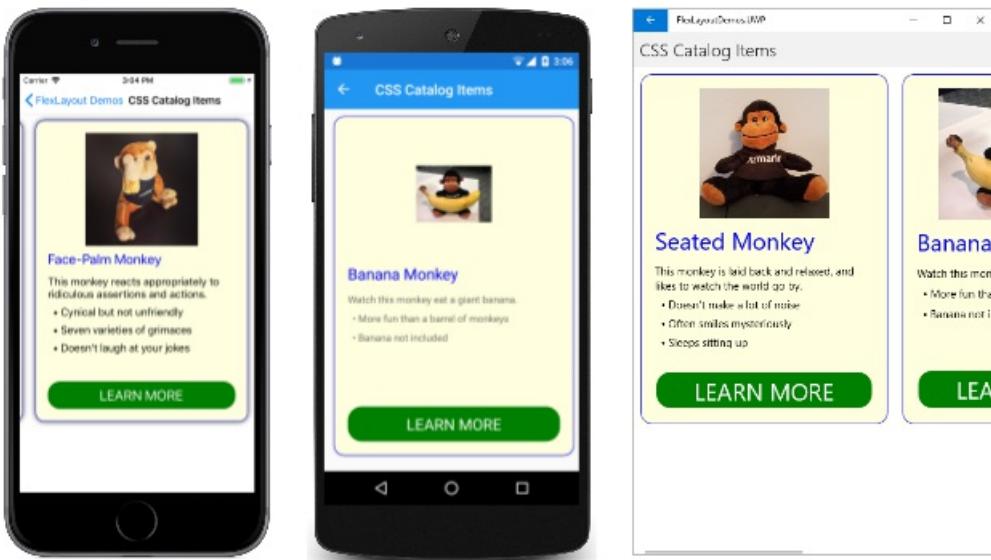


In the Android screenshot, the `Shrink` value for the second `Label` is set to 0, and that `Label` is displayed in its full width. Also, the fourth `Label` is given a `Shrink` value greater than one, and it has shrunk. The UWP screenshot shows both `Label` elements being given a `Shrink` value of 0 to 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=" 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}" />
                    <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=" More fun than a barrel of monkeys" />
                    <Label Text=" 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=" Cynical but not unfriendly" />
                    <Label Text=" Seven varieties of grimaces" />
                    <Label Text=" 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

[Xamarin.Forms 3.0 Flex Layout video](#)

Related links

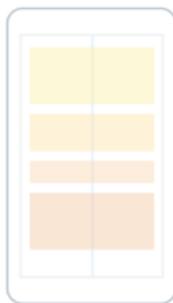
- [FlexLayoutDemos](#)

Xamarin.Forms Grid

8/4/2022 • 15 minutes to read • [Edit Online](#)



[Download the sample](#)



StackLayout



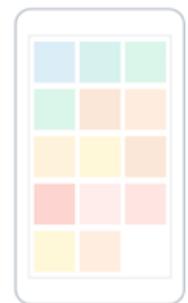
AbsoluteLayout



RelativeLayout



Grid



FlexLayout

The `Grid` is a layout that organizes its children into rows and columns, which can have proportional or absolute sizes. By default, a `Grid` contains one row and one column. In addition, a `Grid` can be used as a parent layout that contains other child layouts.

The `Grid` layout should not be confused with tables, and is not intended to present tabular data. Unlike HTML tables, a `Grid` is intended for laying out content. For displaying tabular data, consider using a [ListView](#), [CollectionView](#), or [TableView](#).

The `Grid` class defines the following properties:

- `Column`, of type `int`, which is an attached property that indicates the column alignment of a view within a parent `Grid`. The default value of this property is 0. A validation callback ensures that when the property is set, its value is greater than or equal to 0.
- `ColumnDefinitions`, of type `ColumnDefinitionCollection`, is a list of `ColumnDefinition` objects that define the width of the grid columns.
- `ColumnSpacing`, of type `double`, indicates the distance between grid columns. The default value of this property is 6 device-independent units.
- `ColumnSpan`, of type `int`, which is an attached property that indicates the total number of columns that a view spans within a parent `Grid`. The default value of this property is 1. A validation callback ensures that when the property is set, its value is greater than or equal to 1.
- `Row`, of type `int`, which is an attached property that indicates the row alignment of a view within a parent `Grid`. The default value of this property is 0. A validation callback ensures that when the property is set, its value is greater than or equal to 0.
- `RowDefinitions`, of type `RowDefinitionCollection`, is a list of `RowDefinition` objects that define the height of the grid rows.
- `RowSpacing`, of type `double`, indicates the distance between grid rows. The default value of this property is 6 device-independent units.
- `RowSpan`, of type `int`, which is an attached property that indicates the total number of rows that a view spans within a parent `Grid`. The default value of this property is 1. A validation callback ensures that when the property is set, its value is greater than or equal to 1.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings and styled.

The `Grid` class derives from the `Layout<T>` class, which defines a `Children` property of type `IList<T>`. The `Children` property is the `ContentProperty` of the `Layout<T>` class, and therefore does not need to be explicitly set from XAML.

TIP

To obtain the best possible layout performance, follow the guidelines at [Optimize layout performance](#).

Rows and columns

By default, a `Grid` contains one row and one column:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="GridTutorial.MainPage">
    <Grid Margin="20,35,20,20">
        <Label Text="By default, a Grid contains one row and one column." />
    </Grid>
</ContentPage>
```

In this example, the `Grid` contains a single child `Label` that's automatically positioned in a single location:



The layout behavior of a `Grid` can be defined with the `RowDefinitions` and `ColumnDefinitions` properties, which are collections of `RowDefinition` and `ColumnDefinition` objects, respectively. These collections define the row and column characteristics of a `Grid`, and should contain one `RowDefinition` object for each row in the `Grid`, and one `ColumnDefinition` object for each column in the `Grid`.

The `RowDefinition` class defines a `Height` property, of type `GridLength`, and the `ColumnDefinition` class defines a `Width` property, of type `GridLength`. The `GridLength` struct specifies a row height or a column width in terms of the `GridUnitType` enumeration, which has three members:

- `Absolute` – the row height or column width is a value in device-independent units (a number in XAML).
- `Auto` – the row height or column width is autosized based on the cell contents (`Auto` in XAML).
- `Star` – leftover row height or column width is allocated proportionally (a number followed by `*` in XAML).

A `Grid` row with a `Height` property of `Auto` constrains the height of views in that row in the same way as a vertical `StackLayout`. Similarly, a column with a `Width` property of `Auto` works much like a horizontal `StackLayout`.

Caution

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.

The following XAML shows how to create a `Grid` with three rows and two columns:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="GridDemos.Views.BasicGridPage"
    Title="Basic Grid demo">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="*" />
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        ...
    </Grid>
</ContentPage>

```

In this example, the `Grid` has an overall height that is the height of the page. The `Grid` knows that the height of the third row is 100 device-independent units. It subtracts that height from its own height, and allocates the remaining height proportionally between the first and second rows based on the number before the star. In this example, the height of the first row is twice that of the second row.

The two `ColumnDefinition` objects both set the `Width` to `*`, which is the same as `1*`, meaning that the width of the screen is divided equally beneath the two columns.

IMPORTANT

The default value of the `RowDefinition.Height` property is `*`. Similarly, the default value of the `ColumnDefinition.Width` property is `*`. Therefore, it's not necessary to set these properties in cases where these defaults are acceptable.

Child views can be positioned in specific `Grid` cells with the `Grid.Column` and `Grid.Row` attached properties. In addition, to make child views span across multiple rows and columns, use the `Grid.RowSpan` and `Grid.ColumnSpan` attached properties.

The following XAML shows the same `Grid` definition, and also positions child views in specific `Grid` cells:

```

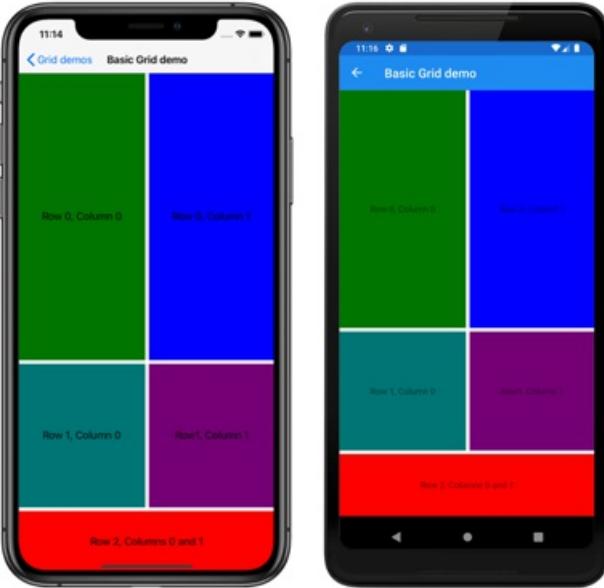
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="GridDemos.Views.BasicGridPage"
    Title="Basic Grid demo">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition />
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <BoxView Color="Green" />
        <Label Text="Row 0, Column 0"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <BoxView Grid.Column="1"
            Color="Blue" />
        <Label Grid.Column="1"
            Text="Row 0, Column 1"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <BoxView Grid.Row="1"
            Color="Teal" />
        <Label Grid.Row="1"
            Text="Row 1, Column 0"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <BoxView Grid.Row="1"
            Grid.Column="1"
            Color="Purple" />
        <Label Grid.Row="1"
            Grid.Column="1"
            Text="Row1, Column 1"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <BoxView Grid.Row="2"
            Grid.ColumnSpan="2"
            Color="Red" />
        <Label Grid.Row="2"
            Grid.ColumnSpan="2"
            Text="Row 2, Columns 0 and 1"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
    </Grid>
</ContentPage>

```

NOTE

The `Grid.Row` and `Grid.Column` properties are both indexed from 0, and so `Grid.Row="2"` refers to the third row while `Grid.Column="1"` refers to the second column. In addition, both of these properties have a default value of 0, and so don't need to be set on child views that occupy the first row or first column of a `Grid`.

In this example, all three `Grid` rows are occupied by `BoxView` and `Label` views. The third row is 100 device-independent units high, with the first two rows occupying the remaining space (the first row is twice as high as the second row). The two columns are equal in width and divide the `Grid` in half. The `BoxView` in the third row spans both columns.



In addition, child views in a `Grid` can share cells. The order that the children appear in the XAML is the order that the children are placed in the `Grid`. In the previous example, the `Label` objects are only visible because they are rendered on top of the `BoxView` objects. The `Label` objects would not be visible if the `BoxView` objects were rendered on top of them.

The equivalent C# code is:

```
public class BasicGridPageCS : ContentPage
{
    public BasicGridPageCS()
    {
        Grid grid = new Grid
        {
            RowDefinitions =
            {
                new RowDefinition { Height = new GridLength(2, GridUnitType.Star) },
                new RowDefinition(),
                new RowDefinition { Height = new GridLength(100) }
            },
            ColumnDefinitions =
            {
                new ColumnDefinition(),
                new ColumnDefinition()
            }
        };

        // Row 0
        // The BoxView and Label are in row 0 and column 0, and so only needs to be added to the
        // Grid.Children collection to get default row and column settings.
        grid.Children.Add(new BoxView
        {
            Color = Color.Green
        });
        grid.Children.Add(new Label
        {
            Text = "Row 0, Column 0",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        });

        // This BoxView and Label are in row 0 and column 1, which are specified as arguments
        // to the Add method.
        grid.Children.Add(new BoxView
        {
            Color = Color.Blue
        });
    }
}
```

```

        s, 1, 0),
        grid.Children.Add(new Label
        {
            Text = "Row 0, Column 1",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        }, 1, 0);

        // Row 1
        // This BoxView and Label are in row 1 and column 0, which are specified as arguments
        // to the Add method overload.
        grid.Children.Add(new BoxView
        {
            Color = Color.Teal
        }, 0, 1, 1, 2);
        grid.Children.Add(new Label
        {
            Text = "Row 1, Column 0",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        }, 0, 1, 1, 2); // These arguments indicate that that the child element goes in the column starting
        // at 0 but ending before 1.
        // They also indicate that the child element goes in the row starting at 1 but
        // ending before 2.

        grid.Children.Add(new BoxView
        {
            Color = Color.Purple
        }, 1, 2, 1, 2);
        grid.Children.Add(new Label
        {
            Text = "Row1, Column 1",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        }, 1, 2, 1, 2);

        // Row 2
        // Alternatively, the BoxView and Label can be positioned in cells with the Grid.SetRow
        // and Grid.SetColumn methods.
        BoxView boxView = new BoxView { Color = Color.Red };
        Grid.SetRow(boxView, 2);
        Grid.SetColumnSpan(boxView, 2);
        Label label = new Label
        {
            Text = "Row 2, Column 0 and 1",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
        Grid.SetRow(label, 2);
        Grid.SetColumnSpan(label, 2);

        grid.Children.Add(boxView);
        grid.Children.Add(label);

        Title = "Basic Grid demo";
        Content = grid;
    }
}

```

In code, to specify the height of a [RowDefinition](#) object, and the width of a [ColumnDefinition](#) object, you use values of the [GridLength](#) structure, often in combination with the [GridUnitType](#) enumeration.

The example code above also shows several different approaches to adding children to the [Grid](#), and specifying the cells in which they reside. 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 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, which assumes a 2x2 `Grid`, shows equivalent code using both `Add` overloads:

```
// left, top
grid.Children.Add(topLeft, 0, 0);           // first column, first row
grid.Children.Add(topRight, 1, 0);           // second column, first row
grid.Children.Add(bottomLeft, 0, 1);          // first column, second row
grid.Children.Add(bottomRight, 1, 1);         // second column, second row

// left, right, top, bottom
grid.Children.Add(topLeft, 0, 1, 0, 1);       // first column, first row
grid.Children.Add(topRight, 1, 2, 0, 1);       // second column, first row
grid.Children.Add(bottomLeft, 0, 1, 1, 2);     // first column, second row
grid.Children.Add(bottomRight, 1, 2, 1, 2);    // second column, second row
```

NOTE

In addition, child views can be added to a `Grid` with the `AddHorizontal` and `AddVertical` methods, which add children to a single row or single column `Grid`. The `Grid` then expands in rows or columns as these calls are made, as well as automatically positioning children in the correct cells.

Simplify row and column definitions

In XAML, the row and column characteristics of a `Grid` can be specified using a simplified syntax that avoids having to define `RowDefinition` and `ColumnDefinition` objects for each row and column. Instead, the `RowDefinitions` and `ColumnDefinitions` properties can be set to strings containing comma-delimited `GridUnitType` values, from which type converters built into Xamarin.Forms create `RowDefinition` and `ColumnDefinition` objects:

```
<Grid RowDefinitions="1*, Auto, 25, 14, 20"
      ColumnDefinitions="*, 2*, Auto, 300">
  ...
</Grid>
```

In this example, the `Grid` has five rows and four columns. The third, forth, and fifth rows are set to absolute heights, with the second row auto-sizing to its content. The remaining height is then allocated to the first row.

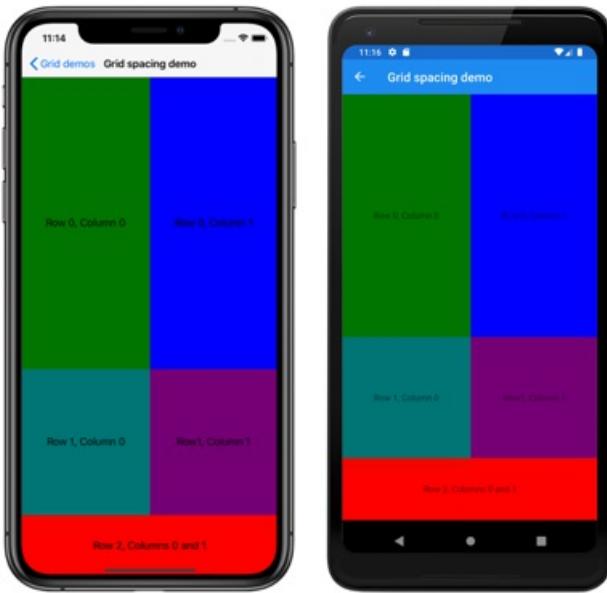
The forth column is set to an absolute width, with the third column auto-sizing to its content. The remaining width is allocated proportionally between the first and second columns based on the number before the star. In this example, the width of the second column is twice that of the first column (because `*` is identical to `1*`).

Space between rows and columns

By default, `Grid` rows are separated by 6 device-independent units of space. Similarly, `Grid` columns are separated by 6 device-independent units of space. These defaults can be changed by setting the `RowSpacing` and `ColumnSpacing` properties, respectively:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="GridDemos.Views.GridSpacingPage"
             Title="Grid spacing demo">
  <Grid RowSpacing="0"
        ColumnSpacing="0">
    ..
  </Grid>
</ContentPage>
```

This example creates a `Grid` that has no spacing between its rows and columns:



TIP

The `RowSpacing` and `ColumnSpacing` properties can be set to negative values to make cell contents overlap.

The equivalent C# code is:

```
public GridSpacingPageCS()
{
    Grid grid = new Grid
    {
        RowSpacing = 0,
        ColumnSpacing = 0,
        // ...
    };
    // ...

    Content = grid;
}
```

Alignment

Child views in a `Grid` can be positioned within their cells by the `HorizontalOptions` and `VerticalOptions` properties. These properties can be set to the following fields from the `LayoutOptions` struct:

- `Start`
- `Center`
- `End`
- `Fill`

IMPORTANT

The `AndExpands` fields in the `LayoutOptions` struct are only applicable to `StackLayout` objects.

The following XAML creates a `Grid` with nine equal-size cells, and places a `Label` in each cell with a different alignment:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="GridDemos.Views.GridAlignmentPage"
    Title="Grid alignment demo">
    <Grid RowSpacing="0"
        ColumnSpacing="0">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

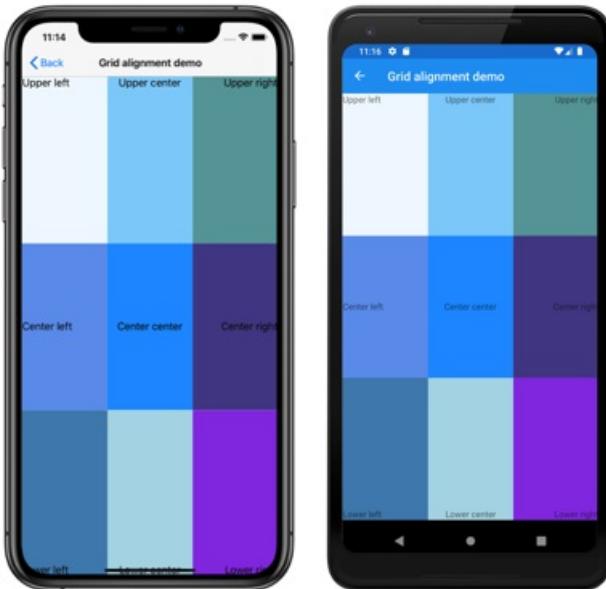
        <BoxView Color="AliceBlue" />
        <Label Text="Upper left"
            HorizontalOptions="Start"
            VerticalOptions="Start" />
        <BoxView Grid.Column="1"
            Color="LightSkyBlue" />
        <Label Grid.Column="1"
            Text="Upper center"
            HorizontalOptions="Center"
            VerticalOptions="Start"/>
        <BoxView Grid.Column="2"
            Color="CadetBlue" />
        <Label Grid.Column="2"
            Text="Upper right"
            HorizontalOptions="End"
            VerticalOptions="Start" />
        <BoxView Grid.Row="1"
            Color="CornflowerBlue" />
        <Label Grid.Row="1"
            Text="Center left"
            HorizontalOptions="Start"
            VerticalOptions="Center" />
        <BoxView Grid.Row="1"
            Grid.Column="1"
            Color="DodgerBlue" />
        <Label Grid.Row="1"
            Grid.Column="1"
            Text="Center center"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <BoxView Grid.Row="1"
            Grid.Column="2"
            Color="DarkSlateBlue" />
        <Label Grid.Row="1"
            Grid.Column="2"
            Text="Center right"
            HorizontalOptions="End"
            VerticalOptions="Center" />
        <BoxView Grid.Row="2"
            Color="SteelBlue" />
        <Label Grid.Row="2"
            Text="Lower left"
            HorizontalOptions="Start"
            VerticalOptions="End" />
        <BoxView Grid.Row="2"
            Grid.Column="1"
            Color="LightBlue" />
        <Label Grid.Row="2"
            Grid.Column="1"
            Text="Lower center"
            HorizontalOptions="Center"
```

```

        VerticalOptions="End" />
<BoxView Grid.Row="2"
        Grid.Column="2"
        Color="BlueViolet" />
<Label Grid.Row="2"
        Grid.Column="2"
        Text="Lower right"
        HorizontalOptions="End"
        VerticalOptions="End" />
</Grid>
</ContentPage>

```

In this example, the `Label` objects in each row are all identically aligned vertically, but use different horizontal alignments. Alternatively, this can be thought of as the `Label` objects in each column being identically aligned horizontally, but using different vertical alignments:



The equivalent C# code is:

```

public class GridAlignmentPageCS : ContentPage
{
    public GridAlignmentPageCS()
    {
        Grid grid = new Grid
        {
            RowSpacing = 0,
            ColumnSpacing = 0,
            RowDefinitions =
            {
                new RowDefinition(),
                new RowDefinition(),
                new RowDefinition()
            },
            ColumnDefinitions =
            {
                new ColumnDefinition(),
                new ColumnDefinition(),
                new ColumnDefinition()
            }
        };

        // Row 0
        grid.Children.Add(new BoxView
        {
            Color = Color.AliceBlue
        });
    }
}

```

```
grid.Children.Add(new Label
{
    Text = "Upper left",
    HorizontalOptions = LayoutOptions.Start,
    VerticalOptions = LayoutOptions.Start
});

grid.Children.Add(new BoxView
{
    Color = Color.LightSkyBlue
}, 1, 0);
grid.Children.Add(new Label
{
    Text = "Upper center",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Start
}, 1, 0);

grid.Children.Add(new BoxView
{
    Color = Color.CadetBlue
}, 2, 0);
grid.Children.Add(new Label
{
    Text = "Upper right",
    HorizontalOptions = LayoutOptions.End,
    VerticalOptions = LayoutOptions.Start
}, 2, 0);

// Row 1
grid.Children.Add(new BoxView
{
    Color = Color.CornflowerBlue
}, 0, 1);
grid.Children.Add(new Label
{
    Text = "Center left",
    HorizontalOptions = LayoutOptions.Start,
    VerticalOptions = LayoutOptions.Center
}, 0, 1);

grid.Children.Add(new BoxView
{
    Color = Color.DodgerBlue
}, 1, 1);
grid.Children.Add(new Label
{
    Text = "Center center",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center
}, 1, 1);

grid.Children.Add(new BoxView
{
    Color = Color.DarkSlateBlue
}, 2, 1);
grid.Children.Add(new Label
{
    Text = "Center right",
    HorizontalOptions = LayoutOptions.End,
    VerticalOptions = LayoutOptions.Center
}, 2, 1);

// Row 2
grid.Children.Add(new BoxView
{
    Color = Color.SteelBlue
}, 0, 2);
grid.Children.Add(new Label
```

```

    {
        Text = "Lower left",
        HorizontalOptions = LayoutOptions.Start,
        VerticalOptions = LayoutOptions.End
    }, 0, 2);

    grid.Children.Add(new BoxView
    {
        Color = Color.LightBlue
    }, 1, 2);
    grid.Children.Add(new Label
    {
        Text = "Lower center",
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.End
    }, 1, 2);

    grid.Children.Add(new BoxView
    {
        Color = Color.BlueViolet
    }, 2, 2);
    grid.Children.Add(new Label
    {
        Text = "Lower right",
        HorizontalOptions = LayoutOptions.End,
        VerticalOptions = LayoutOptions.End
    }, 2, 2);

    Title = "Grid alignment demo";
    Content = grid;
}
}

```

Nested Grid objects

A `Grid` can be used as a parent layout that contains nested child `Grid` objects, or other child layouts. When nesting `Grid` objects, the `Grid.Row`, `Grid.Column`, `Grid.RowSpan`, and `Grid.ColumnSpan` attached properties always refer to the position of views within their parent `Grid`.

The following XAML shows an example of nesting `Grid` objects:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:converters="clr-namespace:GridDemos.Converters"
    x:Class="GridDemos.Views.ColorSlidersGridPage"
    Title="Nested Grids demo">

    <ContentPage.Resources>
        <converters:DoubleToIntConverter x:Key="doubleToInt" />

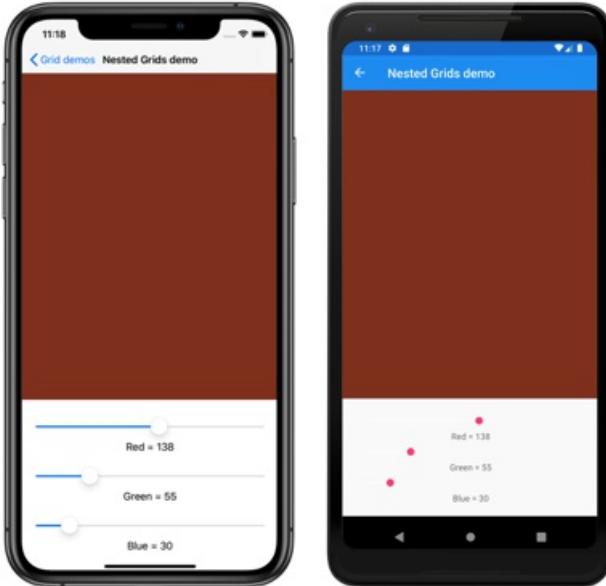
        <Style TargetType="Label">
            <Setter Property="HorizontalTextAlignment"
                Value="Center" />
        </Style>
    </ContentPage.Resources>

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

        <BoxView x:Name="boxView"
            Color="Black" />
        <Grid Grid.Row="1"
            Margin="20">
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Slider x:Name="redSlider"
                ValueChanged="OnSliderValueChanged" />
            <Label Grid.Row="1"
                Text="{Binding Source={x:Reference redSlider},
                    Path=Value,
                    Converter={StaticResource doubleToInt},
                    ConverterParameter=255,
                    StringFormat='Red = {0}'}" />
            <Slider x:Name="greenSlider"
                Grid.Row="2"
                ValueChanged="OnSliderValueChanged" />
            <Label Grid.Row="3"
                Text="{Binding Source={x:Reference greenSlider},
                    Path=Value,
                    Converter={StaticResource doubleToInt},
                    ConverterParameter=255,
                    StringFormat='Green = {0}'}" />
            <Slider x:Name="blueSlider"
                Grid.Row="4"
                ValueChanged="OnSliderValueChanged" />
            <Label Grid.Row="5"
                Text="{Binding Source={x:Reference blueSlider},
                    Path=Value,
                    Converter={StaticResource doubleToInt},
                    ConverterParameter=255,
                    StringFormat='Blue = {0}'}" />
        </Grid>
    </Grid>
</ContentPage>

```

In this example, the root `Grid` layout contains a `BoxView` in its first row, and a child `Grid` in its second row. The child `Grid` contains `Slider` objects that manipulate the color displayed by the `BoxView`, and `Label` objects that display the value of each `Slider`:



IMPORTANT

The deeper you nest `Grid` objects and other layouts, the more the nested layouts will impact performance. For more information, see [Choose the correct layout](#).

The equivalent C# code is:

```
public class ColorSlidersGridPageCS : ContentPage
{
    BoxView boxView;
    Slider redSlider;
    Slider greenSlider;
    Slider blueSlider;

    public ColorSlidersGridPageCS()
    {
        // Create an implicit style for the Labels
        Style labelStyle = new Style(typeof(Label))
        {
            Setters =
            {
                new Setter { Property = Label.HorizontalTextAlignmentProperty, Value = TextAlignment.Center
            }
        }
    };
    Resources.Add(labelStyle);

    // Root page layout
    Grid rootGrid = new Grid
    {
        RowDefinitions =
        {
            new RowDefinition(),
            new RowDefinition()
        }
    };

    boxView = new BoxView { Color = Color.Black };
    rootGrid.Children.Add(boxView);

    // Child page layout
    Grid childGrid = new Grid
    {
        Margin = new Thickness(20),
        Children =
        {
            new BoxView { Color = Color.Red },
            new BoxView { Color = Color.Green },
            new BoxView { Color = Color.Blue }
        }
    };
    rootGrid.Children.Add(childGrid);
}
```

```

RowDefinitions =
{
    new RowDefinition(),
    new RowDefinition(),
    new RowDefinition(),
    new RowDefinition(),
    new RowDefinition(),
    new RowDefinition()
}
};

DoubleToIntConverter doubleToInt = new DoubleToIntConverter();

redSlider = new Slider();
redSlider.ValueChanged += OnSliderValueChanged;
childGrid.Children.Add(redSlider);

Label redLabel = new Label();
redLabel.SetBinding(Label.TextProperty, new Binding("Value", converter: doubleToInt,
converterParameter: "255", stringFormat: "Red = {0}", source: redSlider));
Grid.SetRow(redLabel, 1);
childGrid.Children.Add(redLabel);

greenSlider = new Slider();
greenSlider.ValueChanged += OnSliderValueChanged;
Grid.SetRow(greenSlider, 2);
childGrid.Children.Add(greenSlider);

Label greenLabel = new Label();
greenLabel.SetBinding(Label.TextProperty, new Binding("Value", converter: doubleToInt,
converterParameter: "255", stringFormat: "Green = {0}", source: greenSlider));
Grid.SetRow(greenLabel, 3);
childGrid.Children.Add(greenLabel);

blueSlider = new Slider();
blueSlider.ValueChanged += OnSliderValueChanged;
Grid.SetRow(blueSlider, 4);
childGrid.Children.Add(blueSlider);

Label blueLabel = new Label();
blueLabel.SetBinding(Label.TextProperty, new Binding("Value", converter: doubleToInt,
converterParameter: "255", stringFormat: "Blue = {0}", source: blueSlider));
Grid.SetRow(blueLabel, 5);
childGrid.Children.Add(blueLabel);

// Place the child Grid in the root Grid
rootGrid.Children.Add(childGrid, 0, 1);

Title = "Nested Grids demo";
Content = rootGrid;
}

void OnSliderValueChanged(object sender, ValueChangedEventArgs e)
{
    boxView.Color = new Color(redSlider.Value, greenSlider.Value, blueSlider.Value);
}
}

```

Related links

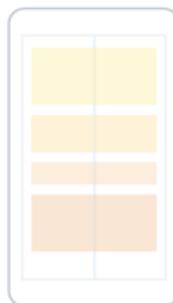
- [Grid demos \(sample\)](#)
- [Layout Options in Xamarin.Forms](#)
- [Choose a Xamarin.Forms Layout](#)
- [Improve Xamarin.Forms App Performance](#)

Xamarin.Forms RelativeLayout

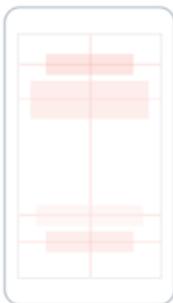
8/4/2022 • 10 minutes to read • [Edit Online](#)



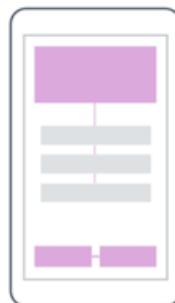
[Download the sample](#)



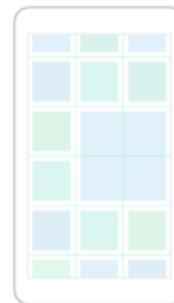
StackLayout



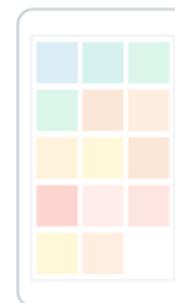
AbsoluteLayout



RelativeLayout



Grid



FlexLayout

A `RelativeLayout` is used to position and size children relative to properties of the layout or sibling elements. This allows UIs to be created that scale proportionally across device sizes. In addition, unlike some other layout classes, `RelativeLayout` is able to position children so that overlap.

The `RelativeLayout` class defines the following properties:

- `XConstraint`, of type `Constraint`, which is an attached property that represents the constraint on the X position of the child.
- `YConstraint`, of type `Constraint`, which is an attached property that represents the constraint on the Y position of the child.
- `WidthConstraint`, of type `Constraint`, which is an attached property that represents the constraint on the width of the child.
- `HeightConstraint`, of type `Constraint`, which is an attached property that represents the constraint on the height of the child.
- `BoundsConstraint`, of type `BoundsConstraint`, which is an attached property that represents the constraint on the position and size of the child. This property can't be easily consumed from XAML.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings and styled. For more information about attached properties, see [Xamarin.Forms Attached Properties](#).

NOTE

The width and height of a child in a `RelativeLayout` can also be specified through the child's `WidthRequest` and `HeightRequest` properties, instead of the `WidthConstraint` and `HeightConstraint` attached properties.

The `RelativeLayout` class derives from the `Layout<T>` class, which defines a `Children` property of type `IList<T>`. The `Children` property is the `ContentProperty` of the `Layout<T>` class, and therefore does not need to be explicitly set from XAML.

TIP

Avoid using a `RelativeLayout` whenever possible. It will result in the CPU having to perform significantly more work.

Constraints

Within a `RelativeLayout`, the position and size of children are specified as constraints using absolute values or relative values. When constraints aren't specified, a child will be positioned in the upper left corner of the layout.

The following table shows how to specify constraints in XAML and C#:

	XAML	C#
Absolute values	Absolute constraints are specified by setting the <code>RelativeLayout</code> attached properties to <code>double</code> values.	Absolute constraints are specified by the <code>Constraint.Constant</code> method, or by using the <code>Children.Add</code> overload that requires a <code>Func<Rectangle></code> argument.
Relative values	Relative constraints are specified by setting the <code>RelativeLayout</code> attached properties to <code>Constraint</code> objects that are returned by the <code>ConstraintExpression</code> markup extension.	Relative constraints are specified by <code>Constraint</code> objects that are returned by methods of the <code>Constraint</code> class.

For more information about specifying constraints using absolute values, see [Absolute positioning and sizing](#).

For more information about specifying constraints using relative values, see [Relative positioning and sizing](#).

In C#, children can be added to `RelativeLayout` by three `Add` overloads. The first overload requires a `Expression<Func<Rectangle>>` to specify the position and size of a child. The second overload requires optional `Expression<Func<double>>` objects for the `x`, `y`, `width`, and `height` arguments. The third overload requires optional `Constraint` objects for the `x`, `y`, `width`, and `height` arguments.

It's possible to change the position and size of a child in a `RelativeLayout` with the `SetXConstraint`, `SetYConstraint`, `SetWidthConstraint`, and `SetHeightConstraint` methods. The first argument to each of these methods is the child, and the second is a `Constraint` object. In addition, the `SetBoundsConstraint` method can also be used to change the position and size of a child. The first argument to this method is the child, and the second is a `BoundsConstraint` object.

Absolute positioning and sizing

A `RelativeLayout` can position and size children using absolute values, specified in device-independent units, which explicitly define where children should be placed in the layout. This is achieved by adding children to the `Children` collection of a `RelativeLayout` and setting the `XConstraint`, `YConstraint`, `WidthConstraint`, and `HeightConstraint` attached properties on each child to absolute position and/or size values.

WARNING

Using absolute values for positioning and sizing children can be problematic, because different devices have different screen sizes and resolutions. Therefore, the coordinates for the center of the screen on one device may be offset on other devices.

The following XAML shows a `RelativeLayout` whose children are positioned using absolute values:

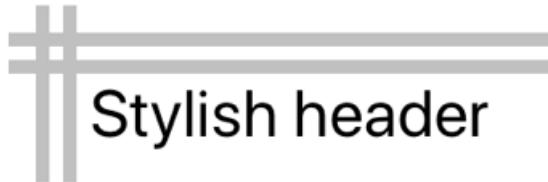
```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="RelativeLayoutDemos.Views.StylishHeaderDemoPage"
    Title="Stylish header demo">
    <RelativeLayout Margin="20">
        <BoxView Color="Silver"
            RelativeLayout.XConstraint="0"
            RelativeLayout.YConstraint="10"
            RelativeLayout.WidthConstraint="200"
            RelativeLayout.HeightConstraint="5" />
        <BoxView Color="Silver"
            RelativeLayout.XConstraint="0"
            RelativeLayout.YConstraint="20"
            RelativeLayout.WidthConstraint="200"
            RelativeLayout.HeightConstraint="5" />
        <BoxView Color="Silver"
            RelativeLayout.XConstraint="10"
            RelativeLayout.YConstraint="0"
            RelativeLayout.WidthConstraint="5"
            RelativeLayout.HeightConstraint="65" />
        <BoxView Color="Silver"
            RelativeLayout.XConstraint="20"
            RelativeLayout.YConstraint="0"
            RelativeLayout.WidthConstraint="5"
            RelativeLayout.HeightConstraint="65" />
        <Label Text="Stylish header"
            FontSize="24"
            RelativeLayout.XConstraint="30"
            RelativeLayout.YConstraint="25" />
    </RelativeLayout>
</ContentPage>

```

In this example, the position of each `BoxView` object is defined using the values specified in the `XConstraint` and `YConstraint` attached properties. The size of each `BoxView` is defined using the values specified in the `WidthConstraint` and `HeightConstraint` attached properties. The position of the `Label` object is also defined using the values specified in the `XConstraint` and `YConstraint` attached properties. However, size values are not specified for the `Label`, and so it's unconstrained and sizes itself. In all cases, the absolute values represent device-independent units.

The following screenshots show the resulting layout:



The equivalent C# code is shown below:

```

public class StylishHeaderDemoPageCS : ContentPage
{
    public StylishHeaderDemoPageCS()
    {
        RelativeLayout relativeLayout = new RelativeLayout
        {
            Margin = new Thickness(20)
        };

        relativeLayout.Children.Add(new BoxView
        {
            Color = Color.Silver
        }, () => new Rectangle(0, 10, 200, 5));

        relativeLayout.Children.Add(new BoxView
        {
            Color = Color.Silver
        }, () => new Rectangle(0, 20, 200, 5));

        relativeLayout.Children.Add(new BoxView
        {
            Color = Color.Silver
        }, () => new Rectangle(10, 0, 5, 65));

        relativeLayout.Children.Add(new BoxView
        {
            Color = Color.Silver
        }, () => new Rectangle(20, 0, 5, 65));

        relativeLayout.Children.Add(new Label
        {
            Text = "Stylish Header",
            FontSize = 24
        }, Constraint.Constant(30), Constraint.Constant(25));

        Title = "Stylish header demo";
        Content = relativeLayout;
    }
}

```

In this example, `BoxView` objects are added to the `RelativeLayout` using an `Add` overload that requires a `Expression<Func<Rectangle>>` to specify the position and size of each child. The position of the `Label` is defined using an `Add` overload that requires optional `Constraint` objects, in this case created by the `Constraint.Constant` method.

NOTE

A `RelativeLayout` that uses absolute values can position and size children so that they don't fit within the bounds of the layout.

Relative positioning and sizing

A `RelativeLayout` can position and size children using values that are relative to properties of the layout, or sibling elements. This is achieved by adding children to the `Children` collection of the `RelativeLayout` and setting the `XConstraint`, `YConstraint`, `WidthConstraint`, and `HeightConstraint` attached properties on each child to relative values using `Constraint` objects.

Constraints can be a constant, relative to a parent, or relative to a sibling. The type of constraint is represented by the `ConstraintType` enumeration, which defines the following members:

- `RelativeToParent`, which indicates a constraint that is relative to a parent.
- `RelativeToView`, which indicates a constraint that is relative to a view (or sibling).
- `Constant`, which indicates a constant constraint.

Constraint markup extension

In XAML, a `Constraint` object can be created by the `ConstraintExpression` markup extension. This markup extension is typically used to relate the position and size of a child within a `RelativeLayout` to its parent, or to a sibling.

The `ConstraintExpression` class defines the following properties:

- `Constant`, of type `double`, which represents the constraint constant value.
- `ElementName`, of type `string`, which represents the name of a source element against which to calculate the constraint.
- `Factor`, of type `double`, which represents the factor by which to scale a constrained dimension, relative to the source element. This property defaults to 1.
- `Property`, of type `string`, which represents the name of the property on the source element to use in the constraint calculation.
- `Type`, of type `ConstraintType`, which represents the type of the constraint.

For more information about Xamarin.Forms markup extensions, see [XAML Markup Extensions](#).

The following XAML shows a `RelativeLayout` whose children are constrained by the `ConstraintExpression` markup extension:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="RelativeLayoutDemos.Views.RelativePositioningAndSizingDemoPage"
    Title="RelativeLayout demo">
    <RelativeLayout>
        <BoxView Color="Red"
            RelativeLayout.XConstraint="{ConstraintExpression Type=Constant, Constant=0}"
            RelativeLayout.YConstraint="{ConstraintExpression Type=Constant, Constant=0}" />
        <BoxView Color="Green"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
            Constant=-40}"
            RelativeLayout.YConstraint="{ConstraintExpression Type=Constant, Constant=0}" />
        <BoxView Color="Blue"
            RelativeLayout.XConstraint="{ConstraintExpression Type=Constant, Constant=0}"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
            Constant=-40}" />
        <BoxView Color="Yellow"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
            Constant=-40}"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
            Constant=-40}" />

        <!-- Centered and 1/3 width and height of parent -->
        <BoxView x:Name="oneThird"
            Color="Silver"
            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="Black"
            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}"
            RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToView,
            ElementName=oneThird, Property=Height, Factor=0.33}" />
    </RelativeLayout>
</ContentPage>

```

In this example, the position of each `BoxView` object is defined by setting the `XConstraint` and `YConstraint` attached properties. The first `BoxView` has its `XConstraint` and `YConstraint` attached properties set to constants, which are absolute values. The remaining `BoxView` objects all have their position set by using at least one relative value. For example, the yellow `BoxView` object sets the `XConstraint` attached property to the width of its parent (the `RelativeLayout`) minus 40. Similarly, this `BoxView` sets the `YConstraint` attached property to the height of its parent minus 40. This ensures that the yellow `BoxView` appears in the lower-right corner of the screen.

NOTE

`BoxView` objects that don't specify a size are automatically sized to 40x40 by Xamarin.Forms.

The silver `BoxView` named `oneThird` is positioned centrally, relative to its parent. It's also sized relative to its parent, being one third of its width and height. This is achieved by setting the `XConstraint` and `WidthConstraint`

attached properties to the width of the parent (the `RelativeLayout`), multiplied by 0.33. Similarly, the `YConstraint` and `HeightConstraint` attached properties are set to the height of the parent, multiplied by 0.33.

The black `BoxView` is positioned and sized relative to the `oneThird` `BoxView`. This is achieved by setting its `XConstraint` and `YConstraint` attached properties to the `x` and `y` values, respectively, of the sibling element. Similarly, its size is set to one third of the width and height of its sibling element. This is achieved by setting its `WidthConstraint` and `HeightConstraint` attached properties to the `Width` and `Height` values of the sibling element, respectively, which are then multiplied by 0.33.

The following screenshot shows the resulting layout:



Constraint objects

The `Constraint` class defines the following public static methods, which return `Constraint` objects:

- `Constant`, which constrains a child to a size specified with a `double`.
- `FromExpression`, which constrains a child using a lambda expression.
- `RelativeToParent`, which constrains a child relative to its parent's size.
- `RelativeToView`, which constrains a child relative to the size of a view.

In addition, the `BoundsConstraint` class defines a single method, `FromExpression`, which returns a `BoundsConstraint` that constrains a child's position and size with a `Expression<Func<Rectangle>>`. This method can be used to set the `BoundsConstraint` attached property.

The following C# code shows a `RelativeLayout` whose children are constrained by `Constraint` objects:

```
public class RelativePositioningAndSizingDemoPageCS : ContentPage
{
```

```

public RelativePositioningAndSizingDemoPageCS()
{
    RelativeLayout relativeLayout = new RelativeLayout();

    // Four BoxView's
    relativeLayout.Children.Add(
        new BoxView { Color = Color.Red },
        Constraint.Constant(0),
        Constraint.Constant(0));

    relativeLayout.Children.Add(
        new BoxView { Color = Color.Green },
        Constraint.RelativeToParent((parent) =>
    {
        return parent.Width - 40;
    }), Constraint.Constant(0));

    relativeLayout.Children.Add(
        new BoxView { Color = Color.Blue },
        Constraint.Constant(0),
        Constraint.RelativeToParent((parent) =>
    {
        return parent.Height - 40;
    })); 

    relativeLayout.Children.Add(
        new BoxView { Color = Color.Yellow },
        Constraint.RelativeToParent((parent) =>
    {
        return parent.Width - 40;
    }),
        Constraint.RelativeToParent((parent) =>
    {
        return parent.Height - 40;
    })); 

    // Centered and 1/3 width and height of parent
    BoxView silverBoxView = new BoxView { Color = Color.Silver };
    relativeLayout.Children.Add(
        silverBoxView,
        Constraint.RelativeToParent((parent) =>
    {
        return parent.Width * 0.33;
    }),
        Constraint.RelativeToParent((parent) =>
    {
        return parent.Height * 0.33;
    }),
        Constraint.RelativeToParent((parent) =>
    {
        return parent.Width * 0.33;
    }),
        Constraint.RelativeToParent((parent) =>
    {
        return parent.Height * 0.33;
    })); 

    // 1/3 width and height of previous
    relativeLayout.Children.Add(
        new BoxView { Color = Color.Black },
        Constraint.RelativeToView(silverBoxView, (parent, sibling) =>
    {
        return sibling.X;
    }),
        Constraint.RelativeToView(silverBoxView, (parent, sibling) =>
    {
        return sibling.Y;
    }),
        Constraint.RelativeToView(silverBoxView, (parent, sibling) =>

```

```
        {
            return sibling.Width * 0.33;
        }),
        Constraint.RelativeToView(silverBoxView, (parent, sibling) =>
    {
        return sibling.Height * 0.33;
    }));
}

Title = "RelativeLayout demo";
Content = relativeLayout;
}
}
```

In this example, children are added to the `RelativeLayout` using the `Add` overload that requires an optional `Constraint` object for the `x`, `y`, `width`, and `height` arguments.

NOTE

A `RelativeLayout` that uses relative values can position and size children so that they don't fit within the bounds of the layout.

Related links

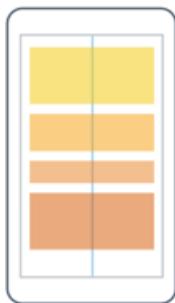
- [RelativeLayout demos \(sample\)](#)
- [Xamarin.Forms Attached Properties](#)
- [XAML Markup Extensions](#)
- [Choose a Xamarin.Forms Layout](#)
- [Improve Xamarin.Forms App Performance](#)

Xamarin.Forms StackLayout

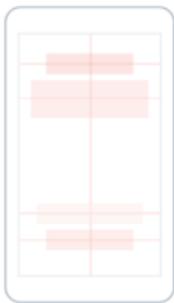
8/4/2022 • 9 minutes to read • [Edit Online](#)



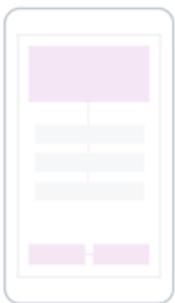
[Download the sample](#)



StackLayout



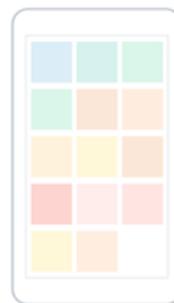
AbsoluteLayout



RelativeLayout



Grid



FlexLayout

A `StackLayout` organizes child views in a one-dimensional stack, either horizontally or vertically. By default, a `StackLayout` is oriented vertically. In addition, a `StackLayout` can be used as a parent layout that contains other child layouts.

The `StackLayout` class defines the following properties:

- `Orientation`, of type `StackOrientation`, represents the direction in which child views are positioned. The default value of this property is `Vertical`.
- `Spacing`, of type `double`, indicates the amount of space between each child view. The default value of this property is six device-independent units.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings and styled.

The `StackLayout` class derives from the `Layout<T>` class, which defines a `Children` property of type `IList<T>`. The `children` property is the `ContentProperty` of the `Layout<T>` class, and therefore does not need to be explicitly set from XAML.

TIP

To obtain the best possible layout performance, follow the guidelines at [Optimize layout performance](#).

Vertical orientation

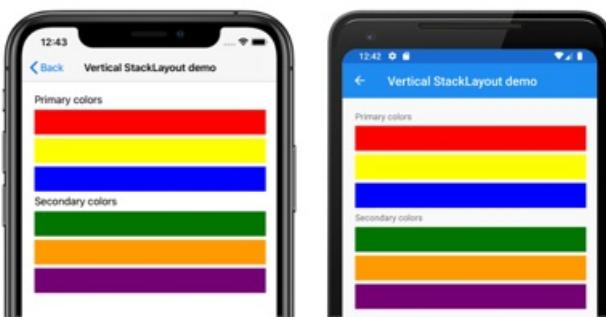
The following XAML shows how to create a vertically oriented `StackLayout` that contains different child views:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.VerticalStackLayoutPage"
    Title="Vertical StackLayout demo">
    <StackLayout Margin="20">
        <Label Text="Primary colors" />
        <BoxView Color="Red" />
        <BoxView Color="Yellow" />
        <BoxView Color="Blue" />
        <Label Text="Secondary colors" />
        <BoxView Color="Green" />
        <BoxView Color="Orange" />
        <BoxView Color="Purple" />
    </StackLayout>
</ContentPage>

```

This example creates a vertical `StackLayout` containing `Label` and `BoxView` objects. By default, there are six device-independent units of space between the child views:



The equivalent C# code is:

```

public class VerticalStackLayoutPageCS : ContentPage
{
    public VerticalStackLayoutPageCS()
    {
        Title = "Vertical StackLayout demo";
        Content = new StackLayout
        {
            Margin = new Thickness(20),
            Children =
            {
                new Label { Text = "Primary colors" },
                new BoxView { Color = Color.Red },
                new BoxView { Color = Color.Yellow },
                new BoxView { Color = Color.Blue },
                new Label { Text = "Secondary colors" },
                new BoxView { Color = Color.Green },
                new BoxView { Color = Color.Orange },
                new BoxView { Color = Color.Purple }
            }
        };
    }
}

```

NOTE

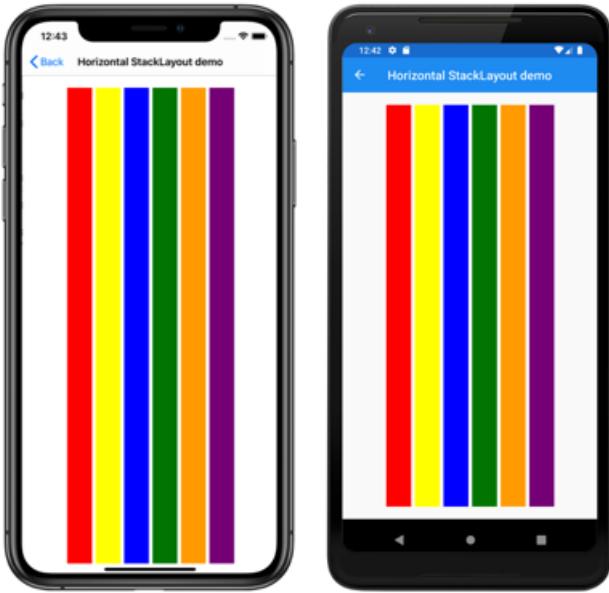
The value of the `Margin` property represents the distance between an element and its adjacent elements. For more information, see [Margin and Padding](#).

Horizontal orientation

The following XAML shows how to create a horizontally oriented `StackLayout` by setting its `Orientation` property to `Horizontal`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.HorizontalStackLayoutPage"
    Title="Horizontal StackLayout demo">
    <StackLayout Margin="20"
        Orientation="Horizontal"
        HorizontalOptions="Center">
        <BoxView Color="Red" />
        <BoxView Color="Yellow" />
        <BoxView Color="Blue" />
        <BoxView Color="Green" />
        <BoxView Color="Orange" />
        <BoxView Color="Purple" />
    </StackLayout>
</ContentPage>
```

This example creates a horizontal `StackLayout` containing `BoxView` objects, with six device-independent units of space between the child views:



The equivalent C# code is:

```

public HorizontalStackLayoutPageCS()
{
    Title = "Horizontal StackLayout demo";
    Content = new StackLayout
    {
        Margin = new Thickness(20),
        Orientation = StackOrientation.Horizontal,
        HorizontalOptions = LayoutOptions.Center,
        Children =
        {
            new BoxView { Color = Color.Red },
            new BoxView { Color = Color.Yellow },
            new BoxView { Color = Color.Blue },
            new BoxView { Color = Color.Green },
            new BoxView { Color = Color.Orange },
            new BoxView { Color = Color.Purple }
        }
    };
}

```

Space between child views

The spacing between child views in a `StackLayout` can be changed by setting the `Spacing` property to a `double` value:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.StackLayoutSpacingPage"
    Title="StackLayout Spacing demo">
    <StackLayout Margin="20"
        Spacing="0">
        <Label Text="Primary colors" />
        <BoxView Color="Red" />
        <BoxView Color="Yellow" />
        <BoxView Color="Blue" />
        <Label Text="Secondary colors" />
        <BoxView Color="Green" />
        <BoxView Color="Orange" />
        <BoxView Color="Purple" />
    </StackLayout>
</ContentPage>

```

This example creates a vertical `StackLayout` containing `Label` and `BoxView` objects that have no spacing between them:



TIP

The `Spacing` property can be set to negative values to make child views overlap.

The equivalent C# code is:

```

public class StackLayoutSpacingPageCS : ContentPage
{
    public StackLayoutSpacingPageCS()
    {
        Title = "StackLayout Spacing demo";
        Content = new StackLayout
        {
            Margin = new Thickness(20),
            Spacing = 0,
            Children =
            {
                new Label { Text = "Primary colors" },
                new BoxView { Color = Color.Red },
                new BoxView { Color = Color.Yellow },
                new BoxView { Color = Color.Blue },
                new Label { Text = "Secondary colors" },
                new BoxView { Color = Color.Green },
                new BoxView { Color = Color.Orange },
                new BoxView { Color = Color.Purple }
            }
        };
    }
}

```

Position and size of child views

The size and position of child views within a `StackLayout` depends upon the values of the child views' `HeightRequest` and `WidthRequest` properties, and the values of their `HorizontalOptions` and `VerticalOptions` properties. In a vertical `StackLayout`, child views expand to fill the available width when their size isn't explicitly set. Similarly, in a horizontal `StackLayout`, child views expand to fill the available height when their size isn't explicitly set.

The `HorizontalOptions` and `VerticalOptions` properties of a `StackLayout`, and its child views, can be set to fields from the `LayoutOptions` struct, which encapsulates two layout preferences:

- *Alignment* determines the position and size of a child view within its parent layout.
- *Expansion* indicates if the child view should use extra space, if it's available.

TIP

Don't set the `HorizontalOptions` and `VerticalOptions` properties of a `StackLayout` unless you need to. 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 back to the default values.

Alignment

The following XAML example sets alignment preferences on each child view in the `StackLayout`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.AlignmentPage"
    Title="Alignment demo">
    <StackLayout Margin="20">
        <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>
</ContentPage>

```

In this example, alignment preferences are set on the `Label` objects to control their position within the `StackLayout`. The `Start`, `Center`, `End`, and `Fill` fields are used to define the alignment of the `Label` objects within the parent `StackLayout`:



A `StackLayout` only respects the alignment preferences on child views that are in the opposite direction to the `StackLayout` orientation. Therefore, the `Label` child views within the vertically oriented `StackLayout` set their `HorizontalOptions` properties to one of the alignment fields:

- `Start`, which positions the `Label` on the left-hand side of the `StackLayout`.
- `Center`, which centers the `Label` in the `StackLayout`.
- `End`, which positions the `Label` on the right-hand side of the `StackLayout`.
- `Fill`, which ensures that the `Label` fills the width of the `StackLayout`.

The equivalent C# code is:

```

public class AlignmentPageCS : ContentPage
{
    public AlignmentPageCS()
    {
        Title = "Alignment demo";
        Content = new StackLayout
        {
            Margin = new Thickness(20),
            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 }
            }
        };
    }
}

```

Expansion

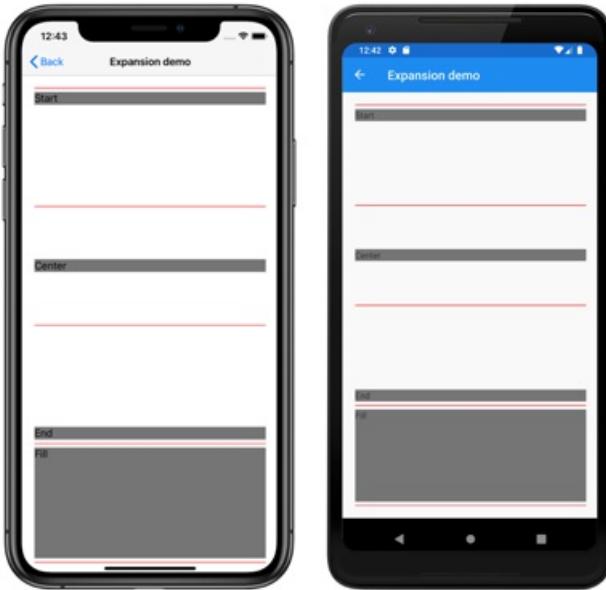
The following XAML example sets expansion preferences on each `Label` in the `StackLayout`:

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.ExpansionPage"
    Title="Expansion demo">
    <StackLayout Margin="20">
        <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>
</ContentPage>

```

In this example, expansion preferences are set on the `Label` objects to control their size within the `StackLayout`. The `StartAndExpand`, `CenterAndExpand`, `EndAndExpand`, and `FillAndExpand` fields are used to define the alignment preference, and whether the `Label` will occupy more space if available within the parent `StackLayout`:



A `StackLayout` can only expand child views in the direction of its orientation. Therefore, the vertically oriented `StackLayout` can expand `Label` child views that set their `VerticalOptions` properties to one of the expansion fields. This means that, for vertical alignment, 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.

TIP

When using a `StackLayout`, ensure that only one child view 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.

The equivalent C# code is:

```
public ExpansionPageCS()
{
    Title = "Expansion demo";
    Content = new StackLayout
    {
        Margin = new Thickness(20),
        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 }
        }
    };
}
```

IMPORTANT

When all the space in a `StackLayout` is used, expansion preferences have no effect.

For more information about alignment and expansion, see [Layout Options in Xamarin.Forms](#).

Nested StackLayout objects

A `StackLayout` can be used as a parent layout that contains nested child `StackLayout` objects, or other child layouts.

The following XAML shows an example of nesting `StackLayout` objects:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.CombinedStackLayoutPage"
    Title="Combined StackLayouts demo">
    <StackLayout Margin="20">
        ...
        <Frame BorderColor="Black"
            Padding="5">
            <StackLayout Orientation="Horizontal"
                Spacing="15">
                <BoxView Color="Red" />
                <Label Text="Red"
                    FontSize="Large"
                    VerticalOptions="Center" />
            </StackLayout>
        </Frame>
        <Frame BorderColor="Black"
            Padding="5">
            <StackLayout Orientation="Horizontal"
                Spacing="15">
                <BoxView Color="Yellow" />
                <Label Text="Yellow"
                    FontSize="Large"
                    VerticalOptions="Center" />
            </StackLayout>
        </Frame>
        <Frame BorderColor="Black"
            Padding="5">
            <StackLayout Orientation="Horizontal"
                Spacing="15">
                <BoxView Color="Blue" />
                <Label Text="Blue"
                    FontSize="Large"
                    VerticalOptions="Center" />
            </StackLayout>
        </Frame>
        ...
    </StackLayout>
</ContentPage>
```

In this example, the parent `StackLayout` contains nested `StackLayout` objects inside `Frame` objects. The parent `StackLayout` is oriented vertically, while the child `StackLayout` objects are oriented horizontally:



IMPORTANT

The deeper you nest `StackLayout` objects and other layouts, the more the nested layouts will impact performance. For more information, see [Choose the correct layout](#).

The equivalent C# code is:

```

public class CombinedStackLayoutPageCS : ContentPage
{
    public CombinedStackLayoutPageCS()
    {
        Title = "Combined StackLayouts demo";
        Content = new StackLayout
        {
            Margin = new Thickness(20),
            Children =
            {
                new Label { Text = "Primary colors" },
                new Frame
                {
                    BorderColor = Color.Black,
                    Padding = new Thickness(5),
                    Content = new StackLayout
                    {
                        Orientation = StackOrientation.Horizontal,
                        Spacing = 15,
                        Children =
                        {
                            new BoxView { Color = Color.Red },
                            new Label { Text = "Red", FontSize = Device.GetNamedSize(NamedSize.Large,
typeof(Label)), VerticalOptions = LayoutOptions.Center }
                        }
                    }
                },
                new Frame
                {
                    BorderColor = Color.Black,
                    Padding = new Thickness(5),
                    Content = new StackLayout
                    {
                        Orientation = StackOrientation.Horizontal,
                        Spacing = 15,
                        Children =
                        {
                            new BoxView { Color = Color.Yellow },
                            new Label { Text = "Yellow", FontSize = Device.GetNamedSize(NamedSize.Large,
typeof(Label)), VerticalOptions = LayoutOptions.Center }
                        }
                    }
                },
                new Frame
                {
                    BorderColor = Color.Black,
                    Padding = new Thickness(5),
                    Content = new StackLayout
                    {
                        Orientation = StackOrientation.Horizontal,
                        Spacing = 15,
                        Children =
                        {
                            new BoxView { Color = Color.Blue },
                            new Label { Text = "Blue", FontSize = Device.GetNamedSize(NamedSize.Large,
typeof(Label)), VerticalOptions = LayoutOptions.Center }
                        }
                    }
                },
                // ...
            };
        };
    }
}

```

Related links

- [StackLayout demos \(sample\)](#)
- [Layout Options in Xamarin.Forms](#)
- [Choose a Xamarin.Forms Layout](#)
- [Improve Xamarin.Forms App Performance](#)

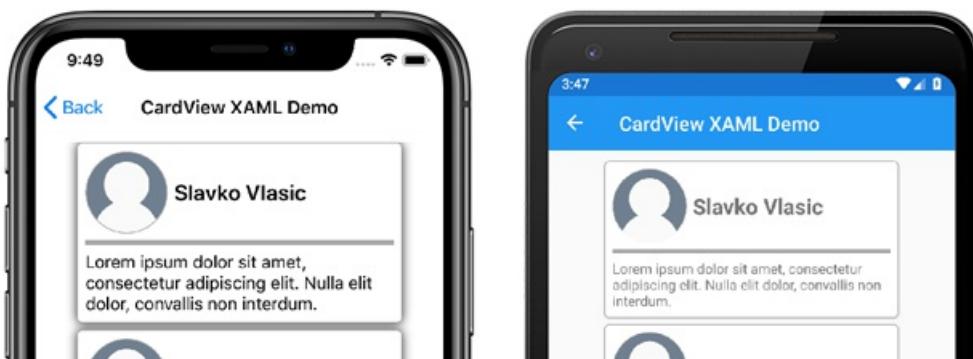
Xamarin.Forms ContentView

8/4/2022 • 4 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `ContentView` class is a type of `Layout` that contains a single child element and is typically used to create custom, reusable controls. The `ContentView` class inherits from `TemplatedView`. This article, and associated sample, explain how to create a custom `CardView` control based on the `ContentView` class.

The following screenshot shows a `CardView` control that derives from the `ContentView` class:



The `ContentView` class defines a single property:

- `Content` is a `View` object. This property is backed by a `BindableProperty` object so it can be the target of data bindings.

The `ContentView` also inherits a property from the `TemplatedView` class:

- `ControlTemplate` is a `ControlTemplate` that can define or override the appearance of the control.

For more information about the `ControlTemplate` property, see [Customize appearance with a ControlTemplate](#).

Create a custom control

The `ContentView` class offers little functionality by itself but can be used to create a custom control. The sample project defines a `CardView` control - a UI element that displays an image, title, and description in a card-like layout.

The process for creating a custom control is to:

1. Create a new class using the `ContentView` template in Visual Studio 2019.
2. Define any unique properties or events in the code-behind file for the new custom control.
3. Create the UI for the custom control.

NOTE

It's possible to create a custom control whose layout is defined in code instead of XAML. For simplicity, the sample application only defines a single `CardView` class with a XAML layout. However, the sample application contains a `CardViewCodePage` class that shows the process of consuming the custom control in code.

Create code-behind properties

The `CardView` custom control defines the following properties:

- `CardTitle` : a `string` object that represents the title shown on the card.
- `CardDescription` : a `string` object that represents the description shown on the card.
- `IconImageSource` : an `ImageSource` object that represents the image shown on the card.
- `IconBackgroundColor` : a `Color` object that represents the background color for the image shown on the card.
- `BorderColor` : a `Color` object that represents the color of the card border, image border, and divider line.
- `CardColor` : a `Color` object that represents the background color of the card.

NOTE

The `BorderColor` property affects multiple items for the purposes of demonstration. This property could be broken out into three properties if needed.

Each property is backed by a `BindableProperty` instance. The backing `BindableProperty` allows each property to be styled, and bound, using the MVVM pattern.

The following example shows how to create a backing `BindableProperty`:

```
public static readonly BindableProperty CardTitleProperty = BindableProperty.Create(
    "CardTitle",           // the name of the bindable property
    typeof(string),        // the bindable property type
    typeof(CardView),      // the parent object type
    string.Empty);         // the default value for the property
```

The custom property uses the `GetValue` and `SetValue` methods to get and set the `BindableProperty` object values:

```
public string CardTitle
{
    get => (string)GetValue(CardView.CardTitleProperty);
    set => SetValue(CardView.CardTitleProperty, value);
}
```

For more information about `BindableProperty` objects, see [Bindable Properties](#).

Define UI

The custom control UI uses a `ContentView` as the root element for the `CardView` control. The following example shows the `CardView` XAML:

```

<ContentView ...
    x:Name="this"
    x:Class="CardViewDemo.Controls.CardView">
    <Frame BindingContext="{x:Reference this}"
        BackgroundColor="{Binding CardColor}"
        BorderColor="{Binding BorderColor}"
        ...>
        <Grid>
            ...
            <Frame BorderColor="{Binding BorderColor, FallbackValue='Black'}"
                BackgroundColor="{Binding IconBackgroundColor, FallbackValue='Grey'}"
                ...>
                <Image Source="{Binding IconImageSource}" ...
                    ... />
            </Frame>
            <Label Text="{Binding CardTitle, FallbackValue='Card Title'}"
                ... />
            <BoxView BackgroundColor="{Binding BorderColor, FallbackValue='Black'}"
                ... />
            <Label Text="{Binding CardDescription, FallbackValue='Card description text.'}"
                ... />
        </Grid>
    </Frame>
</ContentView>

```

The `ContentView` element sets the `x:Name` property to `this`, which can be used to access the object bound to the `CardView` instance. Elements in the layout set bindings on their properties to values defined on the bound object.

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

NOTE

The `FallbackValue` property provides a default value in case the binding is `null`. This also allows the [XAML Previewer](#) in Visual Studio to render the `CardView` control.

Instantiate a custom control

A reference to the custom control namespace must be added to a page that instantiates the custom control. The following example shows a namespace reference called `controls` added to a `ContentPage` instance in XAML:

```

<ContentPage ...
    xmlns:controls="clr-namespace:CardViewDemo.Controls" >

```

Once the reference has been added the `CardView` can be instantiated in XAML, and its properties defined:

```

<controls:CardView BorderColor="DarkGray"
    CardTitle="Slavko Vlasic"
    CardDescription="Lorem ipsum dolor sit..."
    IconBackgroundColor="SlateGray"
    IconImageSource="user.png"/>

```

A `CardView` can also be instantiated in code:

```

CardView card = new CardView
{
    BorderColor = Color.DarkGray,
    CardTitle = "Slavko Vlasic",
    CardDescription = "Lorem ipsum dolor sit...",
    IconBackgroundColor = Color.SlateGray,
    IconImageSource = ImageSource.FromFile("user.png")
};

```

Customize appearance with a ControlTemplate

A custom control that derives from the `ContentView` class can define appearance using XAML, code, or may not define appearance at all. Regardless of how appearance is defined, a `ControlTemplate` object can override the appearance with a custom layout.

The `cardView` layout might occupy too much space for some use cases. A `ControlTemplate` can override the `CardView` layout to provide a more compact view, suitable for a condensed list:

```

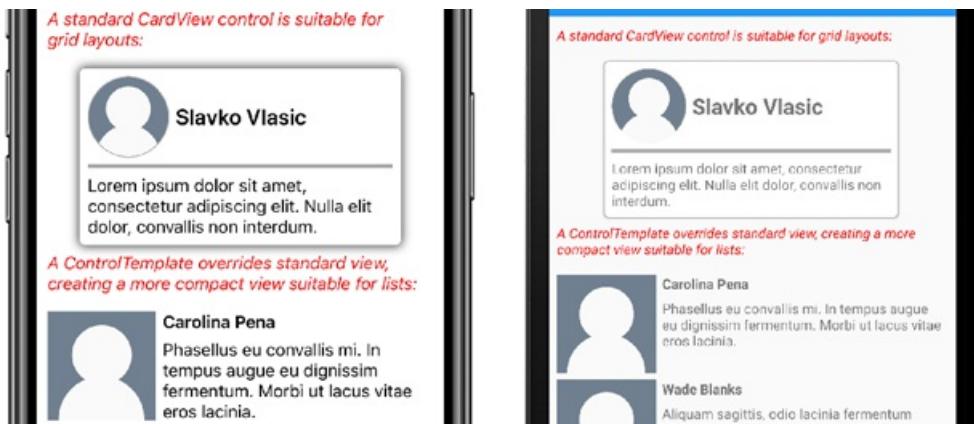
<ContentPage.Resources>
    <ResourceDictionary>
        <ControlTemplate x:Key="CardViewCompressed">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="100" />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="100" />
                    <ColumnDefinition Width="100*" />
                </Grid.ColumnDefinitions>
                <Image Grid.Row="0"
                    Grid.Column="0"
                    Source="{TemplateBinding IconImageSource}"
                    BackgroundColor="{TemplateBinding IconBackgroundColor}"
                    WidthRequest="100"
                    HeightRequest="100"
                    Aspect="AspectFill"
                    HorizontalOptions="Center"
                    VerticalOptions="Center"/>
                <StackLayout Grid.Row="0"
                    Grid.Column="1">
                    <Label Text="{TemplateBinding CardTitle}"
                        FontAttributes="Bold" />
                    <Label Text="{TemplateBinding CardDescription}" />
                </StackLayout>
            </Grid>
        </ControlTemplate>
    </ResourceDictionary>
</ContentPage.Resources>

```

Data binding in a `ControlTemplate` uses the `TemplateBinding` markup extension to specify bindings. The `ControlTemplate` property can then be set to the defined `ControlTemplate` object, by using its `x:key` value. The following example shows the `ControlTemplate` property set on a `cardView` instance:

```
<controls:CardView ControlTemplate="{StaticResource CardViewCompressed}" />
```

The following screenshots show a standard `CardView` instance and `CardView` whose `controlTemplate` has been overridden:



For more information about control templates, see [Xamarin.Forms Control Templates](#).

Related links

- [ContentView sample application](#)
- [Xamarin.Forms Data Binding](#)
- [Bindable Properties](#).
- [Xamarin.Forms Control Templates](#)

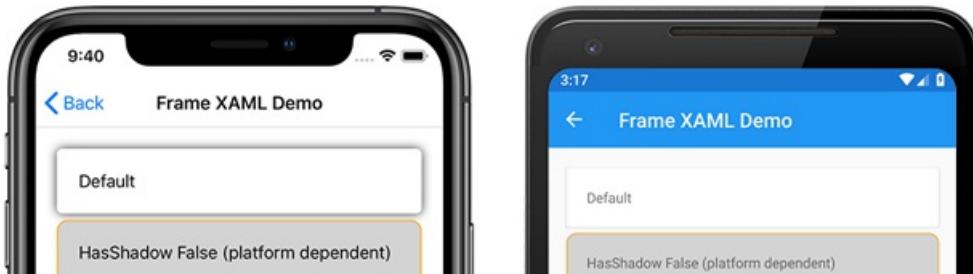
Xamarin.Forms Frame

8/4/2022 • 3 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Xamarin.Forms `Frame` class is a layout used to wrap a view with a border that can be configured with color, shadow, and other options. Frames are commonly used to create borders around controls but can be used to create more complex UI. For more information, see [Advanced Frame usage](#).

The following screenshot shows `Frame` controls on iOS and Android:



The `Frame` class defines the following properties:

- `BorderColor` is a `Color` value that determines the color of the `Frame` border.
- `CornerRadius` is a `float` value that determines the rounded radius of the corner.
- `HasShadow` is a `bool` value that determines whether the frame has a drop shadow.

These properties are backed by `BindableProperty` objects, which means the `Frame` can be the target of data bindings.

NOTE

The `HasShadow` property behavior is platform-dependent. The default value is `true` on all platforms. However, on UWP drop shadows are not rendered. Drop shadows are rendered on both Android and iOS but drop shadows on iOS are darker and occupy more space.

Create a Frame

A `Frame` can be instantiated in XAML. The default `Frame` object has a white background, a drop shadow, and no border. A `Frame` object typically wraps another control. The following example shows a default `Frame` wrapping a `Label` object:

```
<Frame>
    <Label Text="Example" />
</Frame>
```

A `Frame` can also be created in code:

```
Frame defaultFrame = new Frame
{
    Content = new Label { Text = "Example" }
};
```

`Frame` objects can be customized with rounded corners, colorized borders, and drop shadows by setting properties in the XAML. The following example shows a customized `Frame` object:

```
<Frame BorderColor="Orange"
    CornerRadius="10"
    HasShadow="True">
    <Label Text="Example" />
</Frame>
```

These instance properties can also be set in code:

```
Frame frame = new Frame
{
    BorderColor = Color.Orange,
    CornerRadius = 10,
    HasShadow = true,
    Content = new Label { Text = "Example" }
};
```

Advanced Frame usage

The `Frame` class inherits from `ContentView`, which means it can contain any type of `View` object including `Layout` objects. This ability allows the `Frame` to be used to create complex UI objects such as cards.

Create a card with a Frame

Combining a `Frame` object with a `Layout` object such as a `StackLayout` object allows the creation of more complex UI. The following screenshot shows an example card, created using a `Frame` object:

Card Example

Frames can wrap more complex layouts to create more complex UI components, such as this card!

The following XAML shows how to create a card with the `Frame` class:

```
<Frame BorderColor="Gray"
    CornerRadius="5"
    Padding="8">
    <StackLayout>
        <Label Text="Card Example"
            FontSize="Medium"
            FontAttributes="Bold" />
        <BoxView Color="Gray"
            HeightRequest="2"
            HorizontalOptions="Fill" />
        <Label Text="Frames can wrap more complex layouts to create more complex UI components, such as this
card!"/>
    </StackLayout>
</Frame>
```

A card can also be created in code:

```

Frame cardFrame = new Frame
{
    BorderColor = Color.Gray,
    CornerRadius = 5,
    Padding = 8,
    Content = new StackLayout
    {
        Children =
        {
            new Label
            {
                Text = "Card Example",
                FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(Label)),
                FontAttributes = FontAttributes.Bold
            },
            new BoxView
            {
                Color = Color.Gray,
                HeightRequest = 2,
                HorizontalOptions = LayoutOptions.Fill
            },
            new Label
            {
                Text = "Frames can wrap more complex layouts to create more complex UI components, such as  
this card!"
            }
        }
    }
};


```

Round elements

The `cornerRadius` property of the `Frame` control can be used to create a circle image. The following screenshot shows an example of a round image, created using a `Frame` object:



The following XAML shows how to create a circle image in XAML:

```

<Frame Margin="10"
    BorderColor="Black"
    CornerRadius="50"
    HeightRequest="60"
    WidthRequest="60"
    IsClippedToBounds="True"
    HorizontalOptions="Center"
    VerticalOptions="Center">
    <Image Source="outdoors.jpg"
        Aspect="AspectFill"
        Margin="-20"
        HeightRequest="100"
        WidthRequest="100" />
</Frame>

```

A circle image can also be created in code:

```
Frame circleImageFrame = new Frame
{
    Margin = 10,
    BorderColor = Color.Black,
    CornerRadius = 50,
    HeightRequest = 60,
    WidthRequest = 60,
    IsClippedToBounds = true,
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center,
    Content = new Image
    {
        Source = ImageSource.FromFile("outdoors.jpg"),
        Aspect = Aspect.AspectFill,
        Margin = -20,
        HeightRequest = 100,
        WidthRequest = 100
    }
};

});
```

The `outdoors.jpg` image must be added to each platform project, and how this is achieved varies by platform. For more information, see [Images in Xamarin.Forms](#).

NOTE

Rounded corners behave slightly differently across platforms. The `Image` object's `Margin` should be half of the difference between the image width and the parent frame width, and should be negative to center the image evenly within the `Frame` object. However, the width and height requested are not guaranteed, so the `Margin`, `HeightRequest` and `WidthRequest` properties may need to be altered based on your image size and other layout choices.

Related links

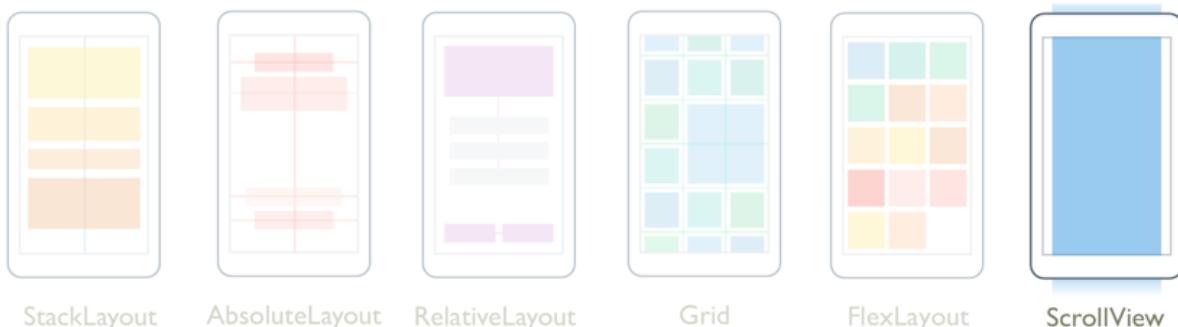
- [Frame Demos](#)
- [Images in Xamarin.Forms](#)

Xamarin.Forms ScrollView

8/4/2022 • 9 minutes to read • [Edit Online](#)



[Download the sample](#)



StackLayout

AbsoluteLayout

RelativeLayout

Grid

FlexLayout

ScrollView

`ScrollView` is a layout that's capable of scrolling its content. The `ScrollView` class derives from the `Layout` class, and by default scrolls its content vertically. A `ScrollView` can only have a single child, although this can be other layouts.

WARNING

`ScrollView` objects should not be nested. In addition, `ScrollView` objects should not be nested with other controls that provide scrolling, such as `CollectionView`, `ListView`, and `WebView`.

`ScrollView` defines the following properties:

- `Content`, of type `View`, represents the content to display in the `ScrollView`.
- `ContentSize`, of type `Size`, represents the size of the content. This is a read-only property.
- `HorizontalScrollBarVisibility`, of type `ScrollBarVisibility`, represents when the horizontal scroll bar is visible.
- `Orientation`, of type `ScrollOrientation`, represents the scrolling direction of the `ScrollView`. The default value of this property is `Vertical`.
- `ScrollX`, of type `double`, indicates the current X scroll position. The default value of this read-only property is 0.
- `ScrollY`, of type `double`, indicates the current Y scroll position. The default value of this read-only property is 0.
- `VerticalScrollBarVisibility`, of type `ScrollBarVisibility`, represents when the vertical scroll bar is visible.

These properties are backed by `BindableProperty` objects, with the exception of the `Content` property, which means that they can be targets of data bindings and styled.

The `Content` property is the `ContentProperty` of the `ScrollView` class, and therefore does not need to be explicitly set from XAML.

TIP

To obtain the best possible layout performance, follow the guidelines at [Optimize layout performance](#).

ScrollView as a root layout

A `ScrollView` can only have a single child, which can be other layouts. It's therefore common for a `ScrollView` to be the root layout on a page. To scroll its child content, `ScrollView` computes the difference between the height of its content and its own height. That difference is the amount that the `ScrollView` can scroll its content.

A `StackLayout` will often be the child of a `ScrollView`. In this scenario, the `ScrollView` causes the `StackLayout` to be as tall as the sum of the heights of its children. Then the `ScrollView` can determine the amount that its content can be scrolled. For more information about the `StackLayout`, see [Xamarin.Forms StackLayout](#).

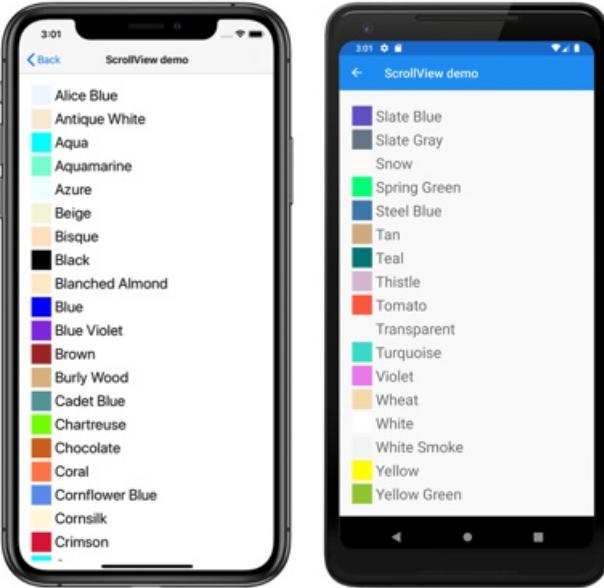
Caution

In a vertical `ScrollView`, avoid setting the `VerticalOptions` property to `Start`, `Center`, or `End`. Doing so tells the `ScrollView` to be only as tall as it needs to be, which could be zero. While Xamarin.Forms protects against this eventuality, it's best to avoid code that suggests something you don't want to happen.

The following XAML example has a `ScrollView` as a root layout on a page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:ScrollViewDemos"
    x:Class="ScrollViewDemos.Views.ColorListPage"
    Title="ScrollView demo">
    <ScrollView>
        <StackLayout BindableLayout.ItemsSource="{x:Static local:NamedColor.All}">
            <BindableLayout.ItemTemplate>
                <DataTemplate>
                    <StackLayout Orientation="Horizontal">
                        <BoxView Color="{Binding Color}"
                            HeightRequest="32"
                            WidthRequest="32"
                            VerticalOptions="Center" />
                        <Label Text="{Binding FriendlyName}"
                            FontSize="24"
                            VerticalOptions="Center" />
                    </StackLayout>
                </DataTemplate>
            </BindableLayout.ItemTemplate>
        </StackLayout>
    </ScrollView>
</ContentPage>
```

In this example, the `ScrollView` has its content set to a `StackLayout` that uses a bindable layout to display the `Color` fields defined by Xamarin.Forms. By default, a `ScrollView` scrolls vertically, which reveals more content:



The equivalent C# code is:

```
public class ColorListPageCode : ContentPage
{
    public ColorListPageCode()
    {
        DataTemplate dataTemplate = new DataTemplate(() =>
        {
            BoxView boxView = new BoxView
            {
                HeightRequest = 32,
                WidthRequest = 32,
                VerticalOptions = LayoutOptions.Center
            };
            boxView.SetBinding(BoxView.ColorProperty, "Color");

            Label label = new Label
            {
                FontSize = 24,
                VerticalOptions = LayoutOptions.Center
            };
            label.SetBinding(Label.TextProperty, "FriendlyName");

            StackLayout horizontalStackLayout = new StackLayout
            {
                Orientation = StackOrientation.Horizontal,
                Children = { boxView, label }
            };
            return horizontalStackLayout;
        });

        StackLayout stackLayout = new StackLayout();
        BindableLayout.SetItemsSource(stackLayout, NamedColor.All);
        BindableLayout.SetItemTemplate(stackLayout, dataTemplate);

        ScrollView scrollView = new ScrollView { Content = stackLayout };

        Title = "ScrollView demo";
        Content = scrollView;
    }
}
```

For more information about bindable layouts, see [Bindable Layouts in Xamarin.Forms](#).

ScrollView as a child layout

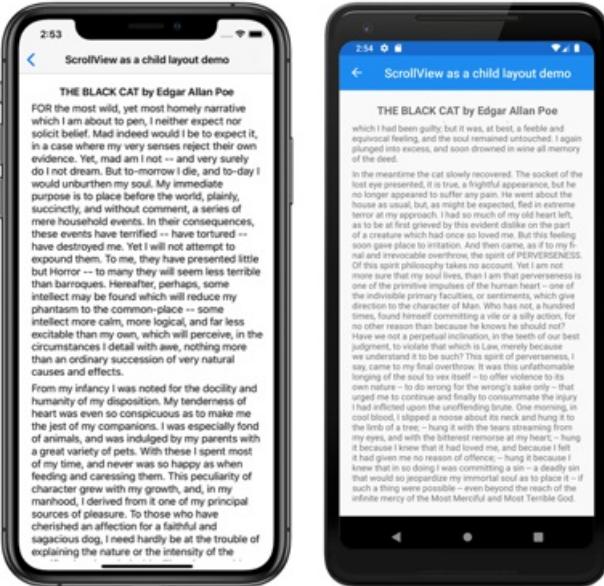
A `ScrollView` can be a child layout to a different parent layout.

A `ScrollView` will often be the child of a `StackLayout`. A `ScrollView` requires a specific height to compute the difference between the height of its content and its own height, with the difference being the amount that the `ScrollView` can scroll its content. When a `ScrollView` is the child of a `StackLayout`, it doesn't receive a specific height. The `StackLayout` wants the `ScrollView` to be as short as possible, which is either the height of the `ScrollView` contents or zero. To handle this scenario, the `VerticalOptions` property of the `ScrollView` should be set to `FillAndExpand`. This will cause the `StackLayout` to give the `ScrollView` all the extra space not required by the other children, and the `ScrollView` will then have a specific height.

The following XAML example has a `ScrollView` as a child layout to a `StackLayout`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ScrollViewDemos.Views.BlackCatPage"
    Title="ScrollView as a child layout demo">
    <StackLayout Margin="20">
        <Label Text="THE BLACK CAT by Edgar Allan Poe"
            FontSize="Medium"
            FontAttributes="Bold"
            HorizontalOptions="Center" />
        <ScrollView VerticalOptions="FillAndExpand">
            <StackLayout>
                <Label Text="FOR the most wild, yet most homely narrative which I am about to pen, I neither expect nor solicit belief. Mad indeed would I be to expect it, in a case where my very senses reject their own evidence. Yet, mad am I not -- and very surely do I not dream. But to-morrow I die, and to-day I would unburthen my soul. My immediate purpose is to place before the world, plainly, succinctly, and without comment, a series of mere household events. In their consequences, these events have terrified -- have tortured -- have destroyed me. Yet I will not attempt to expound them. To me, they have presented little but Horror -- to many they will seem less terrible than barroques. Hereafter, perhaps, some intellect may be found which will reduce my phantasm to the common-place -- some intellect more calm, more logical, and far less excitable than my own, which will perceive, in the circumstances I detail with awe, nothing more than an ordinary succession of very natural causes and effects." />
                <!-- More Label objects go here -->
            </StackLayout>
        </ScrollView>
    </StackLayout>
</ContentPage>
```

In this example, there are two `StackLayout` objects. The first `StackLayout` is the root layout object, which has a `Label` object and a `ScrollView` as its children. The `ScrollView` has a `StackLayout` as its content, with the `StackLayout` containing multiple `Label` objects. This arrangement ensures that the first `Label` is always on-screen, while text displayed by the other `Label` objects can be scrolled:



The equivalent C# code is:

```

public class BlackCatPageCS : ContentPage
{
    public BlackCatPageCS()
    {
        Label titleLabel = new Label
        {
            Text = "THE BLACK CAT by Edgar Allan Poe",
            // More properties set here to define the Label appearance
        };

        ScrollView scrollView = new ScrollView
        {
            VerticalOptions = LayoutOptions.FillAndExpand,
            Content = new StackLayout
            {
                Children =
                {
                    new Label
                    {
                        Text = "FOR the most wild, yet most homely narrative which I am about to pen, I
neither expect nor solicit belief. Mad indeed would I be to expect it, in a case where my very senses reject
their own evidence. Yet, mad am I not -- and very surely do I not dream. But to-morrow I die, and to-day I
would unburthen my soul. My immediate purpose is to place before the world, plainly, succinctly, and without
comment, a series of mere household events. In their consequences, these events have terrified -- have
tortured -- have destroyed me. Yet I will not attempt to expound them. To me, they have presented little but
Horror -- to many they will seem less terrible than barroques. Hereafter, perhaps, some intellect may be
found which will reduce my phantasm to the common-place -- some intellect more calm, more logical, and far
less excitable than my own, which will perceive, in the circumstances I detail with awe, nothing more than
an ordinary succession of very natural causes and effects."
                },
                // More Label objects go here
            }
        };
    }

    Title = "ScrollView as a child layout demo";
    Content = new StackLayout
    {
        Margin = new Thickness(20),
        Children = { titleLabel, scrollView }
    };
}

```

Orientation

`ScrollView` has an `Orientation` property, which represents the scrolling direction of the `ScrollView`. This property is of type `ScrollOrientation`, which defines the following members:

- `Vertical` indicates that the `ScrollView` will scroll vertically. This member is the default value of the `Orientation` property.
- `Horizontal` indicates that the `ScrollView` will scroll horizontally.
- `Both` indicates that the `ScrollView` will scroll horizontally and vertically.
- `Neither` indicates that the `ScrollView` won't scroll.

TIP

Scrolling can be disabled by setting the `Orientation` property to `Neither`.

Detect scrolling

`ScrollView` defines a `Scrolled` event that is fired to indicate that scrolling occurred. The `ScrolledEventArgs` object that accompanies the `Scrolled` event has `ScrollX` and `ScrollY` properties, both of type `double`.

IMPORTANT

The `ScrolledEventArgs.ScrollX` and `ScrolledEventArgs.ScrollY` properties can have negative values, due to the bounce effect that occurs when scrolling back to the start of a `ScrollView`.

The following XAML example shows a `ScrollView` that sets an event handler for the `Scrolled` event:

```
<ScrollView Scrolled="OnScrollViewScrolled">
    ...
</ScrollView>
```

The equivalent C# code is:

```
ScrollView scrollView = new ScrollView();
scrollView.Scrolled += OnScrollViewScrolled;
```

In this example, the `OnScrollViewScrolled` event handler is executed when the `Scrolled` event fires:

```
void OnScrollViewScrolled(object sender, ScrolledEventArgs e)
{
    Console.WriteLine($"ScrollX: {e.ScrollX}, ScrollY: {e.ScrollY}");
}
```

In this example, the `OnScrollViewScrolled` event handler outputs the values of the `ScrolledEventArgs` object that accompanies the event.

NOTE

The `Scrolled` event is fired for user initiated scrolls, and for programmatic scrolls.

Scroll programmatically

`ScrollView` defines two `ScrollToAsync` methods, that asynchronously scroll the `ScrollView`. One of the overloads scrolls to a specified position in the `ScrollView`, while the other scrolls a specified element into view. Both overloads have an additional argument that can be used to indicate whether to animate the scroll.

IMPORTANT

The `ScrollToAsync` methods will not result in scrolling when the `ScrollView.Orientation` property is set to `Neither`.

Scroll a position into view

A position within a `ScrollView` can be scrolled to with the `ScrollToAsync` method that accepts `double` `x` and `y` arguments. Given a vertical `ScrollView` object named `scrollView`, the following example shows how to scroll to 150 device-independent units from the top of the `ScrollView`:

```
await scrollView.ScrollToAsync(0, 150, true);
```

The third argument to the `ScrollToAsync` is the `animated` argument, which determines whether a scrolling animation is displayed when programmatically scrolling a `ScrollView`.

Scroll an element into view

An element within a `ScrollView` can be scrolled into view with the `ScrollToAsync` method that accepts `Element` and `ScrollToPosition` arguments. Given a vertical `ScrollView` named `scrollView`, and a `Label` named `label`, the following example shows how to scroll an element into view:

```
await scrollView.ScrollToAsync(label, ScrollToPosition.End, true);
```

The third argument to the `ScrollToAsync` is the `animated` argument, which determines whether a scrolling animation is displayed when programmatically scrolling a `ScrollView`.

When scrolling an element into view, the exact position of the element after the scroll has completed can be set with the second argument, `position`, of the `ScrollToAsync` method. This argument accepts a `ScrollToPosition` enumeration member:

- `MakeVisible` indicates that the element should be scrolled until it's visible in the `ScrollView`.
- `Start` indicates that the element should be scrolled to the start of the `ScrollView`.
- `Center` indicates that the element should be scrolled to the center of the `ScrollView`.
- `End` indicates that the element should be scrolled to the end of the `ScrollView`.

Scroll bar visibility

`ScrollView` defines `HorizontalScrollBarVisibility` and `VerticalScrollBarVisibility` properties, which are backed by bindable properties. These properties get or set a `ScrollBarVisibility` enumeration value that represents whether the horizontal, or vertical, scroll bar is visible. The `ScrollBarVisibility` enumeration defines the following members:

- `Default` indicates the default scroll bar behavior for the platform, and is the default value of the `HorizontalScrollBarVisibility` and `VerticalScrollBarVisibility` properties.
- `Always` indicates that scroll bars will be visible, even when the content fits in the view.
- `Never` indicates that scroll bars will not be visible, even if the content doesn't fit in the view.

Related links

- [ScrollView demos \(sample\)](#)
- [Xamarin.Forms StackLayout](#)
- [Bindable Layouts in Xamarin.Forms](#)

Bindable Layouts in Xamarin.Forms

8/4/2022 • 7 minutes to read • [Edit Online](#)



[Download the sample](#)

Bindable layouts enable any layout class that derives from the `Layout<T>` class to generate its content by binding to a collection of items, with the option to set the appearance of each item with a `DataTemplate`.

Bindable layouts are provided by the `BindableLayout` class, which exposes the following attached properties:

- `ItemsSource` – specifies the collection of `IEnumerable` items to be displayed by the layout.
- `ItemTemplate` – specifies the `DataTemplate` to apply to each item in the collection of items displayed by the layout.
- `ItemTemplateSelector` – specifies the `DataTemplateSelector` that will be used to choose a `DataTemplate` for an item at runtime.

NOTE

The `ItemTemplate` property takes precedence when both the `ItemTemplate` and `ItemTemplateSelector` properties are set.

In addition, the `BindableLayout` class exposes the following bindable properties:

- `EmptyView` – specifies the `string` or view that will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The default value is `null`.
- `EmptyViewTemplate` – specifies the `DataTemplate` that will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The default value is `null`.

NOTE

The `EmptyViewTemplate` property takes precedence when both the `EmptyView` and `EmptyViewTemplate` properties are set.

All of these properties can be attached to the `AbsoluteLayout`, `FlexLayout`, `Grid`, `RelativeLayout`, and `StackLayout` classes, which all derive from the `Layout<T>` class.

The `Layout<T>` class exposes a `Children` collection, to which the child elements of a layout are added. When the `BindableLayout.ItemsSource` property is set to a collection of items and attached to a `Layout<T>`-derived class, each item in the collection is added to the `Layout<T>.Children` collection for display by the layout. The `Layout<T>`-derived class will then update its child views when the underlying collection changes. For more information about the Xamarin.Forms layout cycle, see [Creating a Custom Layout](#).

Bindable layouts should only be used when the collection of items to be displayed is small, and scrolling and selection isn't required. While scrolling can be provided by wrapping a bindable layout in a `ScrollView`, this is not recommended as bindable layouts lack UI virtualization. When scrolling is required, a scrollable view that includes UI virtualization, such as `ListView` or `CollectionView`, should be used. Failure to observe this recommendation can lead to performance issues.

IMPORTANT

While it's technically possible to attach a bindable layout to any layout class that derives from the `Layout<T>` class, it's not always practical to do so, particularly for the `AbsoluteLayout`, `Grid`, and `RelativeLayout` classes. For example, consider the scenario of wanting to display a collection of data in a `Grid` using a bindable layout, where each item in the collection is an object containing multiple properties. Each row in the `Grid` should display an object from the collection, with each column in the `Grid` displaying one of the object's properties. Because the `DataTemplate` for the bindable layout can only contain a single object, it's necessary for that object to be a layout class containing multiple views that each display one of the object's properties in a specific `Grid` column. While this scenario can be realised with bindable layouts, it results in a parent `Grid` containing a child `Grid` for each item in the bound collection, which is a highly inefficient and problematic use of the `Grid` layout.

Populate a bindable layout with data

A bindable layout is populated with data by setting its `ItemsSource` property to any collection that implements `IEnumerable`, and attaching it to a `Layout<T>`-derived class:

```
<Grid BindableLayout.ItemsSource="{Binding Items}" />
```

The equivalent C# code is:

```
IEnumerable<string> items = ...;
var grid = new Grid();
BindableLayout.SetItemsSource(grid, items);
```

When the `BindableLayout.ItemsSource` attached property is set on a layout, but the `BindableLayout.ItemTemplate` attached property isn't set, every item in the `IEnumerable` collection will be displayed by a `Label` that's created by the `BindableLayout` class.

Define item appearance

The appearance of each item in the bindable layout can be defined by setting the `BindableLayout.ItemTemplate` attached property to a `DataTemplate`:

```
<StackLayout BindableLayout.ItemsSource="{Binding User.TopFollowers}"
            Orientation="Horizontal"
            ...
            >
    <BindableLayout.ItemTemplate>
        <DataTemplate>
            <controls:CircleImage Source="{Binding}"
                                  Aspect="AspectFill"
                                  WidthRequest="44"
                                  HeightRequest="44"
                                  ...
                                  />
        </DataTemplate>
    </BindableLayout.ItemTemplate>
</StackLayout>
```

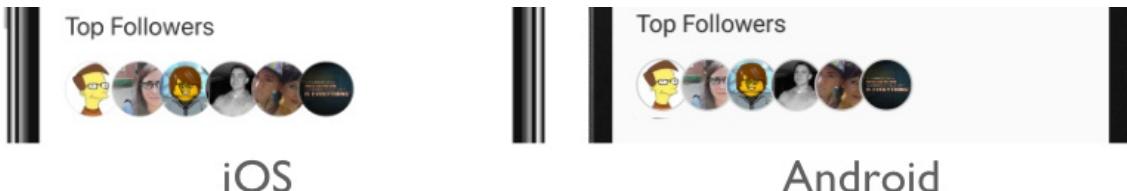
The equivalent C# code is:

```

DataTemplate circleImageTemplate = ...;
var stackLayout = new StackLayout();
BindableLayout.SetItemsSource(stackLayout, viewModel.User.TopFollowers);
BindableLayout.SetItemTemplate(stackLayout, circleImageTemplate);

```

In this example, every item in the `TopFollowers` collection will be displayed by a `CircleImage` view defined in the `DataTemplate`:



For more information about data templates, see [Xamarin.Forms Data Templates](#).

Choose item appearance at runtime

The appearance of each item in the bindable layout can be chosen at runtime, based on the item value, by setting the `BindableLayout.ItemTemplateSelector` attached property to a `DataTemplateSelector`:

```

<FlexLayout BindableLayout.ItemsSource="{Binding User.FavoriteTech}"
           BindableLayout.ItemTemplateSelector="{StaticResource TechItemTemplateSelector}"
           ... />

```

The equivalent C# code is:

```

DataTemplateSelector dataTemplateSelector = new TechItemTemplateSelector { ... };
var flexLayout = new FlexLayout();
BindableLayout.SetItemsSource(flexLayout, viewModel.User.FavoriteTech);
BindableLayout.SetItemTemplateSelector(flexLayout, dataTemplateSelector);

```

The `DataTemplateSelector` used in the sample application is shown in the following example:

```

public class TechItemTemplateSelector : DataTemplateSelector
{
    public DataTemplate DefaultTemplate { get; set; }
    public DataTemplate XamarinFormsTemplate { get; set; }

    protected override DataTemplate OnSelectTemplate(object item, BindableObject container)
    {
        return (string)item == "Xamarin.Forms" ? XamarinFormsTemplate : DefaultTemplate;
    }
}

```

The `TechItemTemplateSelector` class defines `DefaultTemplate` and `XamarinFormsTemplate` `DataTemplate` properties that are set to different data templates. The `OnSelectTemplate` method returns the `XamarinFormsTemplate`, which displays an item in dark red with a heart next to it, when the item is equal to "Xamarin.Forms". When the item isn't equal to "Xamarin.Forms", the `OnSelectTemplate` method returns the `DefaultTemplate`, which displays an item using the default color of a `Label`:



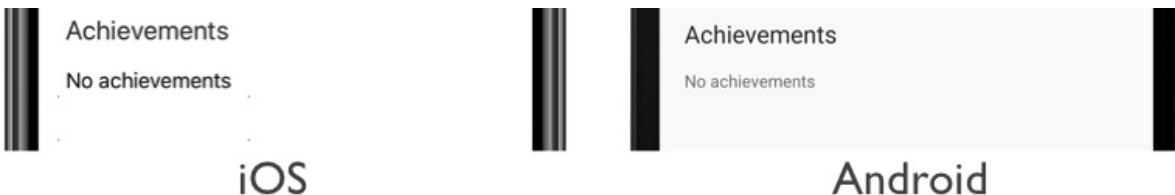
For more information about data template selectors, see [Creating a Xamarin.Forms DataTemplateSelector](#).

Display a string when data is unavailable

The `EmptyView` property can be set to a string, which will be displayed by a `Label` when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The following XAML shows an example of this scenario:

```
<StackLayout BindableLayout.ItemsSource="{Binding UserWithoutAchievements.Achievements}">
    BindableLayout.EmptyView="No achievements"
    ...
</StackLayout>
```

The result is that when the data bound collection is `null`, the string set as the `EmptyView` property value is displayed:

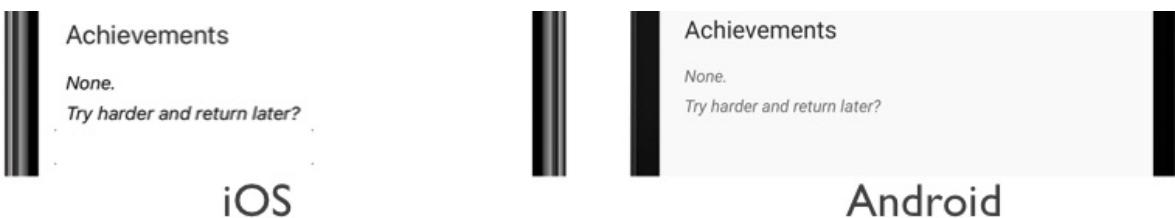


Display views when data is unavailable

The `EmptyView` property can be set to a view, which will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. This can be a single view, or a view that contains multiple child views. The following XAML example shows the `EmptyView` property set to a view that contains multiple child views:

```
<StackLayout BindableLayout.ItemsSource="{Binding UserWithoutAchievements.Achievements}">
    <BindableLayout.EmptyView>
        <StackLayout>
            <Label Text="None."
                  FontAttributes="Italic"
                  FontSize="{StaticResource smallTextSize}" />
            <Label Text="Try harder and return later?"
                  FontAttributes="Italic"
                  FontSize="{StaticResource smallTextSize}" />
        </StackLayout>
    </BindableLayout.EmptyView>
    ...
</StackLayout>
```

The result is that when the data bound collection is `null`, the `StackLayout` and its child views are displayed.

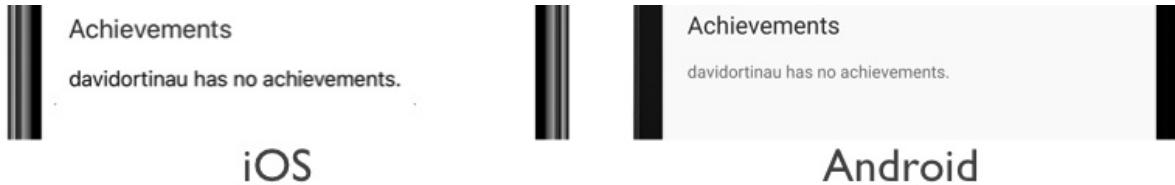


Similarly, the `EmptyViewTemplate` can be set to a `DataTemplate`, which will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The `DataTemplate` can contain a single view, or a view that contains multiple child views. In addition, the `BindingContext` of the `EmptyViewTemplate` will be inherited from the `BindingContext` of the `BindableLayout`. The

following XAML example shows the `EmptyViewTemplate` property set to a `DataTemplate` that contains a single view:

```
<StackLayout BindableLayout.ItemsSource="{Binding UserWithoutAchievements.Achievements}">
    <BindableLayout.EmptyViewTemplate>
        <DataTemplate>
            <Label Text="{Binding Source={x:Reference usernameLabel}, Path=Text, StringFormat='{0} has no achievements.'}" />
        </DataTemplate>
    </BindableLayout.EmptyViewTemplate>
    ...
</StackLayout>
```

The result is that when the data bound collection is `null`, the `Label` in the `DataTemplate` is displayed:



NOTE

The `EmptyViewTemplate` property can't be set via a `DataTemplateSelector`.

Choose an EmptyView at runtime

Views that will be displayed as an `EmptyView` when data is unavailable, can be defined as `ContentView` objects in a `ResourceDictionary`. The `EmptyView` property can then be set to a specific `ContentView`, based on some business logic, at runtime. The following XAML shows an example of this scenario:

```

<ContentPage ...>
    <ContentPage.Resources>
        ...
        <ContentView x:Key="BasicEmptyView">
            <StackLayout>
                <Label Text="No achievements."
                    FontSize="14" />
            </StackLayout>
        </ContentView>
        <ContentView x:Key="AdvancedEmptyView">
            <StackLayout>
                <Label Text="None."
                    FontAttributes="Italic"
                    FontSize="14" />
                <Label Text="Try harder and return later?"
                    FontAttributes="Italic"
                    FontSize="14" />
            </StackLayout>
        </ContentView>
    </ContentPage.Resources>

    <StackLayout>
        ...
        <Switch Toggled="OnEmptyViewSwitchToggled" />

        <StackLayout x:Name="stackLayout"
            BindableLayout.ItemsSource="{Binding UserWithoutAchievements.Achievements}">
            ...
        </StackLayout>
    </StackLayout>
</ContentPage>

```

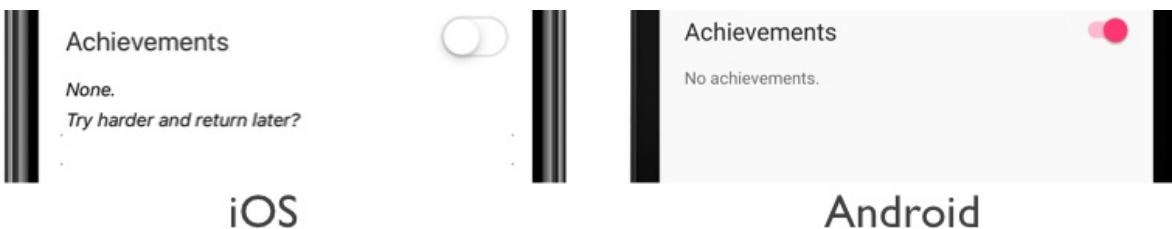
The XAML defines two `ContentView` objects in the page-level `ResourceDictionary`, with the `Switch` object controlling which `ContentView` object will be set as the `EmptyView` property value. When the `Switch` is toggled, the `OnEmptyViewSwitchToggled` event handler executes the `ToggleEmptyView` method:

```

void ToggleEmptyView(bool isToggled)
{
    object view = isToggled ? Resources["BasicEmptyView"] : Resources["AdvancedEmptyView"];
    BindableLayout.SetEmptyView(stackLayout, view);
}

```

The `ToggleEmptyView` method sets the `EmptyView` property of the `stackLayout` object to one of the two `ContentView` objects stored in the `ResourceDictionary`, based on the value of the `Switch.IsToggled` property. Then, when the data bound collection is `null`, the `ContentView` object set as the `EmptyView` property is displayed:



Related links

- [Bindable Layout Demo \(sample\)](#)
- [Creating a Custom Layout](#)
- [Xamarin.Forms Data Templates](#)

- [Creating a Xamarin.Forms DataTemplateSelector](#)

Create a Custom Layout in Xamarin.Forms

8/4/2022 • 14 minutes to read • [Edit Online](#)



[Download the sample](#)

Xamarin.Forms defines five layout classes – `StackLayout`, `AbsoluteLayout`, `RelativeLayout`, `Grid`, and `FlexLayout`, 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.

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.

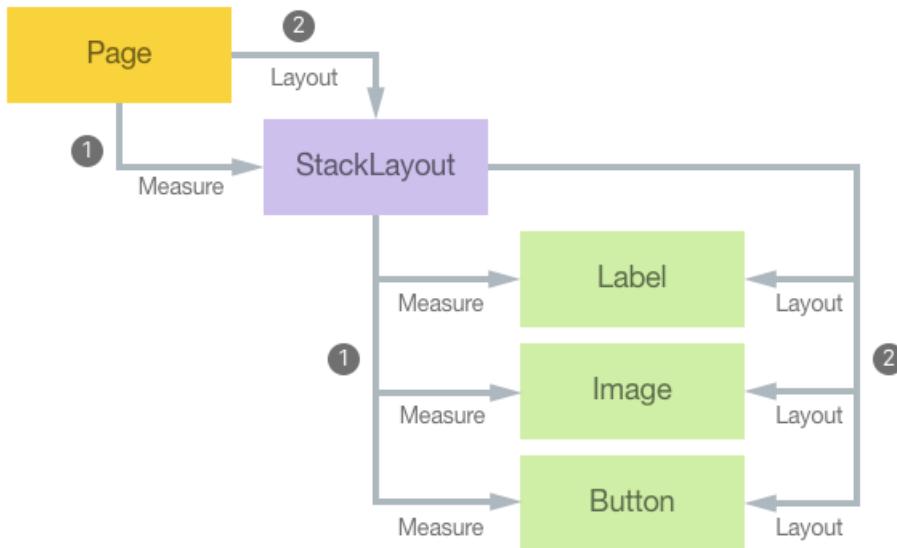
Layout

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.

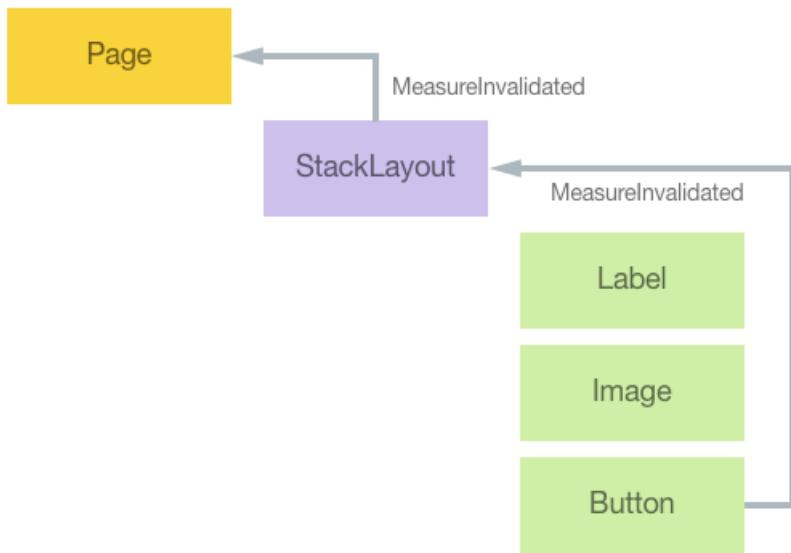
Invalidation

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 [Calculate and Cache Layout Data](#).

Create 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 [Create 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 [Add 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 [Override 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 [Override 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.

5. [optional] Override the `InvalidateLayout` method to be notified when children are added to or removed from the layout. For more information, see [Override the InvalidateLayout Method](#).
6. [optional] Override the `OnChildMeasureInvalidated` method to be notified when one of the layout's children changes size. For more information, see [Override 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 [Calculate and Cache 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 [Consume the WrapLayout](#).

Create 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>();
    ...
}
```

Calculate and Cache 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;
    }
}
```

```

        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.

Add 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 [Override the InvalidateLayout Method](#) and [Override the OnChildMeasureInvalidated Method](#).

Override 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 [Calculate and Cache Layout 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`.

Override 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 [Calculate and Cache Layout Data](#).

Override 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.

Override 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.

Consume 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();

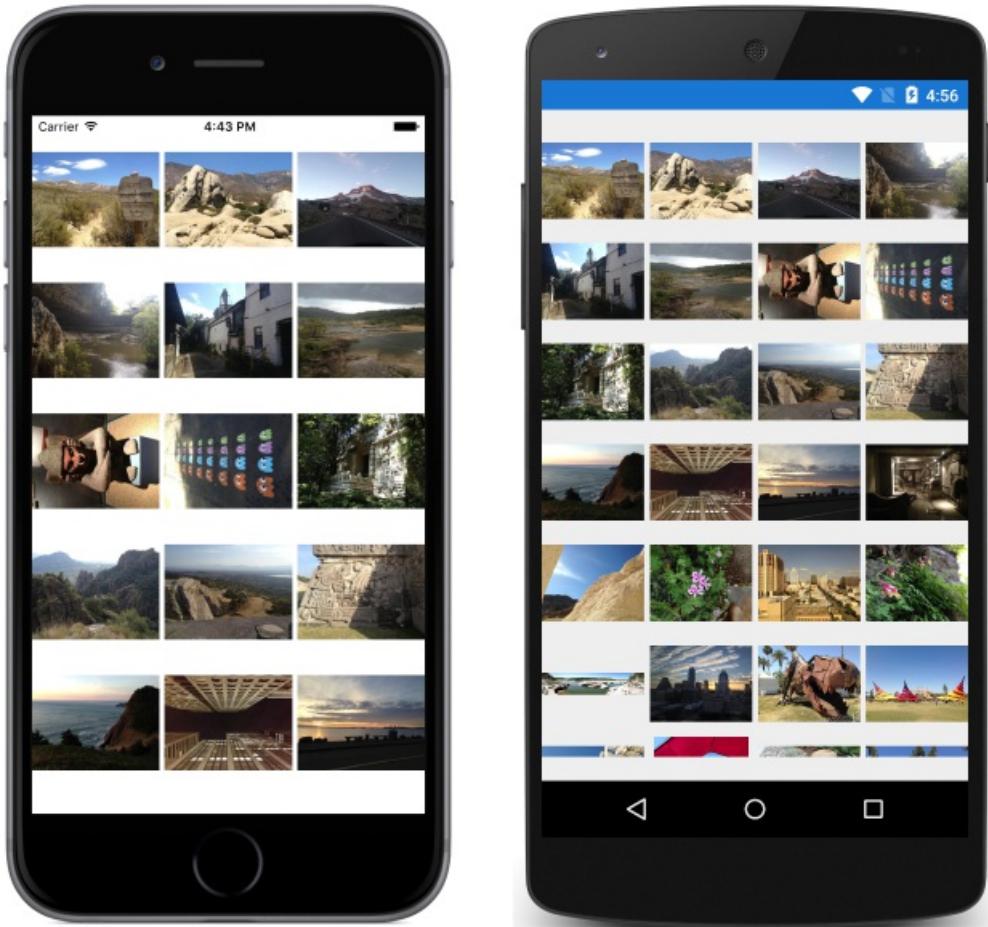
    var images = await GetImageListAsync();
    if (images != null)
    {
        foreach (var photo in images.Photos)
        {
            var image = new Image
            {
                Source = ImageSource.FromUri(new Uri(photo))
            };
            wrapLayout.Children.Add(image);
        }
    }
}

async Task<ImageList> GetImageListAsync()
{
    try
    {
        string requestUri = "https://raw.githubusercontent.com/xamarin/docs-archive/master/Images/stock/small/stock.json";
        string result = await _client.GetStringAsync(requestUri);
        return JsonConvert.DeserializeObject<ImageList>(result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine($"\\tERROR: {ex.Message}");
    }

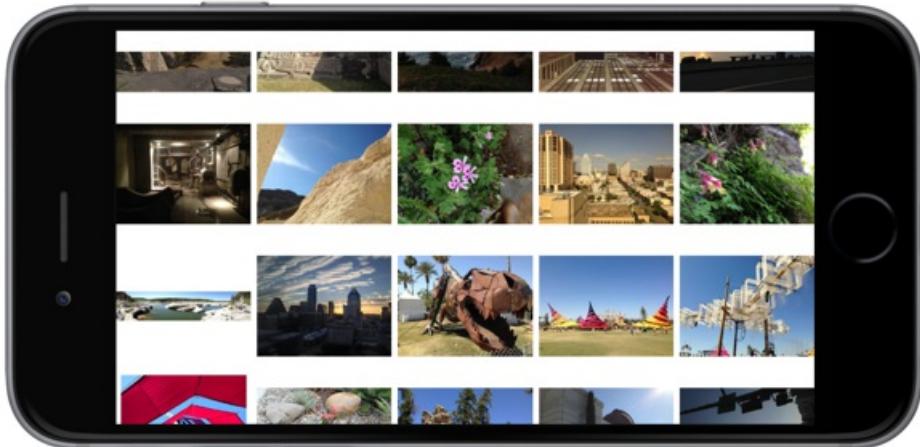
    return null;
}

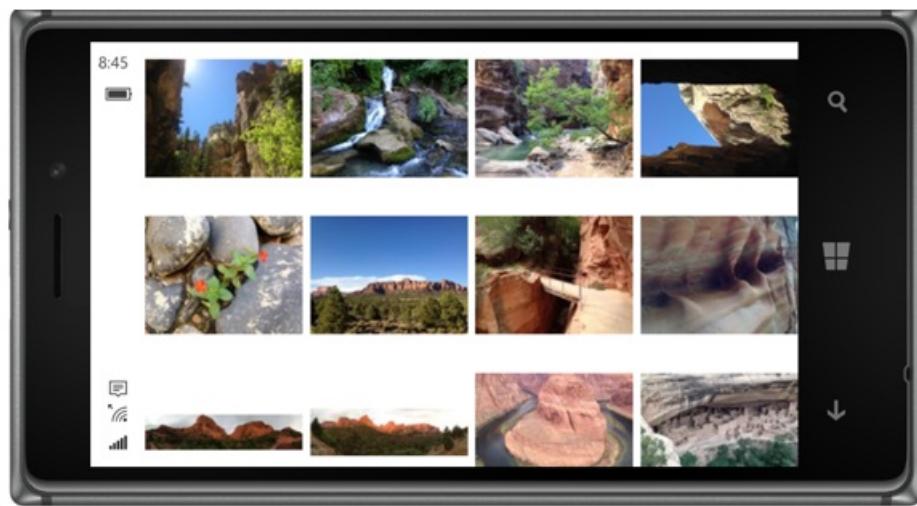
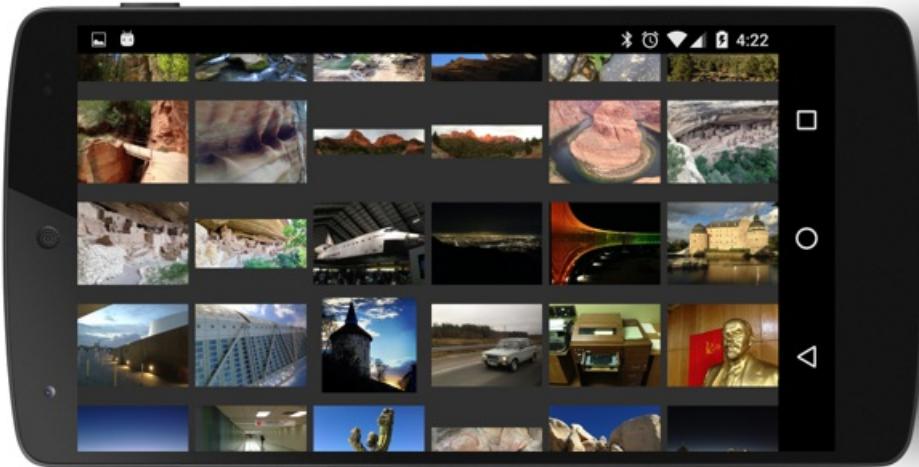
```

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.

Related links

- [WrapLayout \(sample\)](#)
- [Custom Layouts](#)
- [Creating Custom Layouts in Xamarin.Forms \(video\)](#)
- [Layout<T>](#)
- [Layout](#)
- [VisualElement](#)

Device Orientation

8/4/2022 • 9 minutes to read • [Edit Online](#)

 [Download the sample](#)

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

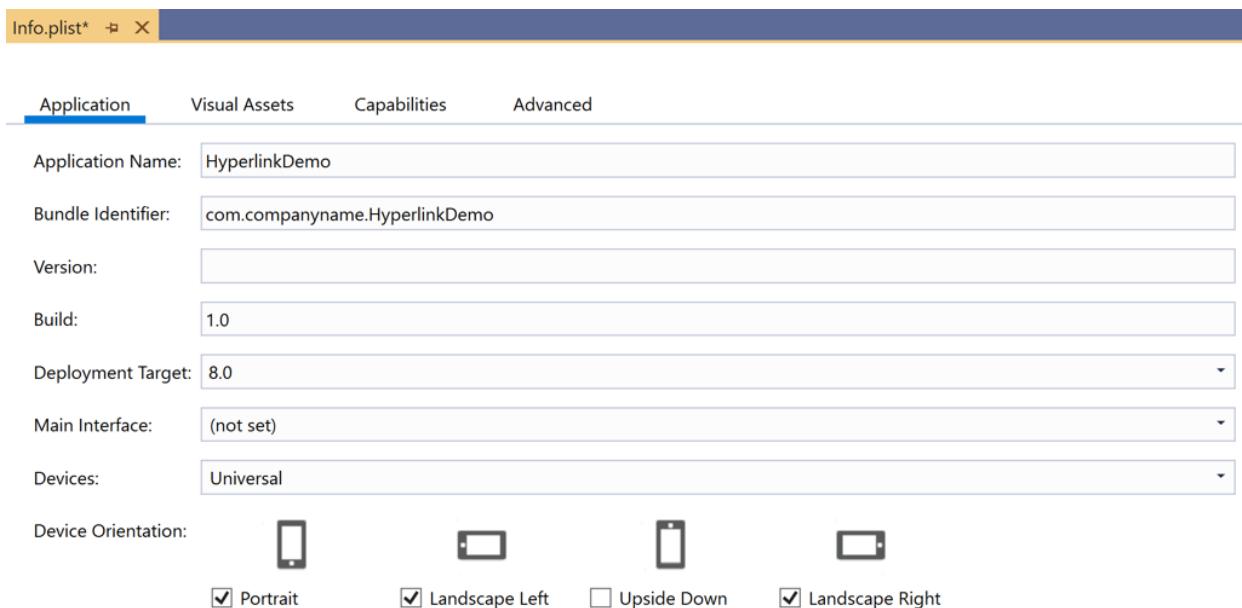
When using Xamarin.Forms, the supported method of controlling device orientation is to use the settings for each individual project.

iOS

On iOS, device orientation is configured for applications using the `Info.plist` file. 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:



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, [Xamarin.Essentials](#) contains a `[DeviceDisplay]` class that provides notifications of orientation changes.

To detect orientations without Xamarin.Essentials, monitor the `SizeChanged` event of the `Page`, which 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](#).

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
}
```

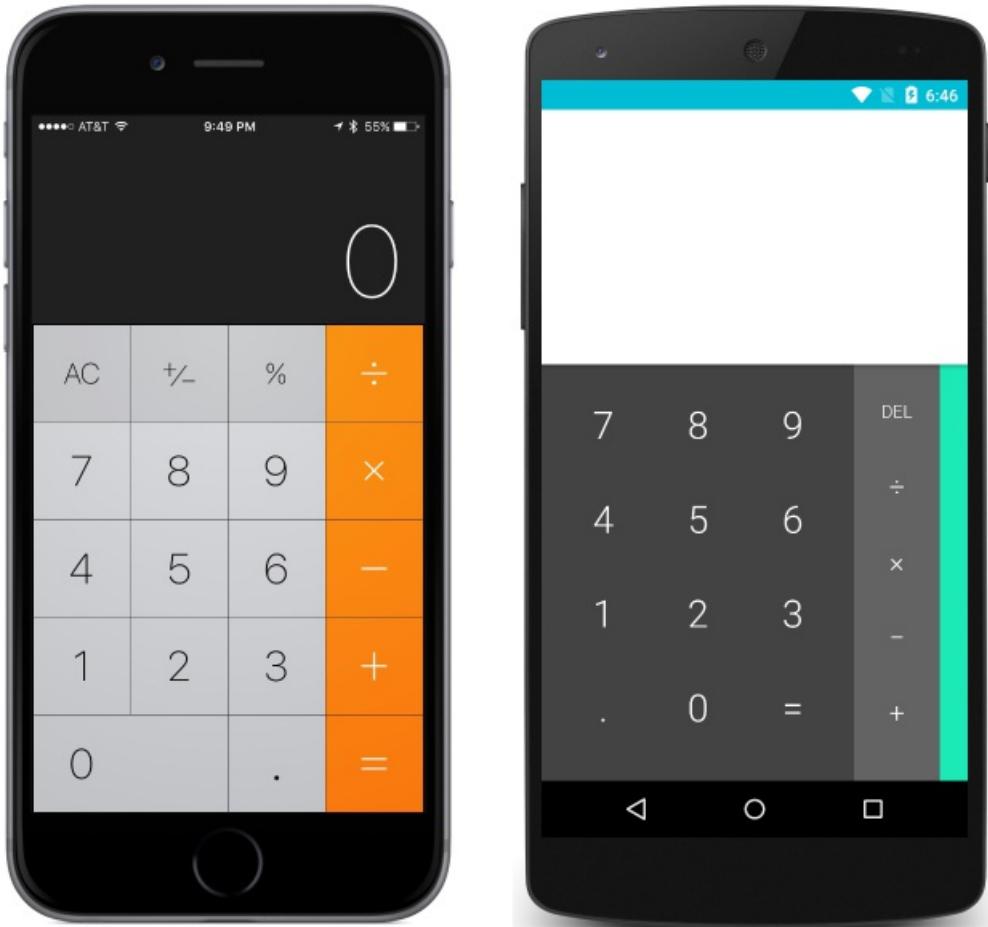
Failure to take that step will result in a non-functioning page.

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:

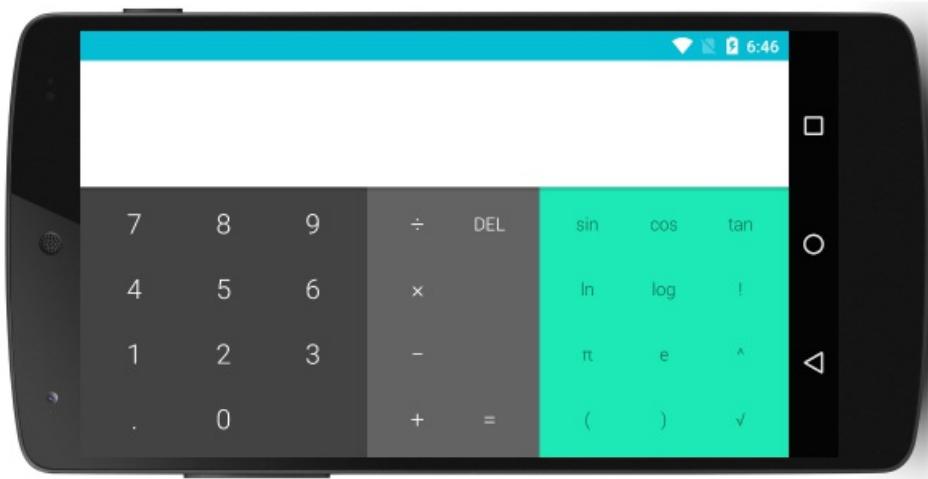
```
private double width = 0;
private double height = 0;

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.

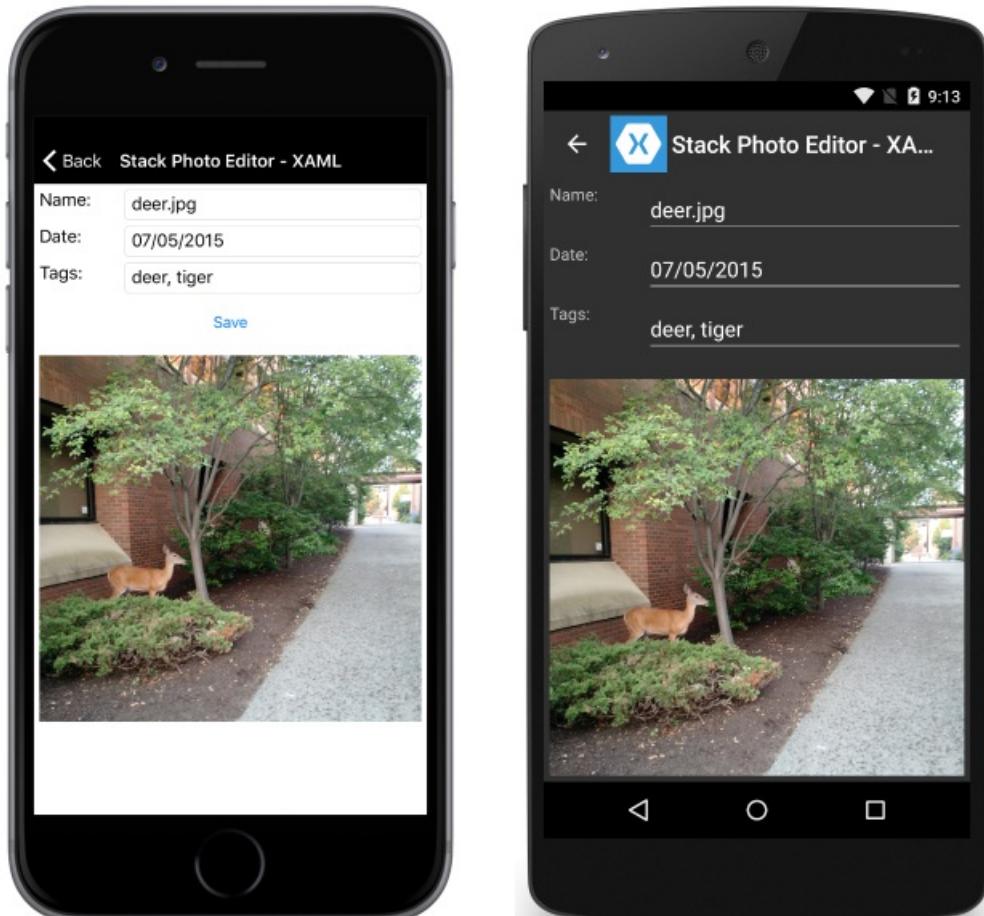
The above rules also apply when implementing interfaces for multiple screen sizes and are generally considered best-practice. The rest of this guide will explain specific examples of responsive layouts using each of the primary layouts in Xamarin.Forms.

NOTE

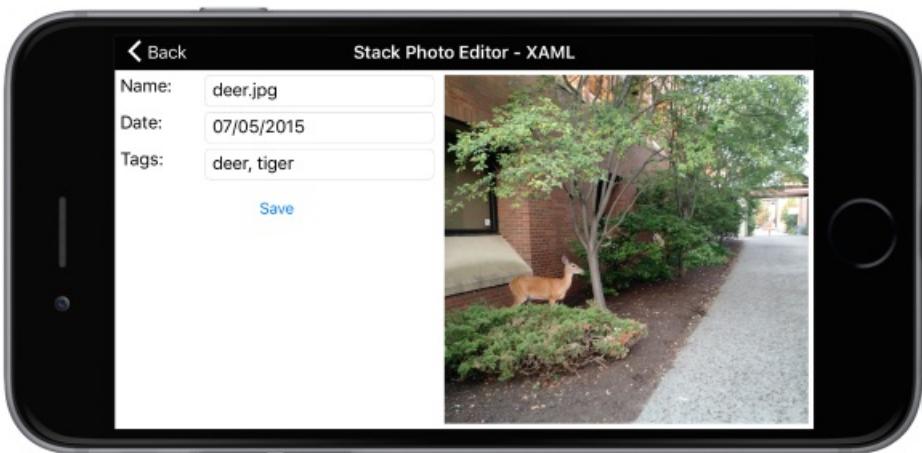
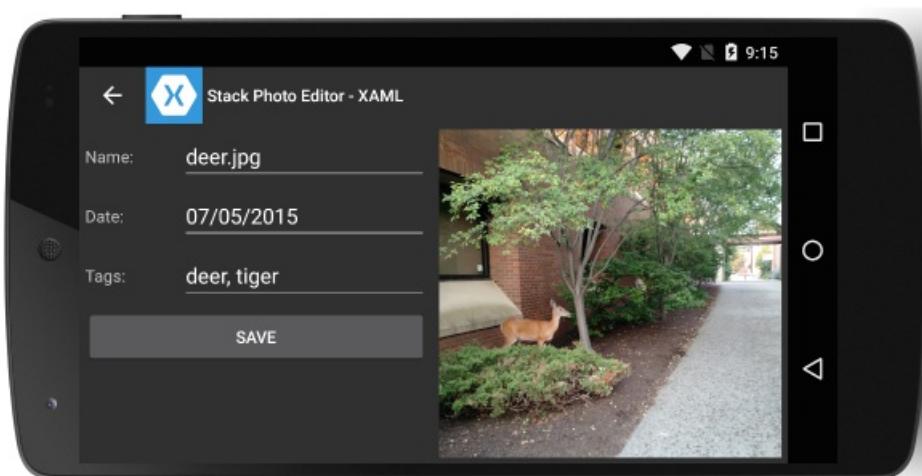
For clarity, the following sections demonstrate how to implement responsive layouts using just one type of `Layout` at a time. In practice, it is often simpler to mix `Layout`s to achieve a desired layout using the simpler or most intuitive `Layout` for each component.

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"> 
            <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>
                <Image Source="deer.jpg" />
            </ScrollView>
        </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;
        }
    }
}

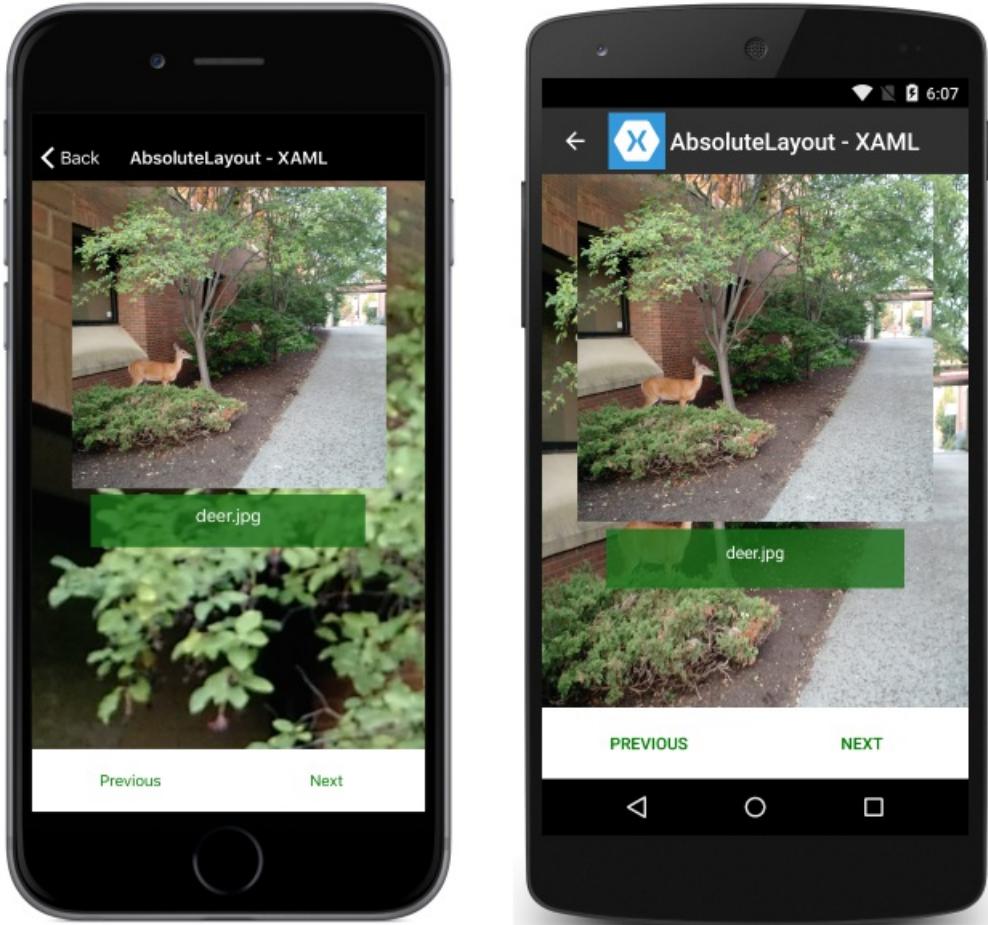
```

Note the following:

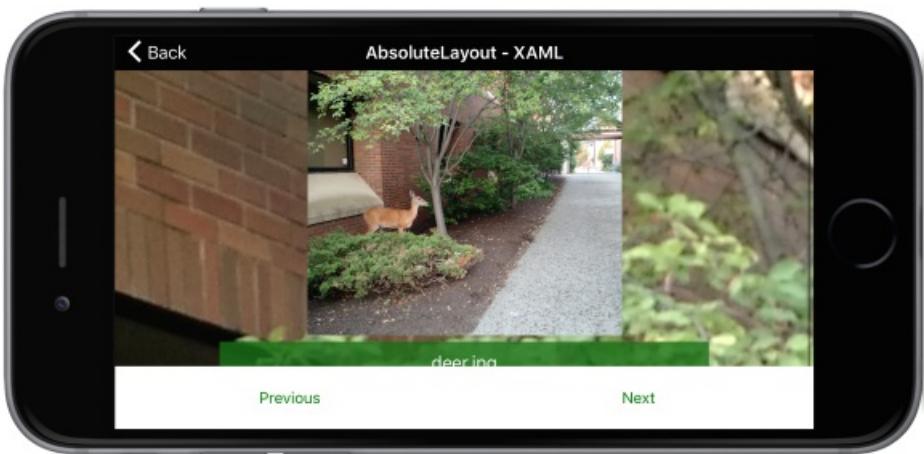
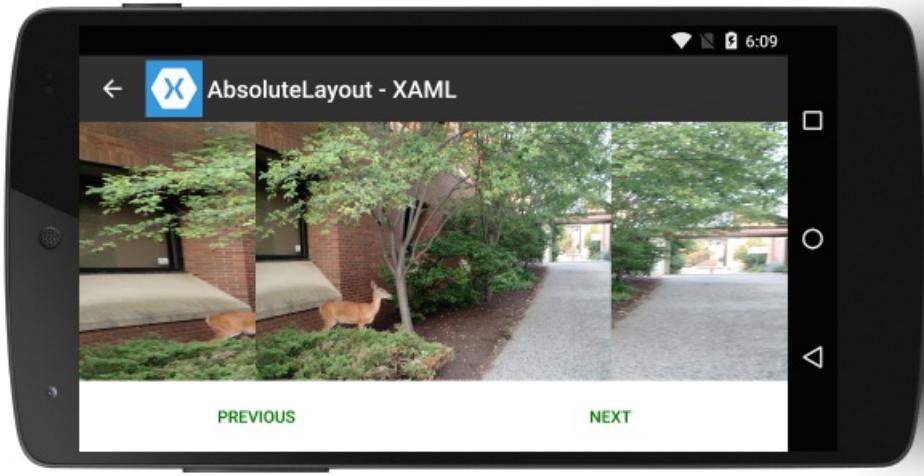
- `outerStack` is adjusted to present the image and controls as a horizontal or vertical stack depending on orientation, to best take advantage of the available space.

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" BackgroundImageSource="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>

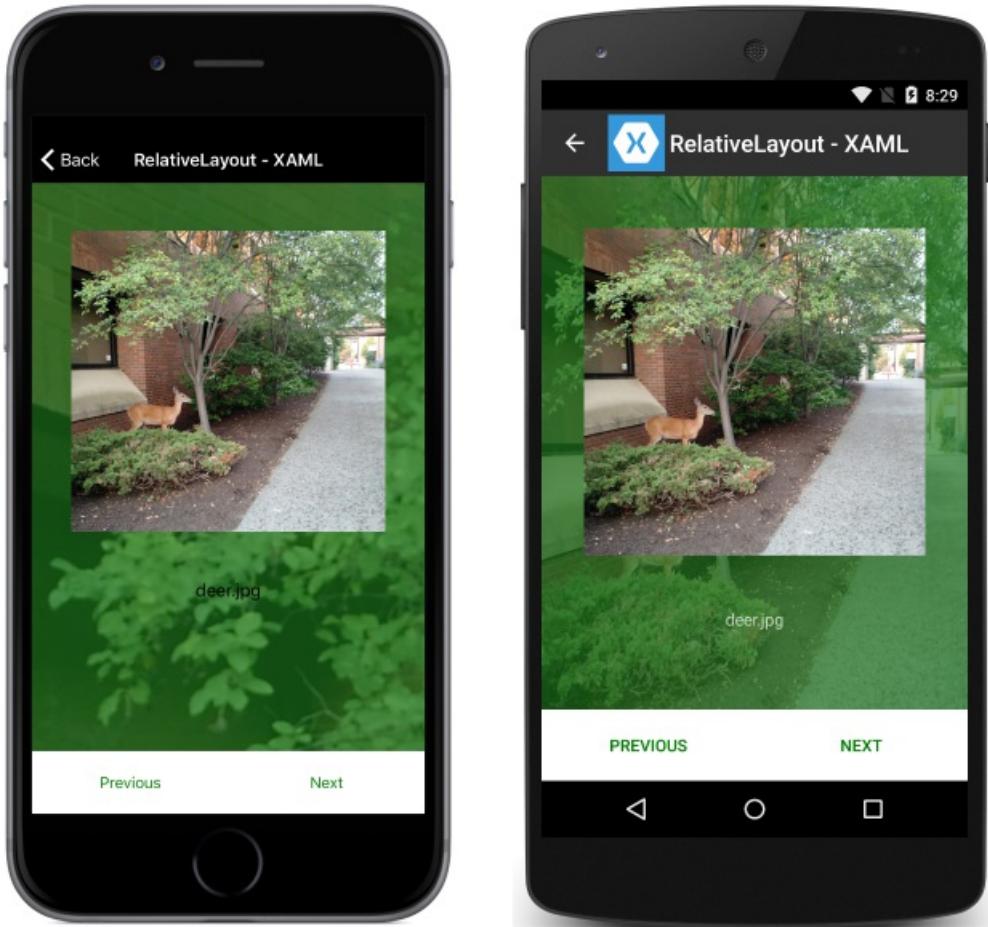
```

Note the following:

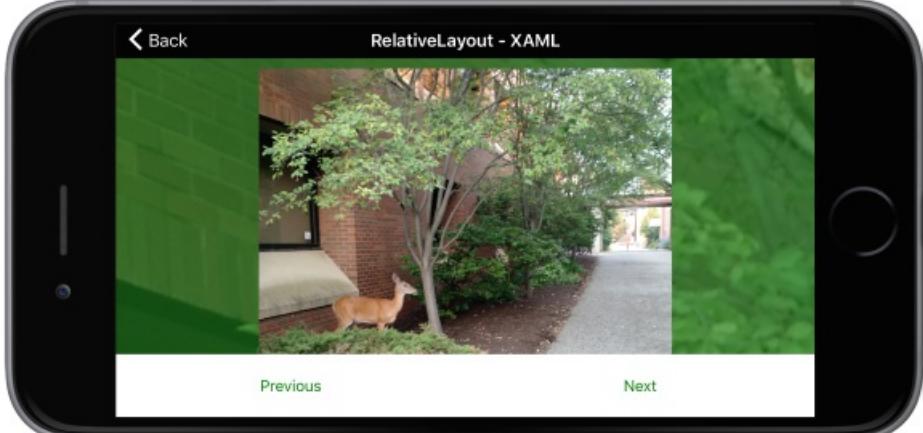
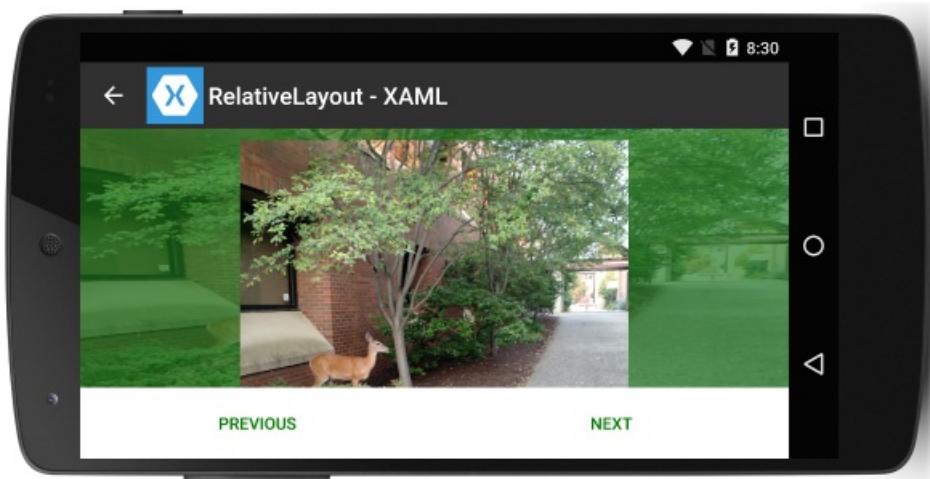
- Because of the way the page has been laid out, there is no need for procedural code to introduce responsiveness.
- The `ScrollView` is being used to allow the label to be visible even when the height of the screen is less than the sum of the fixed heights of the buttons and the image.

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"
    BackgroundImageSource="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}" />
                </Button>
            </RelativeLayout>
        </ContentPage.Content>
    </ContentPage>
```

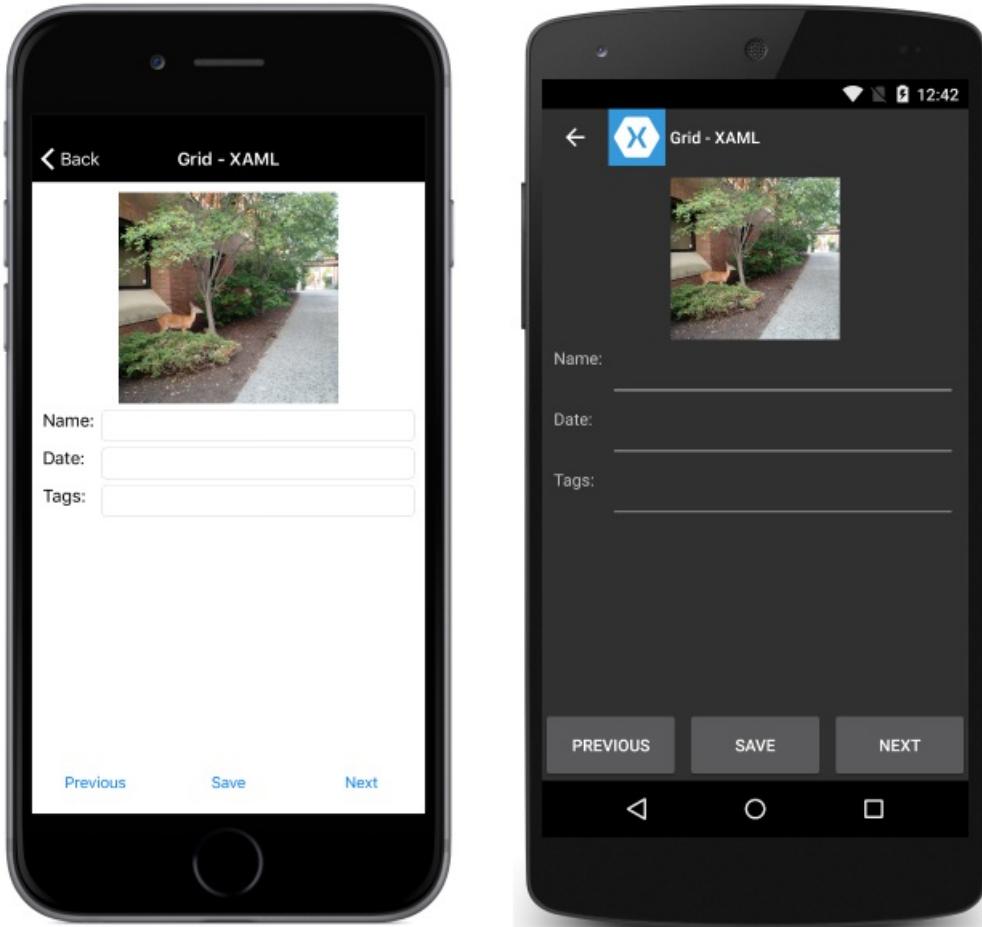
```
Type=RelativeToParent,Property=Width,Factor=.5}"  
/>>  
</RelativeLayout>  
</ContentPage.Content>  
</ContentPage>
```

Note the following:

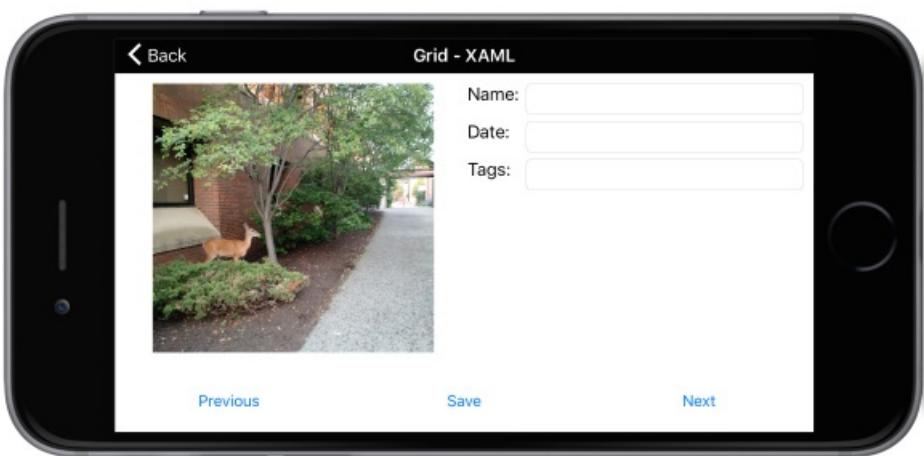
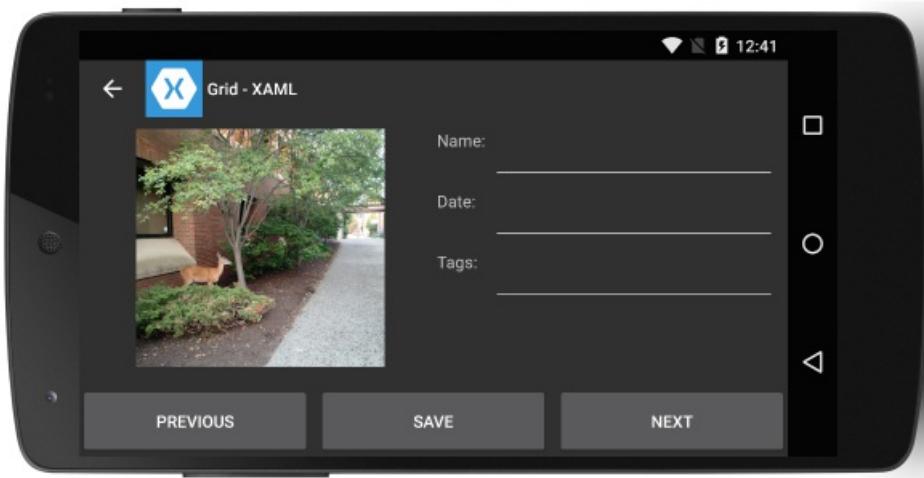
- Because of the way the page has been laid out, there is no need for procedural code to introduce responsiveness.
- The `ScrollView` is being used to allow the label to be visible even when the height of the screen is less than the sum of the fixed heights of the buttons and the image.

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.RowDefinitions.Clear();
        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 Options in Xamarin.Forms

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)

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`.

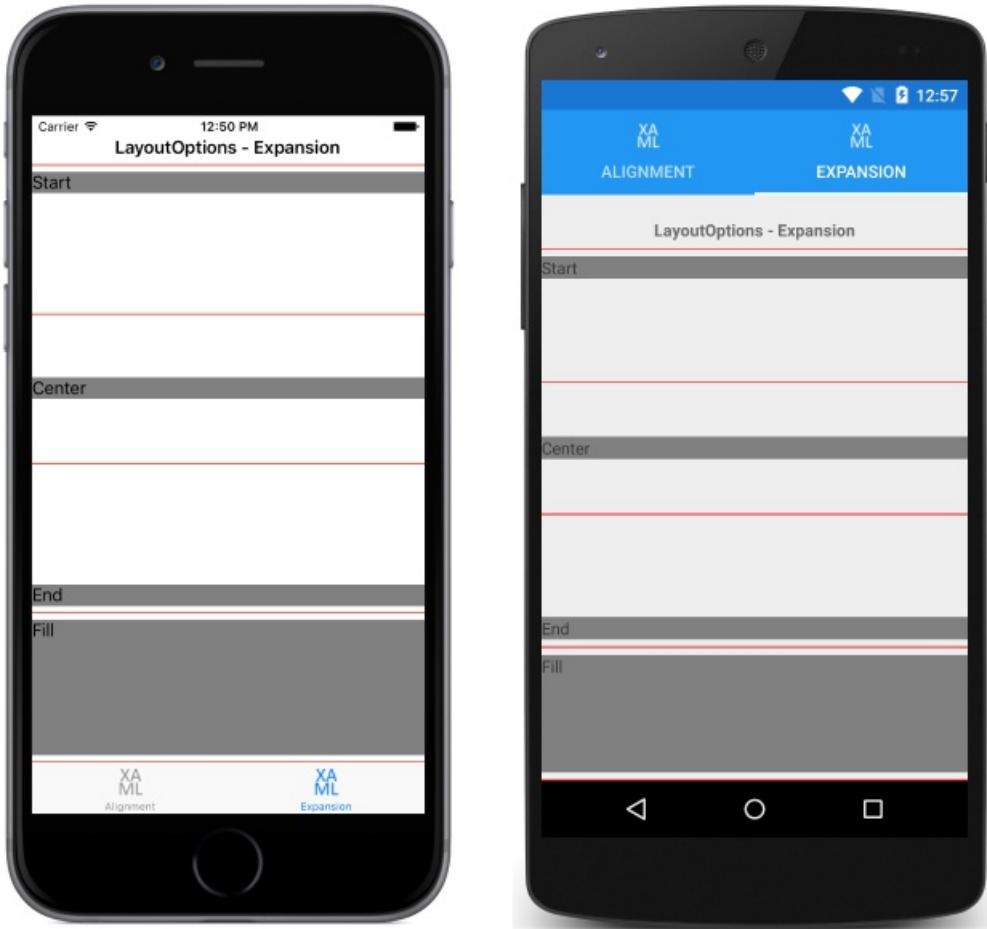
The following XAML code example demonstrates a vertically oriented `StackLayout` where each child `Label` sets its `VerticalOptions` property to one of the four expansion fields from the `LayoutOptions` structure:

```
<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](#)

Layout Compression

8/4/2022 • 4 minutes to read • [Edit Online](#)

 [Download the sample](#)

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

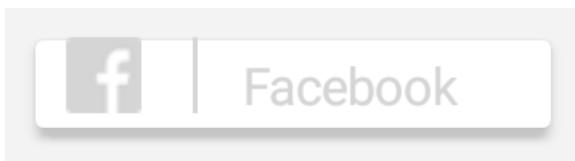
Xamarin.Forms performs layout using two series of recursive method calls:

- 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.

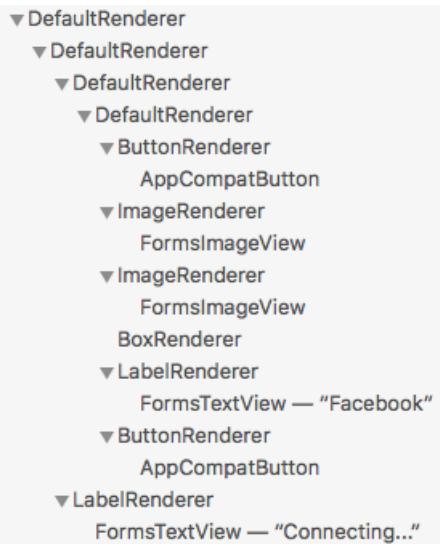
For example, consider the following button from the sample application for logging into Facebook:



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 the Live Visual Tree. On Android, the nested view hierarchy contains 17 views:



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);
```

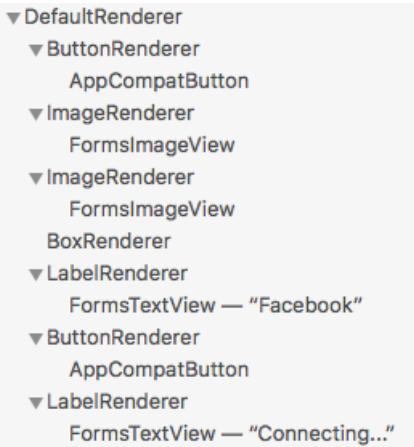
IMPORTANT

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:



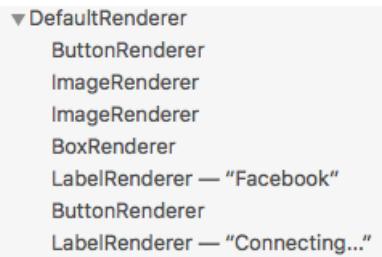
```
▼ DefaultRenderer
  ▼ ButtonRenderer
    AppCompatButton
  ▼ ImageRenderer
    FormsImageview
  ▼ ImageRenderer
    FormsImageview
  BoxRenderer
  ▼ LabelRenderer
    FormsTextView — "Facebook"
  ▼ ButtonRenderer
    AppCompatButton
  ▼ LabelRenderer
    FormsTextView — "Connecting..."
```

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 \(sample\)](#)

Margin and Padding

8/4/2022 • 2 minutes to read • [Edit Online](#)

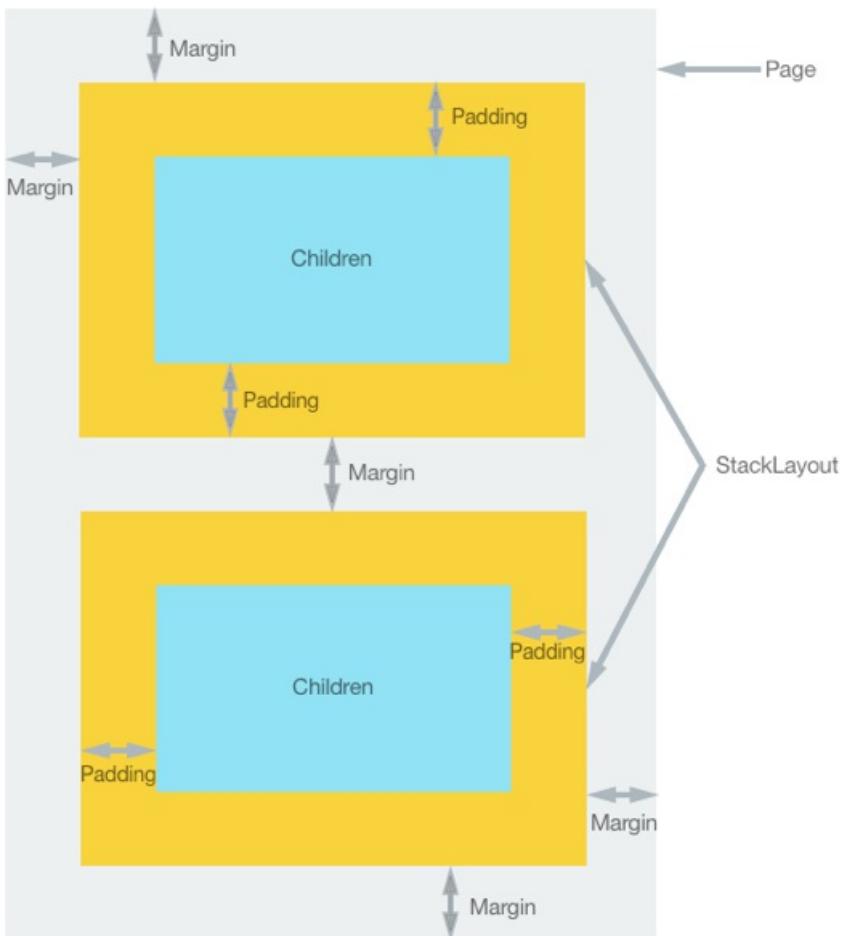
The `Margin` and `Padding` properties control layout behavior when an element is rendered in the user interface. This article demonstrates the difference between the two properties, and how to set them.

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](#)

Layout for Tablet and Desktop apps

8/4/2022 • 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

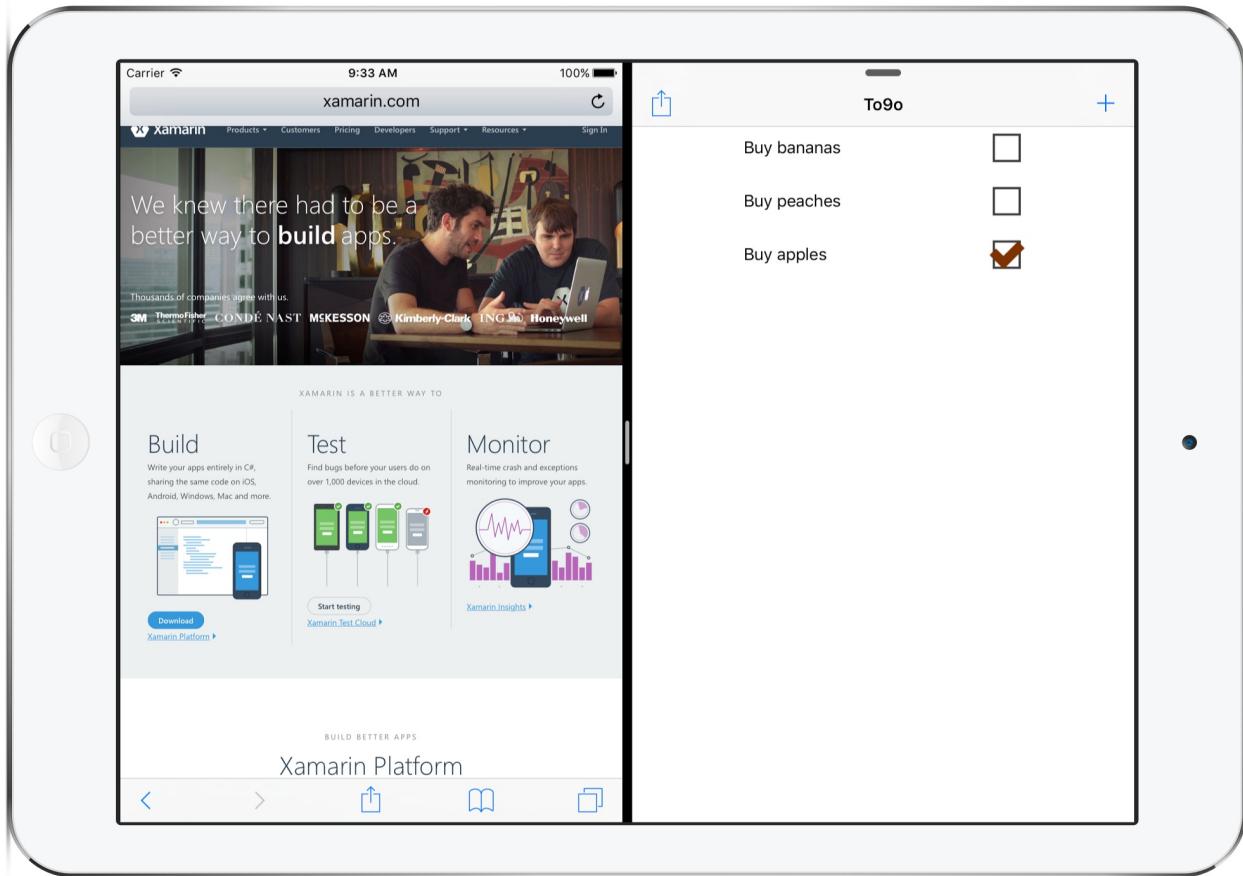
Larger screen devices are available for all of the platforms supported by Xamarin.Forms.

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

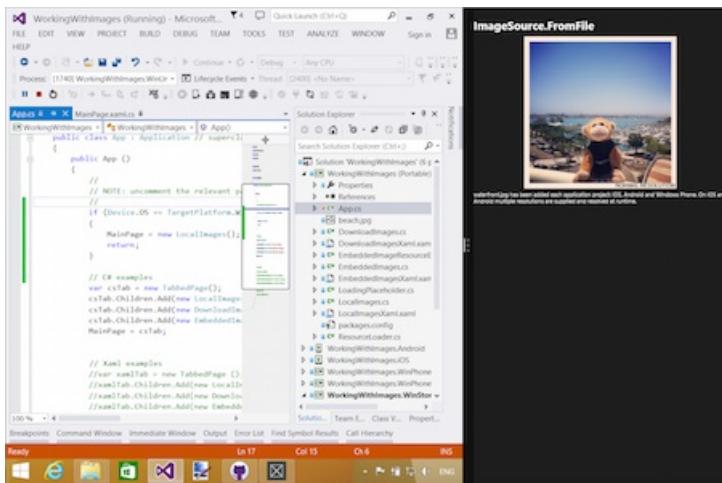
The Android ecosystem has a myriad of supported screen sizes, from small phones up to large tablets. Xamarin.Forms can support all screen sizes, but as with the other platforms you might want to adjust your user interface for larger devices.

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.



Optimize 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.

Leverage FlyoutPage

The `FlyoutPage` 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 `FlyoutPage`).

Related Links

- [Xamarin Blog](#)
- [MyShoppe sample](#)

Android Platform Features

8/4/2022 • 2 minutes to read • [Edit Online](#)

Developing Xamarin.Forms applications for Android requires Visual Studio. The [supported platforms page](#) contains more information about the pre-requisites.

Platform-specifics

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 on Android:

- Controlling the Z-order of visual elements to determine drawing order. For more information, see [VisualElement Elevation on Android](#).
- Disabling legacy color mode on a supported [VisualElement](#). For more information, see [VisualElement Legacy Color Mode on Android](#).

The following platform-specific functionality is provided for Xamarin.Forms views on Android:

- Using the default padding and shadow values of Android buttons. For more information, see [Button Padding and Shadows on Android](#).
- Setting the input method editor options for the soft keyboard for an [Entry](#). For more information, see [Entry Input Method Editor Options on Android](#).
- Enabling a drop shadow on a [ImageButton](#). For more information, see [ImageButton Drop Shadows on Android](#).
- Enabling fast scrolling in a [ListView](#). For more information, see [ListView Fast Scrolling on Android](#).
- Controlling the transition that's used when opening a [SwipeView](#). For more information, see [SwipeView Swipe Transition Mode](#).
- Controlling whether a [WebView](#) can display mixed content. For more information, see [WebView Mixed Content on Android](#).
- Enabling zoom on a [WebView](#). For more information, see [WebView Zoom on Android](#).

The following platform-specific functionality is provided for Xamarin.Forms cells on Android:

- Enabling [ViewCell](#) context actions legacy mode, so that the context actions menu is not updated when the selected item in a [ListView](#) changes. For more information, see [ViewCell Context Actions on Android](#).

The following platform-specific functionality is provided for Xamarin.Forms pages on Android:

- Setting the height of the navigation bar on a [NavigationPage](#). For more information, see [NavigationPage Bar Height on Android](#).
- Disabling transition animations when navigating through pages in a [TabbedPage](#). For more information, see [TabbedPage Page Transition Animations on Android](#).
- Enabling swiping between pages in a [TabbedPage](#). For more information, see [TabbedPage Page Swiping on Android](#).
- Setting the toolbar placement and color on a [TabbedPage](#). For more information, see [TabbedPage Toolbar Placement and Color on Android](#).

The following platform-specific functionality is provided for the Xamarin.Forms `Application` class on Android:

- Setting the operating mode of a soft keyboard. For more information, see [Soft Keyboard Input Mode on Android](#).
- Disabling the `Disappearing` and `Appearing` page lifecycle events on pause and resume respectively, for applications that use AppCompat. For more information, see [Page Lifecycle Events on Android](#).

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.

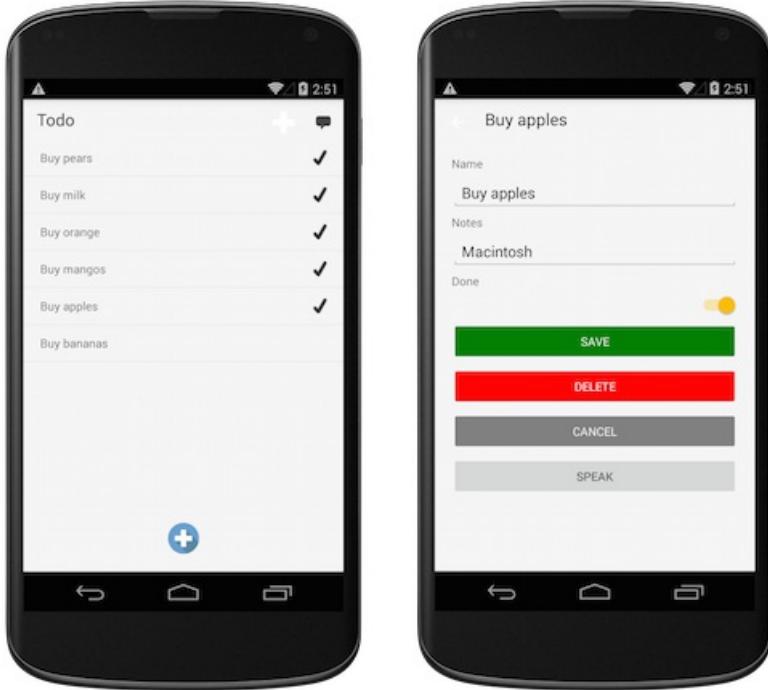
Material design via AppCompat

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.

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.

AndroidX Migration

AndroidX replaces the Android Support Library. To learn about AndroidX and how to migrate a Xamarin.Forms app to use AndroidX libraries, see [AndroidX migration in Xamarin.Forms](#).

AndroidX migration in Xamarin.Forms

8/4/2022 • 3 minutes to read • [Edit Online](#)

AndroidX replaces the Android Support Library. This article explains why AndroidX exists, how it impacts Xamarin.Forms, and how to migrate your application to use the AndroidX libraries.

IMPORTANT

If you are migrating an app to Xamarin.Forms 5.0, see [How do I migrate my app to Xamarin.Forms 5.0?](#).

History of AndroidX

The Android Support Library was created to provide newer features on older versions of Android. It is a compatibility layer that allows developers to use functionality that may not exist on all versions of the Android operating system and have graceful fallbacks for older versions. The Support Library also includes convenience and helper classes, debugging and utility tools, and sophisticated classes that depend on other Support Library classes to function.

While the Support Library was originally a single binary, it has grown and evolved into a suite of libraries, which are almost essential for modern app development. These are some commonly used features from the Support Library:

- The `Fragment` support class.
- The `RecyclerView`, used for managing long lists.
- Multidex support for apps with over 65,536 methods.
- The `ActivityCompat` class.

AndroidX is a replacement for the Support Library, which is no longer maintained - all new library development will occur in the AndroidX library. AndroidX is a redesigned library that uses semantic versioning, clearer package names, and better support for common application architecture patterns. AndroidX version 1.0.0 is the binary equivalent to Support Library version 28.0.0. For a complete list of class mappings from Support Library to AndroidX, see [Support Library class mappings](#) on developer.android.com.

Google created a migration process called the Jetifier with AndroidX. The Jetifier inspects the jar bytecode during the build process and remaps Support Library references, both in app code and in dependencies, to their AndroidX equivalent.

In a Xamarin.Forms app, just as in an Android Java app, the jar dependencies must be migrated to AndroidX. However, the Xamarin bindings must also be migrated to point to the correct, underlying jar files. Xamarin.Forms added support for automatic AndroidX migration in version 4.5.

For more information about AndroidX, see [AndroidX overview](#) on developer.android.com.

Automatic migration in Xamarin.Forms

To automatically migrate to AndroidX, a Xamarin.Forms Android platform project must:

- Target Android API version 29 or greater.
- Use Xamarin.Forms version 4.5 or greater.
- Have direct or transitive dependencies on Android support libraries.

Once you have confirmed these settings in your project, build the Android app in Visual Studio 2019. During the build process, the Intermediate Language (IL) is inspected and Support Library dependencies and bindings are swapped with AndroidX dependencies. If your application has all of the AndroidX dependencies required to build, you will notice no differences in the build process.

IMPORTANT

Manual migration to AndroidX will result in the fastest build process for your app, and is the recommended approach for AndroidX migration. This involves replacing support library dependencies with AndroidX dependencies, and updating your code to consume AndroidX types. For more information, see [Use AndroidX types](#).

If AndroidX dependencies are detected that are not part of the project, a build error is reported that indicates which AndroidX packages are missing. An example build error is shown below:

```
Could not find 37 AndroidX assemblies, make sure to install the following NuGet packages:  
- Xamarin.AndroidX.Lifecycle.LiveData  
- Xamarin.AndroidX.Browser  
- Xamarin.Google.Android.Material  
- Xamarin.AndroidX.Legacy.Supportv4  
You can also copy and paste the following snippet into your .csproj file:  
<PackageReference Include="Xamarin.AndroidX.Lifecycle.LiveData" Version="2.1.0-rc1" />  
<PackageReference Include="Xamarin.AndroidX.Browser" Version="1.0.0-rc1" />  
<PackageReference Include="Xamarin.Google.Android.Material" Version="1.0.0-rc1" />  
<PackageReference Include="Xamarin.AndroidX.Legacy.Support.V4" Version="1.0.0-rc1" />
```

The missing NuGet packages can either be installed via the NuGet Package Manager in Visual Studio, or installed by editing your Android .csproj file to include the `PackageReference` XML items listed in the error.

Once the missing packages are resolved, rebuilding the project loads the missing packages and your project is compiled using AndroidX dependencies instead of Support Library dependencies.

NOTE

If your project, and project dependencies, do not reference Android Support Libraries, the migration process does nothing and is not executed.

Related links

- [How do I migrate my app to Xamarin.Forms 5.0?](#)
- [Android Support Library overview](#) on developer.android.com
- [AndroidX overview](#) on developer.android.com
- [AndroidX class mappings](#)
- [AndroidX assemblies](#)

Button Padding and Shadows on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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:

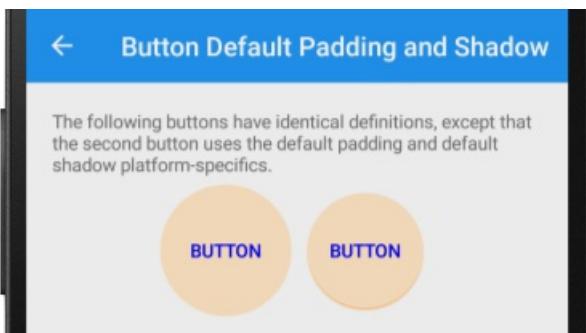
```
<ContentPage ...>
    xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            ...
            <Button ...>
                android:Button.UseDefaultPadding="true"
                android:Button.UseDefaultShadow="true" />
        </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.

Related links

- [PlatformSpecifics \(sample\)](#)

- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

Entry Input Method Editor Options on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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:

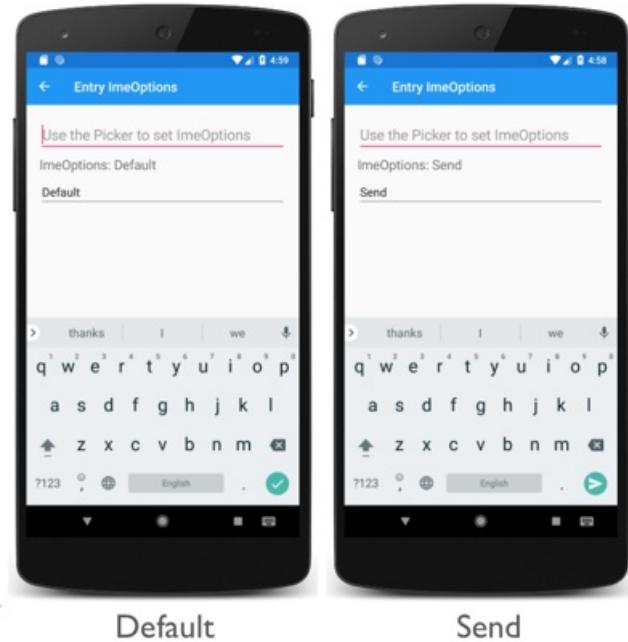
```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
entry.On<Android>().SetImeOptions(ImeFlags.Send);
```

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:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

ImageButton Drop Shadows on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android platform-specific is used to enable a drop shadow on a `ImageButton`. It's consumed in XAML by setting the `ImageButton.IsShadowEnabled` 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`

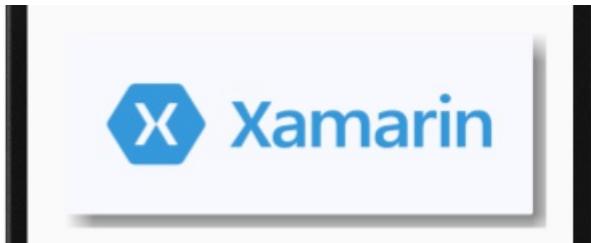
namespace, is used to control whether a drop shadow is enabled on the `ImageButton`. In addition, the following methods can be invoked to control the drop shadow:

- `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`:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

ListView Fast Scrolling on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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:

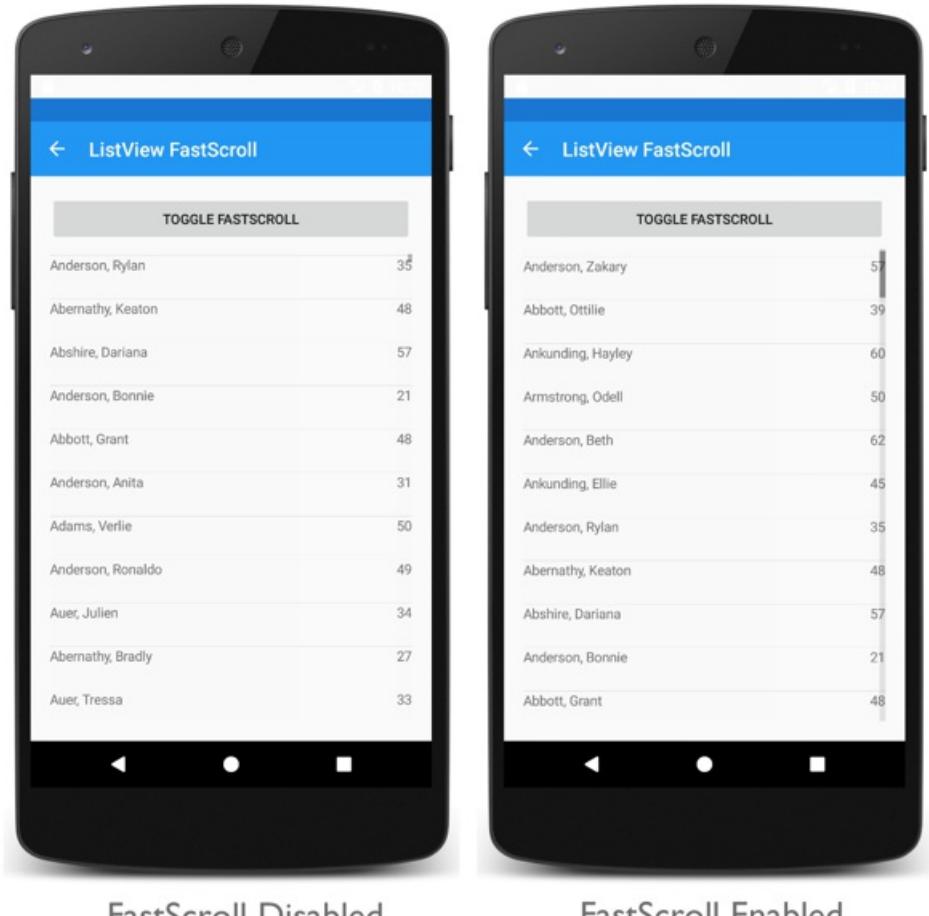
```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

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:



FastScroll Disabled

FastScroll Enabled

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

NavigationPage Bar Height on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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:

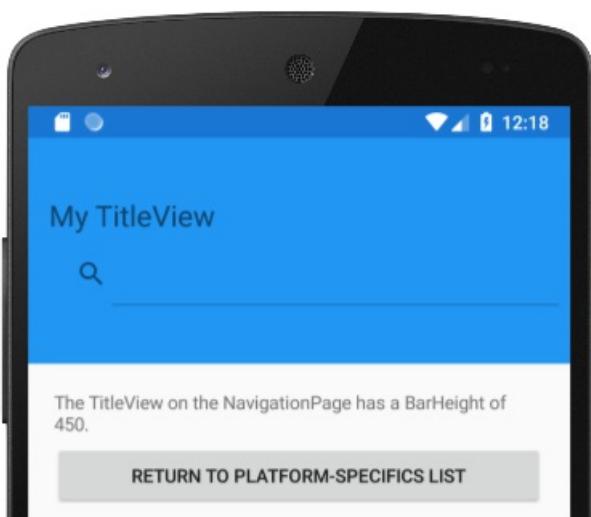
```
<NavigationPage ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat;assembly=Xamarin.Forms.Core"  
    android:NavigationBar.BarHeight="450">  
    ...  
</NavigationPage>
```

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:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

Page Lifecycle Events on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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>
```

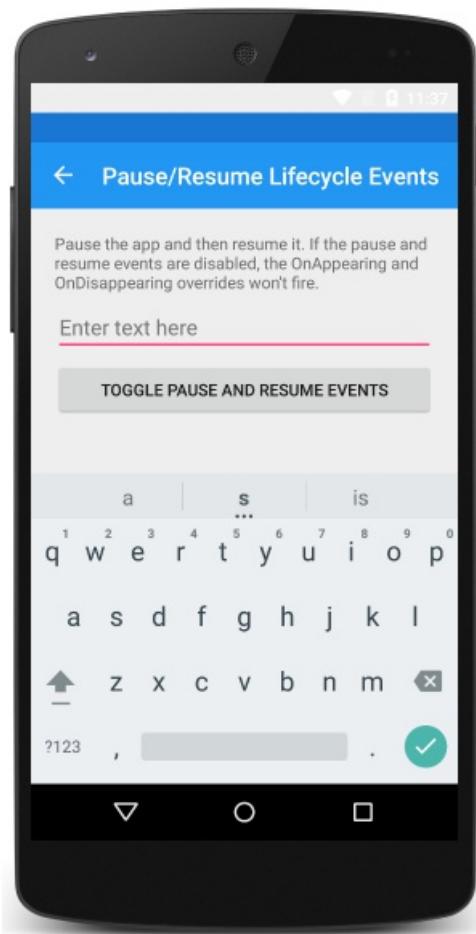
Alternatively, it can be consumed from C# using the fluent API:

```
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:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

Soft Keyboard Input Mode on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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:

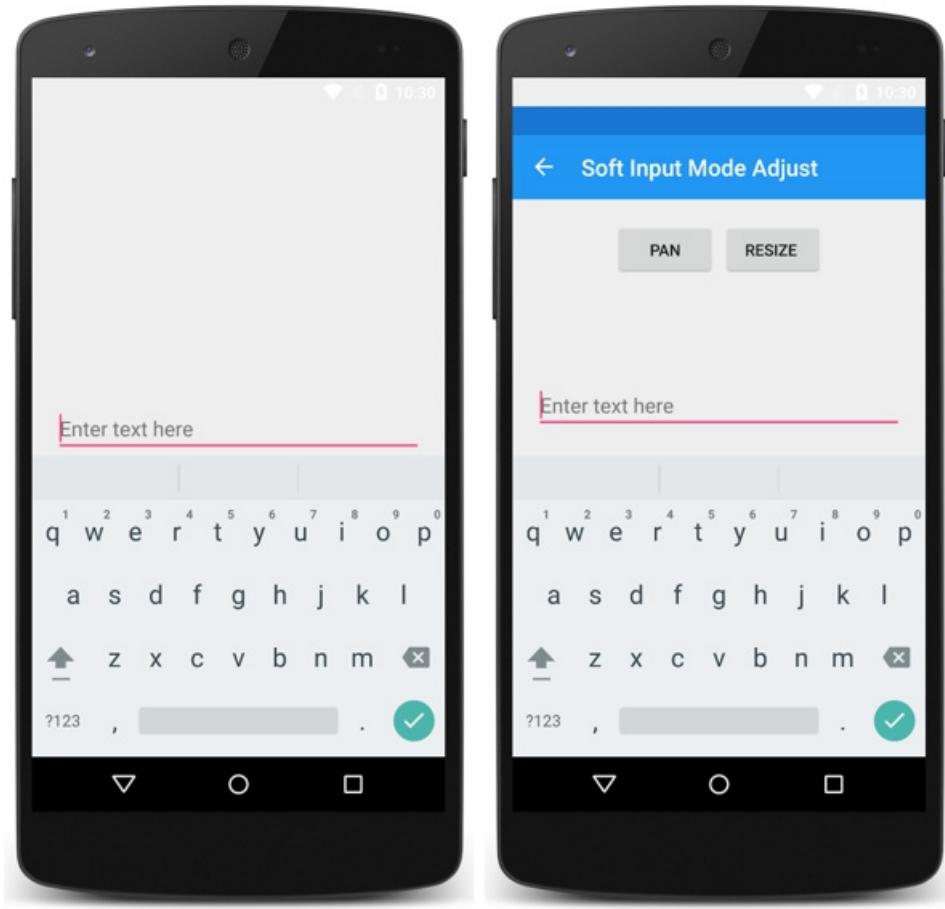
```
<Application ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"  
        android:Application.WindowSoftInputModeAdjust="Resize">  
    ...  
</Application>
```

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

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

SwipeView Swipe Transition Mode on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android platform-specific controls the transition that's used when opening a `SwipeView`. It's consumed in XAML by setting the `SwipeView.SwipeTransitionMode` bindable property to a value of the `SwipeTransitionMode` enumeration:

```
<ContentPage ...>
    xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core" >
    <StackLayout>
        <SwipeView android:SwipeView.SwipeTransitionMode="Drag">
            <SwipeView.LeftItems>
                <SwipeItems>
                    <SwipeItem Text="Delete"
                               IconImageSource="delete.png"
                               BackgroundColor="LightPink"
                               Invoked="OnDeleteSwipeItemInvoked" />
                </SwipeItems>
            </SwipeView.LeftItems>
            <!-- Content -->
        </SwipeView>
    </StackLayout>
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

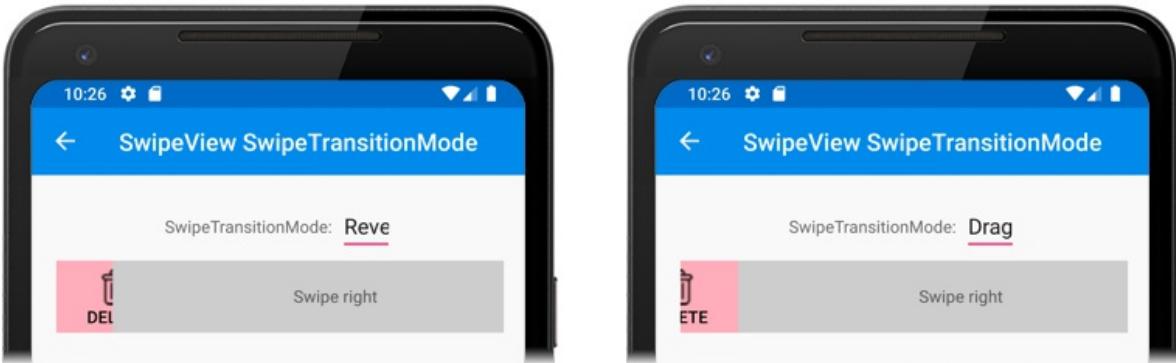
```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
SwipeView swipeView = new Xamarin.Forms.SwipeView();
swipeView.On<Android>().SetSwipeTransitionMode(SwipeTransitionMode.Drag);
// ...
```

The `SwipeView.On<Android>` method specifies that this platform-specific will only run on Android. The `SwipeView.SetSwipeTransitionMode` method, in the `Xamarin.Forms.PlatformConfiguration.iOSspecific` namespace, is used to control the transition that's used when opening a `SwipeView`. The `SwipeTransitionMode` enumeration provides two possible values:

- `Reveal` indicates that the swipe items will be revealed as the `SwipeView` content is swiped, and is the default value of the `SwipeView.SwipeTransitionMode` property.
- `Drag` indicates that the swipe items will be dragged into view as the `SwipeView` content is swiped.

In addition, the `SwipeView.GetSwipeTransitionMode` method can be used to return the `SwipeTransitionMode` that's applied to the `SwipeView`.

The result is that a specified `SwipeTransitionMode` value is applied to the `SwipeView`, which controls the transition that's used when opening the `SwipeView`:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)

TabPage Page Swiping on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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.IsSwipePagingEnabled` attached property to a `boolean` value:

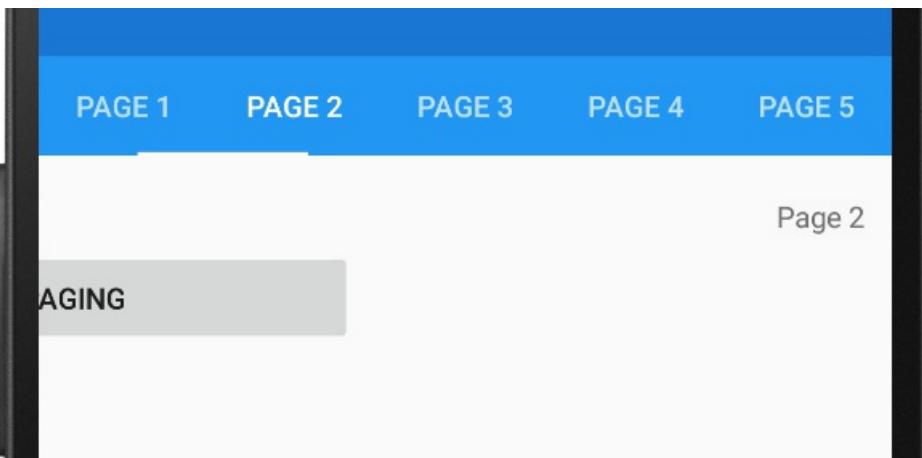
```
<TabPage ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"  
    android:TabbedPage.OffscreenPageLimit="2"  
    android:TabbedPage.IsSwipePagingEnabled="true">  
    ...  
</TabPage>
```

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:



Related links

- [PlatformSpecifics \(sample\)](#)

- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

TabPage Page Transition Animations on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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`:

```
<TabPage ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"  
        android:TabPage.IsSmoothScrollEnabled="false">  
    ...  
</TabPage>
```

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 `TabPage`. 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`.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

TabPage Toolbar Placement and Color on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

IMPORTANT

The platform-specifics that set the color of the toolbar on a `TabPage` are now obsolete and have been replaced by the `SelectedTabColor` and `UnselectedTabColor` properties. For more information, see [Create a TabbedPage](#).

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`:

```
<TabPage ...  
    xmlns:android="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"  
        android:TabbedPane.ToolbarPlacement="Bottom"  
        android:TabbedPane.BarItemColor="Black"  
        android:TabbedPane.BarSelectedItemColor="Red">  
    ...  
</TabPage>
```

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`:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

ViewCell Context Actions on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

By default from Xamarin.Forms 4.3, when a `ViewCell` in an Android application defines context actions for each item in a `ListView`, the context actions menu is updated when the selected item in the `ListView` changes.

However, in previous versions of Xamarin.Forms the context actions menu was not updated, and this behavior is referred to as the `ViewCell` legacy mode. This legacy mode can result in incorrect behavior if a `ListView` uses a `DataTemplateSelector` to set its `ItemTemplate` from `DataTemplate` objects that define different context actions.

This Android platform-specific enables the `ViewCell` context actions menu legacy mode, for backwards compatibility, so that the context actions menu is not updated when the selected item in a `ListView` changes. It's consumed in XAML by setting the `ViewCell.IsEnabled` bindable property to `true`:

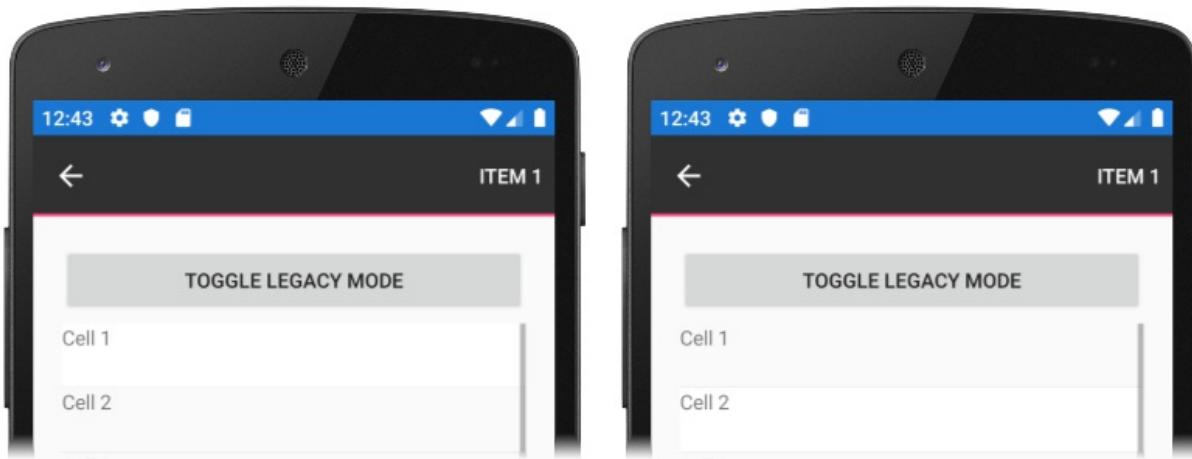
```
<ContentPage ...>
    xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout Margin="20">
            <ListView ItemsSource="{Binding Items}">
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <ViewCell android:ViewCell.IsEnabled="true">
                            <ViewCell.ContextActions>
                                <MenuItem Text="{Binding Item1Text}" />
                                <MenuItem Text="{Binding Item2Text}" />
                            </ViewCell.ContextActions>
                            <Label Text="{Binding Text}" />
                        </ViewCell>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
viewCell.On<Android>().SetIsContextActionsLegacyModeEnabled(true);
```

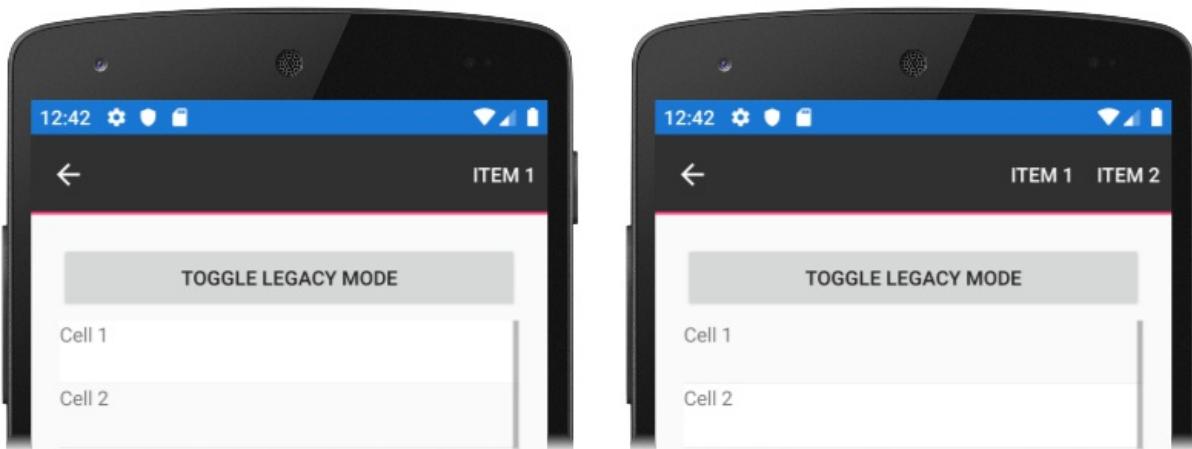
The `ViewCell.On<Android>` method specifies that this platform-specific will only run on Android. The `ViewCell.SetIsContextActionsLegacyModeEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to enable the `ViewCell` context actions menu legacy mode, so that the context actions menu is not updated when the selected item in a `ListView` changes. In addition, the `ViewCell.GetIsContextActionsLegacyModeEnabled` method can be used to return whether the context actions legacy mode is enabled.

The following screenshots show `ViewCell` context actions legacy mode enabled:



In this mode, the displayed context action menu items are identical for cell 1 and cell 2, despite different context menu items being defined for cell 2.

The following screenshots show [ViewCell](#) context actions legacy mode disabled, which is the default Xamarin.Forms behavior:



In this mode, the correct context action menu items are displayed for cell 1 and cell 2.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

VisualElement Elevation on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

VisualElement Legacy Color Mode on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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 Android 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:

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](#).

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

WebView Mixed Content on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android 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 ...>
    <!DOCTYPE html>
    <html ...>
        <head ...>
            <meta ...>
        </head>
        <body ...>
            ...
        </body>
    </html>
</ContentPage>
```

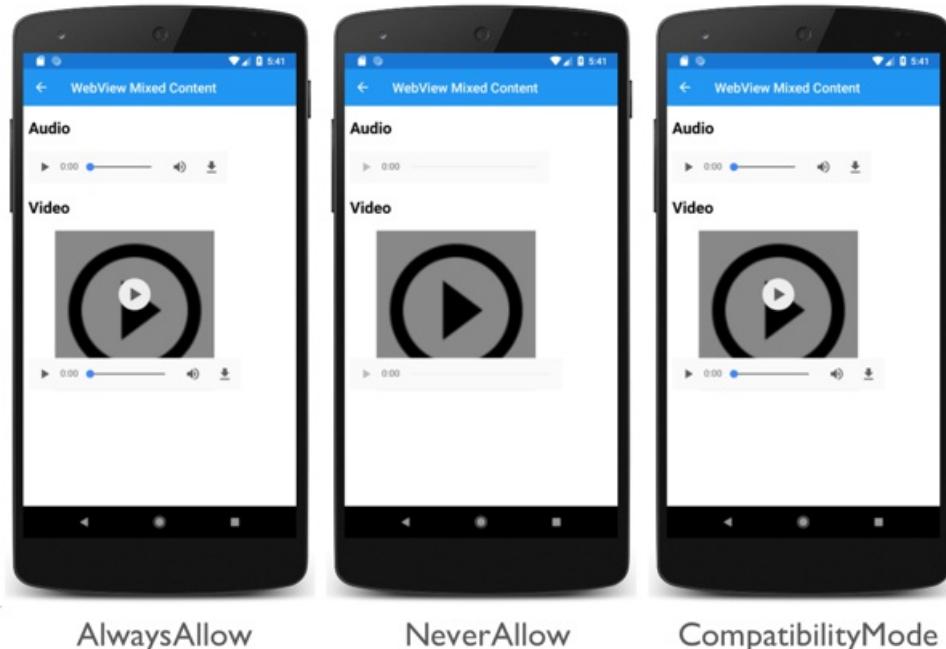
Alternatively, it can be consumed from C# using the fluent API:

```
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:



AlwaysAllow

NeverAllow

CompatibilityMode

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

WebView Zoom on Android

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Android platform-specific enables pinch-to-zoom and a zoom control on a `WebView`. It's consumed in XAML by setting the `WebView.EnableZoomControls` and `WebView.DisplayZoomControls` bindable properties to `boolean` values:

```
<ContentPage ...>
    xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
        <WebView Source="https://www.xamarin.com"
            android:WebView.EnableZoomControls="true"
            android:WebView.DisplayZoomControls="true" />
    </ContentPage>
```

The `WebView.EnableZoomControls` bindable property controls whether pinch-to-zoom is enabled on the `WebView`, and the `WebView.DisplayZoomControls` bindable property controls whether zoom controls are overlaid on the `WebView`.

Alternatively, the platform-specific can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...
webView.On<Android>()
    .EnableZoomControls(true)
    .DisplayZoomControls(true);
```

The `WebView.On<Android>` method specifies that this platform-specific will only run on Android. The `WebView.EnableZoomControls` method, in the `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` namespace, is used to control whether pinch-to-zoom is enabled on the `WebView`. The `WebView.DisplayZoomControls` method, in the same namespace, is used to control whether zoom controls are overlaid on the `WebView`. In addition, the `WebView.ZoomControlsEnabled` and `WebView.ZoomControlsDisplayed` methods can be used to return whether pinch-to-zoom and zoom controls are enabled, respectively.

The result is that pinch-to-zoom can be enabled on a `WebView`, and zoom controls can be overlaid on the `WebView`:



IMPORTANT

Zoom controls must be both enabled and displayed, via the respective bindable properties or methods, to be overlaid on a [WebView](#).

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [AndroidSpecific API](#)
- [AndroidSpecific.AppCompat API](#)

iOS platform features in Xamarin.Forms

8/4/2022 • 3 minutes to read • [Edit Online](#)

Developing Xamarin.Forms applications for iOS requires Visual Studio. The [supported platforms page](#) contains more information about the pre-requisites.

Platform-specifics

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 on iOS:

- Blur support for any `visualElement`. For more information, see [VisualElement Blur on iOS](#).
- Disabling legacy color mode on a supported `VisualElement`. For more information, see [VisualElement Legacy Color Mode on iOS](#).
- Enabling a drop shadow on a `VisualElement`. For more information, see [VisualElement Drop Shadows on iOS](#).
- Enabling a `visualElement` object to become the first responder to touch events. For more information, see [VisualElement First Responder](#).

The following platform-specific functionality is provided for Xamarin.Forms views on iOS:

- Setting the `cell` background color. For more information, see [Cell Background Color on iOS](#).
- Controlling when item selection occurs in a `DatePicker`. For more information, see [DatePicker Item Selection on iOS](#).
- Ensuring that inputted text fits into an `Entry` by adjusting the font size. For more information, see [Entry Font Size on iOS](#).
- Setting the cursor color in a `Entry`. For more information, see [Entry Cursor Color on iOS](#).
- Controlling whether `ListView` header cells float during scrolling. For more information, see [ListView Group Header Style on iOS](#).
- Controlling whether row animations are disabled when the `ListView` items collection is being updated. For more information, see [ListView Row Animations on iOS](#).
- Setting the separator style on a `ListView`. For more information, see [ListView Separator Style on iOS](#).
- Controlling when item selection occurs in a `Picker`. For more information, see [Picker Item Selection on iOS](#).
- Controlling whether a `SearchBar` has a background. For more information, see [SearchBar style on iOS](#).
- 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 [Slider Thumb Tap on iOS](#).
- Controlling the transition that's used when opening a `SwipeView`. For more information, see [SwipeView Swipe Transition Mode](#).
- Controlling when item selection occurs in a `TimePicker`. For more information, see [TimePicker Item Selection on iOS](#).

The following platform-specific functionality is provided for Xamarin.Forms pages on iOS:

- Controlling whether the detail page of a `FlyoutPage` has shadow applied to it, when revealing the flyout page. For more information, see [FlyoutPage Shadow](#).
- Hiding the navigation bar separator on a `NavigationPage`. For more information, see [NavigationPage Bar](#)

Separator on iOS.

- Controlling whether the navigation bar is translucent. For more information, see [Navigation Bar Translucency on iOS](#).
- Controlling whether the status bar text color on a [NavigationPage](#) is adjusted to match the luminosity of the navigation bar. For more information, see [NavigationPage Bar Text Color Mode on iOS](#).
- Controlling whether the page title is displayed as a large title in the page navigation bar. For more information, see [Large Page Titles on iOS](#).
- Setting the visibility of the home indicator on a [Page](#). For more information, see [Home Indicator Visibility on iOS](#).
- Setting the status bar visibility on a [Page](#). For more information, see [Page Status Bar Visibility on iOS](#).
- Ensuring that page content is positioned on an area of the screen that is safe for all iOS devices. For more information, see [Safe Area Layout Guide on iOS](#).
- Setting the presentation style of modal pages. For more information, see [Modal Page Presentation Style](#).
- Setting the translucency mode of the tab bar on a [TabbedPage](#). For more information, see [TabbedPage Translucent TabBar on iOS](#).

The following platform-specific functionality is provided for Xamarin.Forms layouts on iOS:

- Controlling whether a [ScrollView](#) handles a touch gesture or passes it to its content. For more information, see [ScrollView Content Touches on iOS](#).

The following platform-specific functionality is provided for the Xamarin.Forms [Application](#) class on iOS:

- Disabling accessibility scaling for named font sizes. For more information, see [Accessibility Scaling for Named Font Sizes on iOS](#).
- Enabling control layout and rendering updates to be performed on the main thread. For more information, see [Main Thread Control Updates on iOS](#).
- Enabling a [PanGestureRecognizer](#) in a scrolling view to capture and share the pan gesture with the scrolling view. For more information, see [Simultaneous Pan Gesture Recognition on iOS](#).

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 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.



Other iOS features

Using [custom renderers](#), the [DependencyService](#), and the [MessagingCenter](#), it's possible to incorporate a wide variety of native functionality into Xamarin.Forms applications for iOS.

Accessibility Scaling for Named Font Sizes on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific disables accessibility scaling for named font sizes. It's consumed in XAML by setting the `Application.EnableAccessibilityScalingForNamedFontSizes` bindable property to `false`:

```
<Application ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"  
        ios:Application.EnableAccessibilityScalingForNamedFontSizes="false">  
    ...  
</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>().SetEnableAccessibilityScalingForNamedFontSizes(false);
```

The `Application.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Application.SetEnableAccessibilityScalingForNamedFontSizes` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to disable named font sizes being scaled by the iOS accessibility settings. In addition, the `Application.GetEnableAccessibilityScalingForNamedFontSizes` method can be used to return whether named font sizes are scaled by iOS accessibility settings.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Cell Background Color on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific sets the default background color of `Cell` instances. It's consumed in XAML by setting the `Cell.DefaultBackgroundColor` bindable property to a `Color`:

```
<ContentPage ...>
    xmlns:ios="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout Margin="20">
            <ListView ItemsSource="{Binding GroupedEmployees}"
                IsGroupingEnabled="true">
                <ListView.GroupHeaderTemplate>
                    <DataTemplate>
                        <ViewCell ios:Cell.DefaultBackgroundColor="Teal">
                            <Label Margin="10,10"
                                Text="{Binding Key}"
                                FontAttributes="Bold" />
                        </ViewCell>
                    </DataTemplate>
                </ListView.GroupHeaderTemplate>
                ...
            </ListView>
        </StackLayout>
    </ContentPage>
```

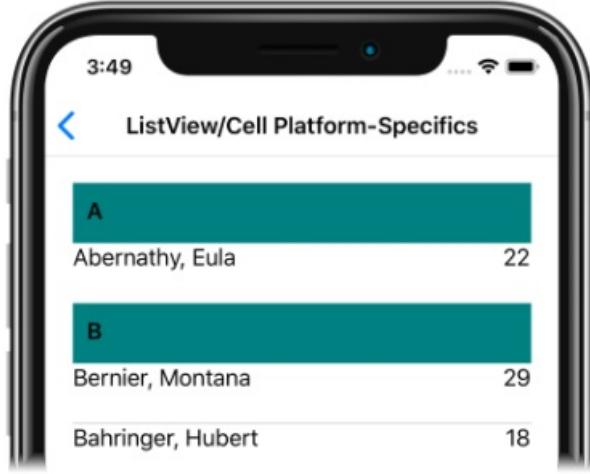
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

var viewCell = new ViewCell { View = ... };
viewCell.On<iOS>().SetDefaultBackgroundColor(Color.Teal);
```

The `ListView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Cell.SetDefaultBackgroundColor` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, sets the cell background color to a specified `Color`. In addition, the `Cell.DefaultBackgroundColor` method can be used to retrieve the current cell background color.

The result is that the background color in a `Cell` can be set to a specific `Color`:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

DatePicker item selection on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific controls when item selection occurs in a `DatePicker`, 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 `DatePicker.UpdateMode` attached property to a value of the `UpdateMode` enumeration:

```
<ContentPage ...>
    xmlns:ios="clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            <DatePicker MinimumDate="01/01/2020"
                        MaximumDate="12/31/2020"
                        ios:DatePicker.UpdateMode="WhenFinished" />
            ...
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
datePicker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
```

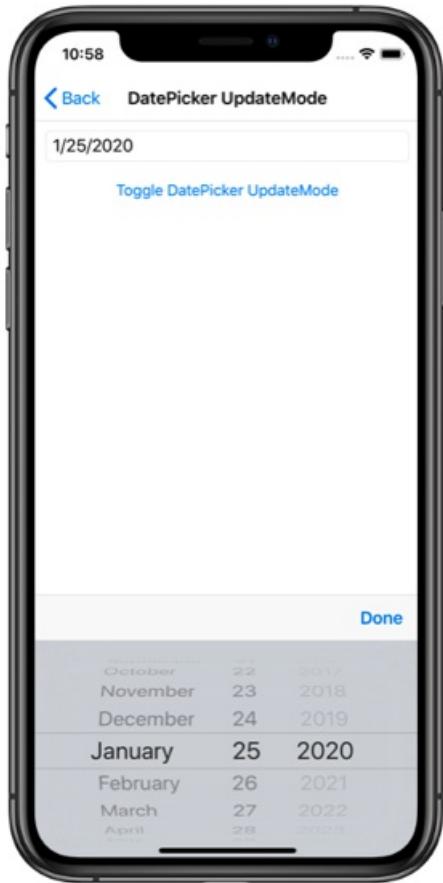
The `DatePicker.On<iOS>` method specifies that this platform-specific will only run on iOS. The `DatePicker.SetUpdateMode` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control when item selection occurs, with the `UpdateMode` enumeration providing two possible values:

- `Immediately` – item selection occurs as the user browses items in the `DatePicker`. This is the default behavior in Xamarin.Forms.
- `WhenFinished` – item selection only occurs once the user has pressed the **Done** button in the `DatePicker`.

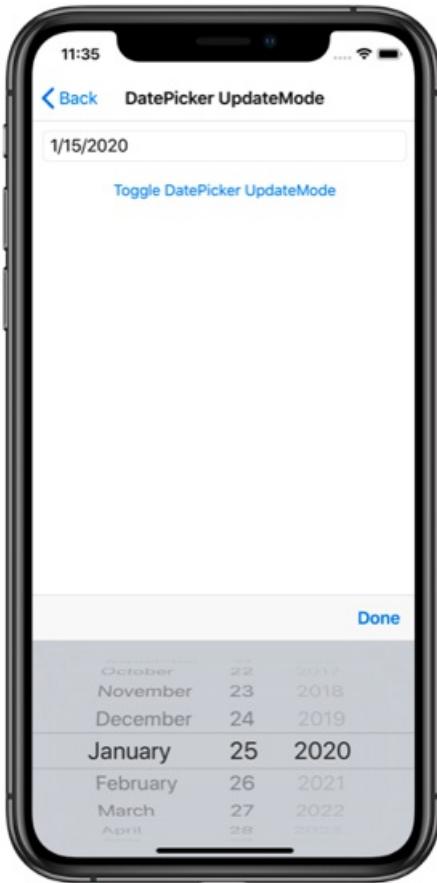
In addition, the `SetUpdateMode` method can be used to toggle the enumeration values by calling the `UpdateMode` method, which returns the current `UpdateMode`:

```
switch (datePicker.On<iOS>().UpdateMode())
{
    case UpdateMode.Immediately:
        datePicker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
        break;
    case UpdateMode.WhenFinished:
        datePicker.On<iOS>().SetUpdateMode(UpdateMode.Immediately);
        break;
}
```

The result is that a specified `UpdateMode` is applied to the `DatePicker`, which controls when item selection occurs:



UpdateMode.Immediately



UpdateMode.WhenFinished

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Entry Cursor Color on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific sets the cursor color of an `Entry` to a specified color. It's consumed in XAML by setting the `Entry.CursorColor` bindable property to a `Color`:

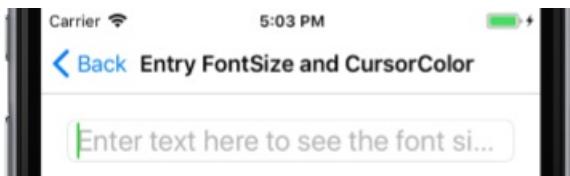
```
<ContentPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">  
    <StackLayout>  
        <Entry ... ios:Entry.CursorColor="LimeGreen" />  
    </StackLayout>  
</ContentPage>
```

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`:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Entry Font Size on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS 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.iOS;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>
```

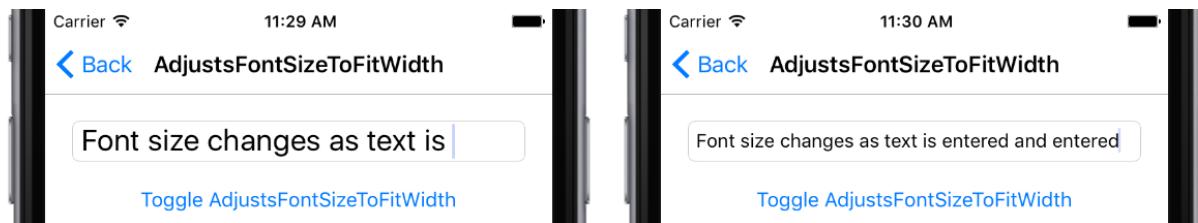
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOS;
...
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.iOS` 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.iOS` 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:



Related links

- [PlatformSpecifics \(sample\)](#)

- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

FlyoutPage Shadow on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This platform-specific controls whether the detail page of a `FlyoutPage` has shadow applied to it, when revealing the flyout page. It's consumed in XAML by setting the `FlyoutPage.ApplyShadow` bindable property to `true`:

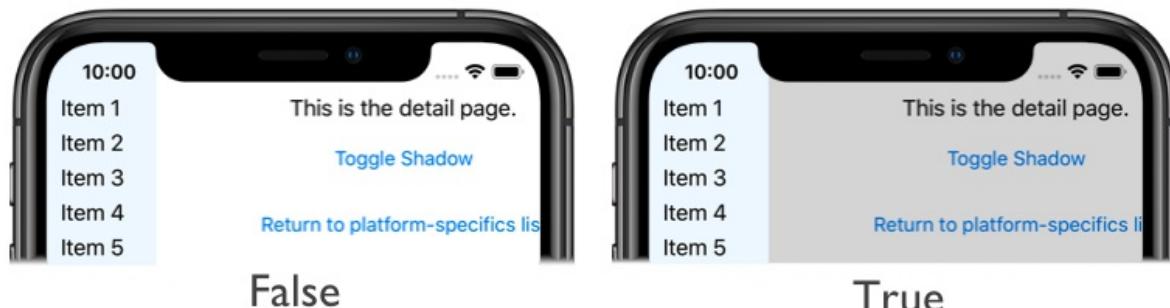
```
<FlyoutPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"  
    ios:FlyoutPage.ApplyShadow="true">  
    ...  
</FlyoutPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
public class iOSFlyoutPageCS : FlyoutPage  
{  
    public iOSFlyoutPageCS(ICommand restore)  
    {  
        On<iOS>().SetApplyShadow(true);  
        // ...  
    }  
}
```

The `FlyoutPage.On<iOS>` method specifies that this platform-specific will only run on iOS. The `FlyoutPage.SetApplyShadow` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether the detail page of a `FlyoutPage` has shadow applied to it, when revealing the flyout page. In addition, the `GetApplyShadow` method can be used to determine whether shadow is applied to the detail page of a `FlyoutPage`.

The result is that the detail page of a `FlyoutPage` can have shadow applied to it, when revealing the flyout page:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)

- [iOSSpecific API](#)

Adding iOS-specific Formatting

8/4/2022 • 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 `TabPage` 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 TabbedPage` 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](#)

Modal Page Presentation Style on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific is used to set the presentation style of a modal page, and in addition can be used to display modal pages that have transparent backgrounds. 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="OverFullScreen">  
    ...  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

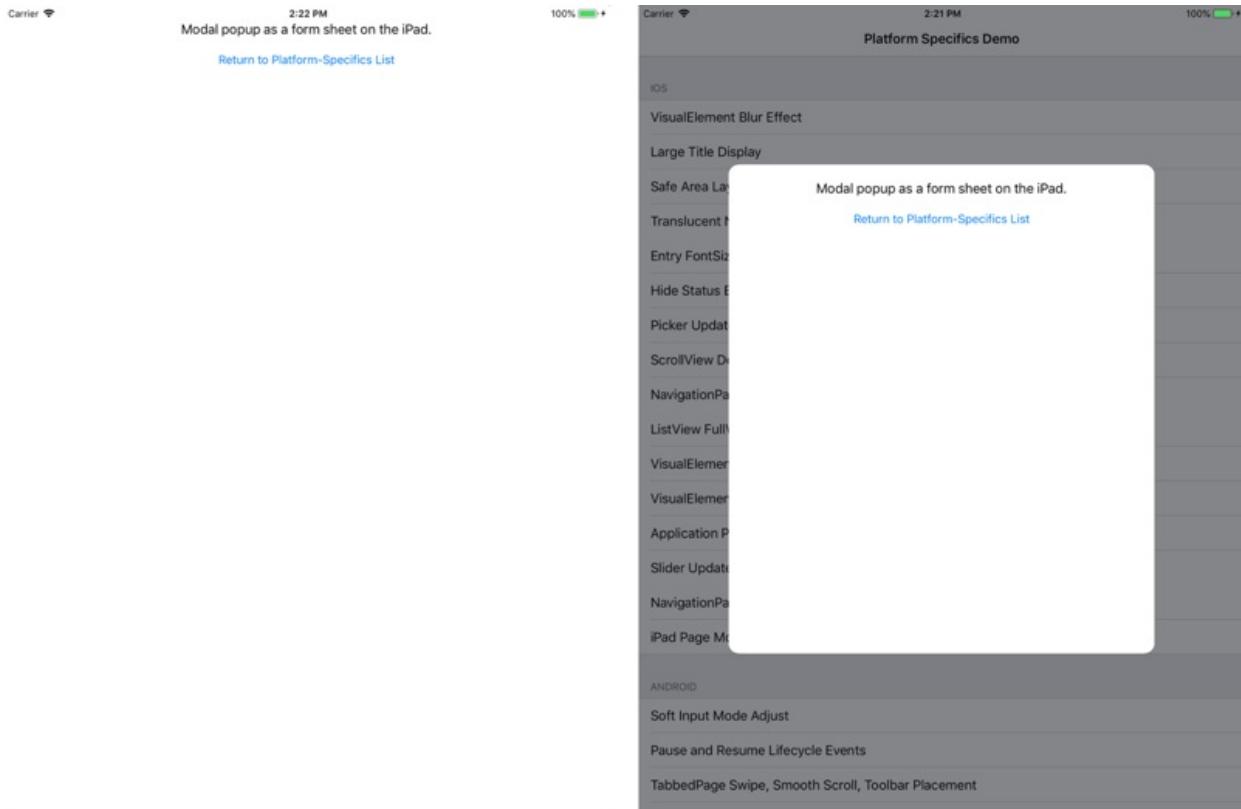
```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
public class iOSModalFormSheetPageCS : ContentPage  
{  
    public iOSModalFormSheetPageCS()  
    {  
        On<iOS>().SetModalPresentationStyle(UIModalPresentationStyle.OverFullScreen);  
        ...  
    }  
}
```

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.
- `Automatic`, which sets the modal presentation style to the default chosen by the system. For most view controllers, `UIKit` maps this to `UIModalPresentationStyle.PageSheet`, but some system view controllers may map it to a different style.
- `OverFullScreen`, which sets the modal presentation style to cover the screen.
- `PageSheet`, which sets the modal presentation style to cover the underlying content.

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](#).

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Large Page Titles on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific is used to display the page title as a large title on the navigation bar of a `NavigationPage`, 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 `LargeTitleDisplayMode` 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.

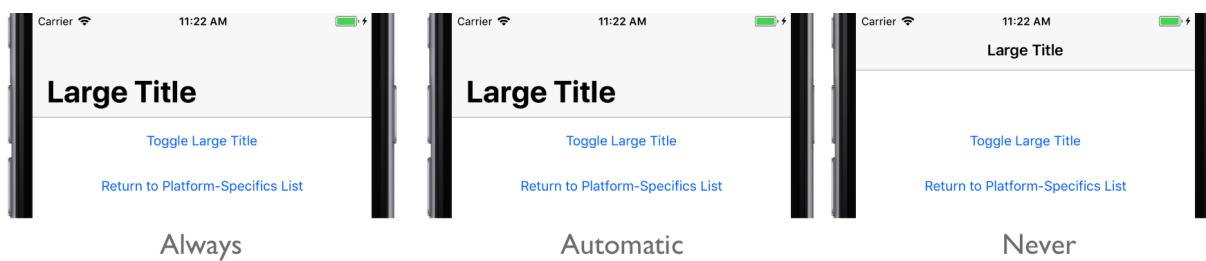
In addition, the `SetLargeTitleDisplay` method can be used to toggle the enumeration values by calling the `LargeTitleDisplayStyle` method, which returns the current `LargeTitleDisplayStyle`:

```

switch (On<iOS>().LargeTitleDisplayStyle)
{
    case LargeTitleDisplayStyle.Always:
        On<iOS>().SetLargeTitleDisplayStyle(LargeTitleDisplayStyle.Automatic);
        break;
    case LargeTitleDisplayStyle.Automatic:
        On<iOS>().SetLargeTitleDisplayStyle(LargeTitleDisplayStyle.Never);
        break;
    case LargeTitleDisplayStyle.Never:
        On<iOS>().SetLargeTitleDisplayStyle(LargeTitleDisplayStyle.Always);
        break;
}

```

The result is that a specified `LargeTitleDisplayStyle` is applied to the `Page`, which controls the large title behavior:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

ListView Group Header Style on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific controls whether `ListView` header cells float during scrolling. It's consumed in XAML by setting the `ListView.GroupHeaderStyle` bindable property to a value of the `GroupHeaderStyle` enumeration:

```
<ContentPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">  
    <StackLayout Margin="20">  
        <ListView ... ios:ListView.GroupHeaderStyle="Grouped">  
            ...  
        </ListView>  
    </StackLayout>  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

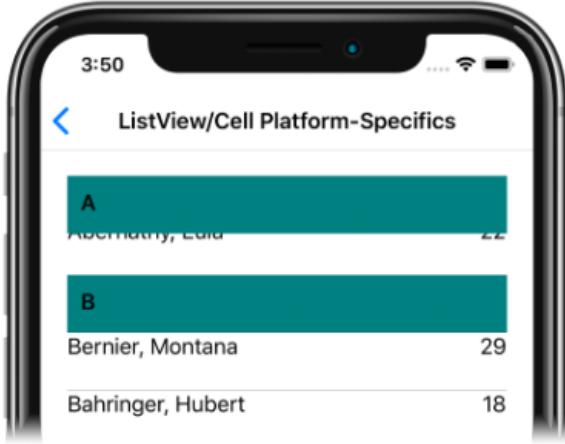
```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
listView.On<iOS>().SetGroupHeaderStyle(GroupHeaderStyle.Grouped);
```

The `ListView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `ListView.SetGroupHeaderStyle` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether `ListView` header cells float during scrolling. The `GroupHeaderStyle` enumeration provides two possible values:

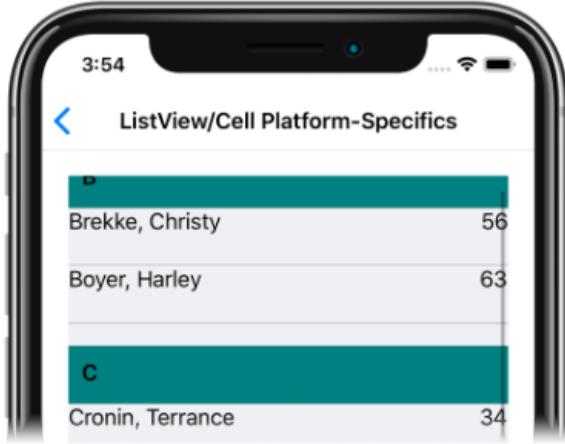
- `Plain` – indicates that header cells float when the `ListView` is scrolled (default).
- `Grouped` – indicates that header cells do not float when the `ListView` is scrolled.

In addition, the `ListView.GetGroupHeaderStyle` method can be used to return the `GroupHeaderStyle` that's applied to the `ListView`.

The result is that a specified `GroupHeaderStyle` value is applied to the `ListView`, which controls whether header cells float during scrolling:



Plain



Grouped

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

ListView Row Animations on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific controls whether row animations are disabled when the `ListView` items collection is being updated. It's consumed in XAML by setting the `ListView.RowAnimationsEnabled` bindable property to `false`:

```
<ContentPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">  
    <StackLayout Margin="20">  
        <ListView ... ios:ListView.RowAnimationsEnabled="false">  
            ...  
        </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>().SetRowAnimationsEnabled(false);
```

The `ListView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `ListView.SetRowAnimationsEnabled` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether row animations are disabled when the `ListView` items collection is being updated. In addition, the `ListView.GetRowAnimationsEnabled` method can be used to return whether row animations are disabled on the `ListView`.

NOTE

`ListView` row animations are enabled by default. Therefore, an animation occurs when a new row is inserted into a `ListView`.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

ListView Separator Style on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This iOS 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.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Main Thread Control Updates on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS 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.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

NavigationPage Bar Separator on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This iOS 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`:

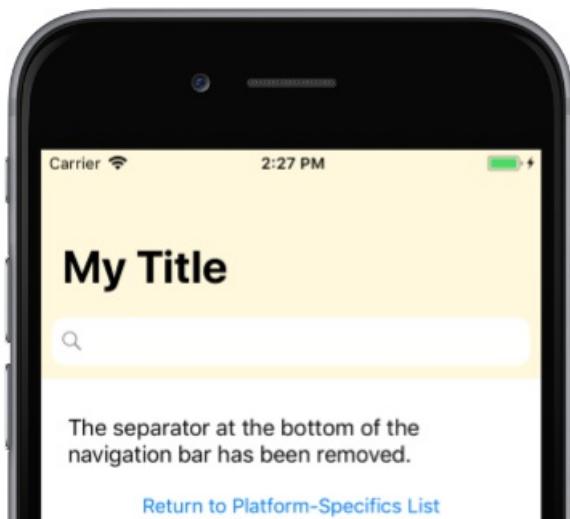
```
<NavigationPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"  
    ios:NavigationBar.HideNavigationBarSeparator="true">  
  
</NavigationPage>
```

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:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

NavigationPage Bar Text Color Mode on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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:

```
<FlyoutPage 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">
    <FlyoutPage.Flyout>
        <ContentPage Title="Flyout Page Title" />
    </FlyoutPage.Flyout>
    <FlyoutPage.Detail>
        <NavigationPage BarBackgroundColor="Blue" BarTextColor="White"
            ios:NavigationBar.StatusBarTextColorMode="MatchNavigationBarTextLuminosity">
            <x:Arguments>
                <ContentPage>
                    <Label Text="Slide the master page to see the status bar text color mode change." />
                </ContentPage>
            </x:Arguments>
        </NavigationPage>
    </FlyoutPage.Detail>
</FlyoutPage>
```

Alternatively, it can be consumed from C# using the fluent API:

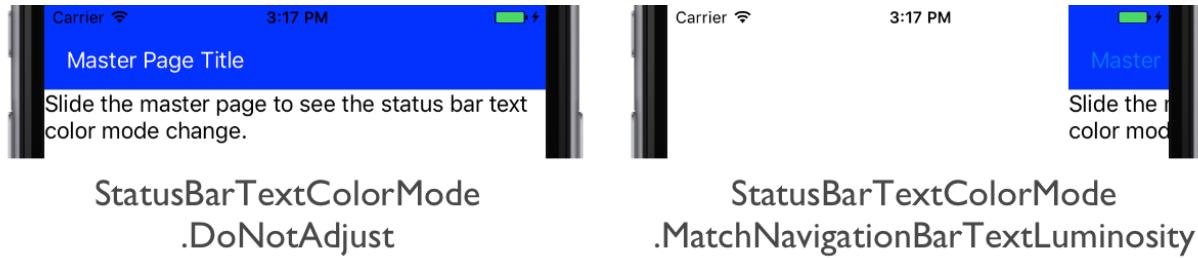
```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
IsPresentedChanged += (sender, e) =>
{
    var flyoutPage = sender as FlyoutPage;
    if (flyoutPage.IsPresented)
        ((Xamarin.Forms.NavigationPage)flyoutPage.Detail)
            .On<iOS>()
            .SetStatusBarTextColorMode(StatusBarTextColorMode.DoNotAdjust);
    else
        ((Xamarin.Forms.NavigationPage)flyoutPage.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 `Flyout` and `Detail` pages of a `FlyoutPage`:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

NavigationPage Bar Translucency on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This iOS platform-specific is used to change the transparency of the navigation bar on a `NavigationPage`, and is consumed in XAML by setting the `NavigationPage.IsNavigationBarTranslucent` attached property to a `boolean` value:

```
<NavigationPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"  
    BackgroundColor="Blue"  
    ios:NavigationPage.IsNavigationBarTranslucent="true">  
    ...  
</NavigationPage>
```

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:



Related links

- [PlatformSpecifics \(sample\)](#)

- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Home Indicator Visibility on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This iOS platform-specific sets the visibility of the home indicator on a `Page`. It's consumed in XAML by setting the `Page.PrefersHomeIndicatorAutoHidden` bindable property to a `boolean`:

```
<ContentPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"  
    ios:Page.PrefersHomeIndicatorAutoHidden="true">  
    ...  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
On<iOS>().SetPrefersHomeIndicatorAutoHidden(true);
```

The `Page.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Page.SetPrefersHomeIndicatorAutoHidden` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, controls the visibility of the home indicator. In addition, the `Page.PrefersHomeIndicatorAutoHidden` method can be used to retrieve the visibility of the home indicator.

The result is that the visibility of the home indicator on a `Page` can be controlled:



NOTE

This platform-specific can be applied to `ContentPage`, `FlyoutPage`, `NavigationPage`, and `TabbedPage` objects.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Page Status Bar Visibility on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This iOS 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:

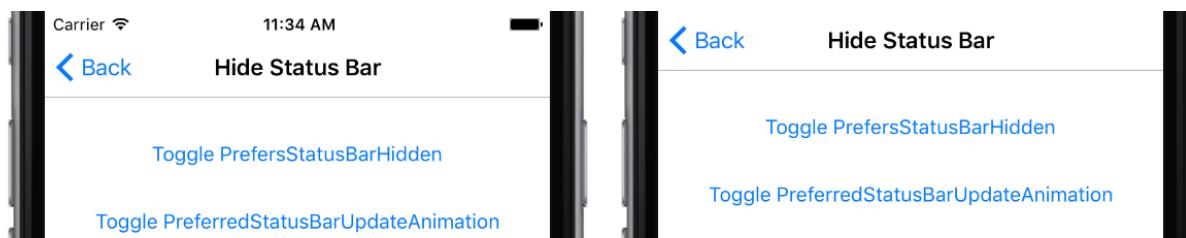
```
<ContentPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"  
        ios:Page.PrefersStatusBarHidden="True"  
        ios:Page.PreferredStatusBarUpdateAnimation="Fade">  
    ...  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
On<iOS>().SetPrefersStatusBarHidden(StatusBarHiddenMode.True)  
    .SetPreferredStatusBarUpdateAnimation(UIStatusBarAnimation.Fade);
```

The `Page.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Page.SetPrefersStatusBarHidden` 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 `TabPage`, 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`.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Picker Item Selection on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS 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.iOSSpecific;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.iOSSpecific;
...
picker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
```

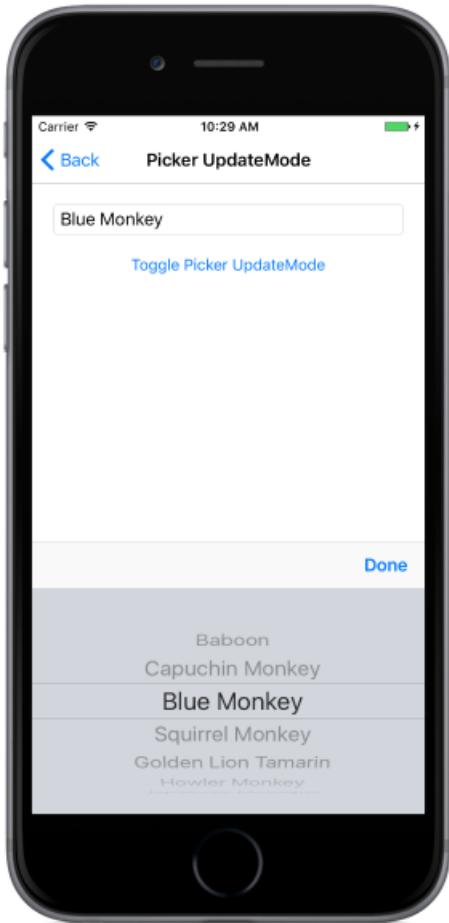
The `Picker.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Picker.SetUpdateMode` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control when item selection occurs, with the `UpdateMode` enumeration providing two possible values:

- `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`.

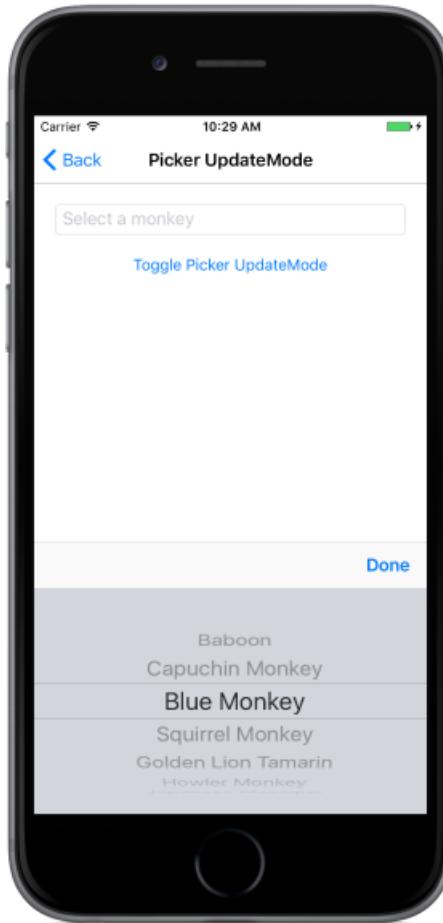
In addition, the `SetUpdateMode` method can be used to toggle the enumeration values by calling the `UpdateMode` method, which returns the current `UpdateMode`:

```
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:



UpdateMode.Immediately



UpdateMode.WhenFinished

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Safe Area Layout Guide on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS 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 `OnAppearing` override:

```
protected override void OnAppearing()
{
    base.OnAppearing();

    var safeInsets = On<iOS>().SafeAreaInsets();
    safeInsets.Left = 20;
    Padding = safeInsets;
}
```

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

ScrollView Content Touches on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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:

```
<FlyoutPage ...  
    xmlns:ios="clr-  
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">  
    <FlyoutPage.Flyout>  
        <ContentPage Title="Menu" BackgroundColor="Blue" />  
    </FlyoutPage.Flyout>  
    <FlyoutPage.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>  
    </FlyoutPage.Detail>  
</FlyoutPage>
```

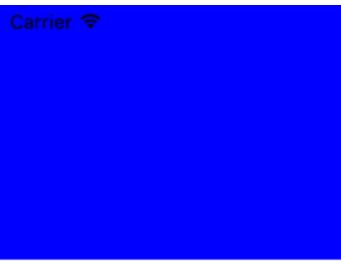
Alternatively, it can be consumed from C# using the fluent API:

```
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 `FlyoutPage`:

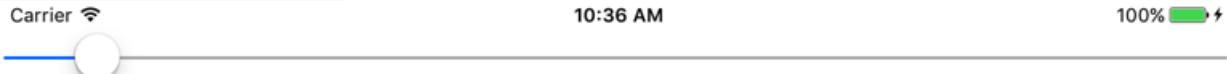


10:35 AM

100%

Toggle ScrollView DelayContentTouches

ShouldDelayContentTouches = true



10:36 AM

100%

Toggle ScrollView DelayContentTouches

ShouldDelayContentTouches = false

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

SearchBar style on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific controls whether a `SearchBar` has a background. It's consumed in XAML by setting the `SearchBar.SearchBarStyle` bindable property to a value of the `UISearchBarStyle` enumeration:

```
<ContentPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">  
    <StackLayout>  
        <SearchBar ios:SearchBar.SearchBarStyle="Minimal"  
            Placeholder="Enter search term" />  
        ...  
    </StackLayout>  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

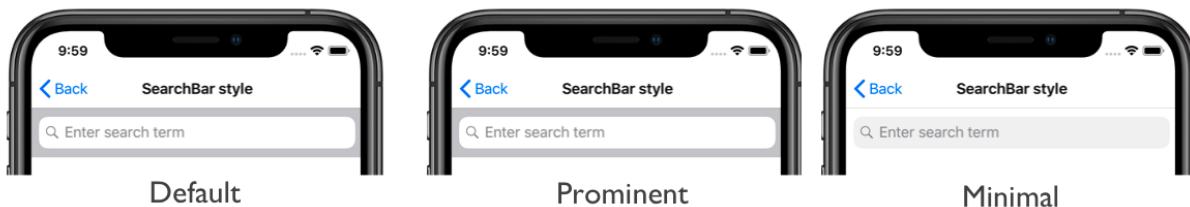
```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
SearchBar searchBar = new SearchBar { Placeholder = "Enter search term" };  
searchBar.On<iOS>().SetearchBarStyle(UISearchBarStyle.Minimal);
```

The `SearchBar.On<iOS>` method specifies that this platform-specific will only run on iOS. The `SearchBar.SetearchBarStyle` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control whether the `SearchBar` has a background. The `UISearchBarStyle` enumeration provides three possible values:

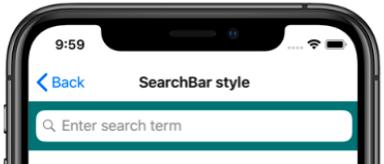
- `Default` indicates that the `SearchBar` has the default style. This is the default value of the `SearchBar.SearchBarStyle` bindable property.
- `Prominent` indicates that the `SearchBar` has a translucent background, and the search field is opaque.
- `Minimal` indicates that the `SearchBar` has no background, and the search field is translucent.

In addition, the `SearchBar.GetearchBarStyle` method can be used to return the `UISearchBarStyle` that's applied to the `SearchBar`.

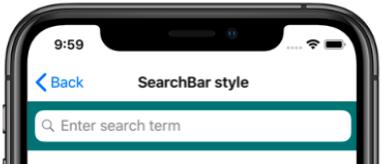
The result is that a specified `UISearchBarStyle` member is applied to a `SearchBar`, which controls whether the `SearchBar` has a background:



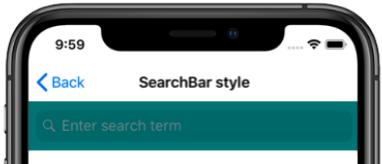
The following screenshots show the `UISearchBarStyle` members applied to `SearchBar` objects that have their `BackgroundColor` property set:



Default



Prominent



Minimal

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Simultaneous Pan Gesture Recognition on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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 iOS 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.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Slider Thumb Tap on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This iOS 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`:

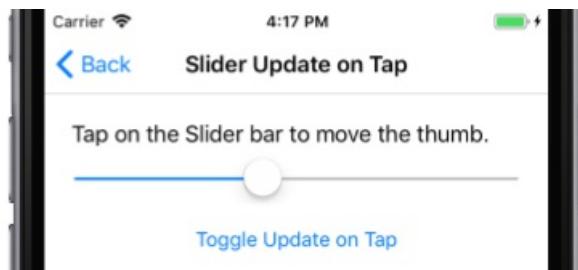
```
<ContentPage ...>
    xmlns:ios="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout ...>
            <Slider ... ios:Slider.UpdateOnTap="true" />
            ...
        </StackLayout>
    </ContentPage>
```

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:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

SwipeView Swipe Transition Mode on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific controls the transition that's used when opening a `SwipeView`. It's consumed in XAML by setting the `SwipeView.SwipeTransitionMode` bindable property to a value of the `SwipeTransitionMode` enumeration:

```
<ContentPage ...>
    xmlns:ios="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            <SwipeView ios:SwipeView.SwipeTransitionMode="Drag">
                <SwipeView.LeftItems>
                    <SwipeItems>
                        <SwipeItem Text="Delete"
                            IconImageSource="delete.png"
                            BackgroundColor="LightPink"
                            Invoked="OnDeleteSwipeItemInvoked" />
                    </SwipeItems>
                </SwipeView.LeftItems>
                <!-- Content -->
            </SwipeView>
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

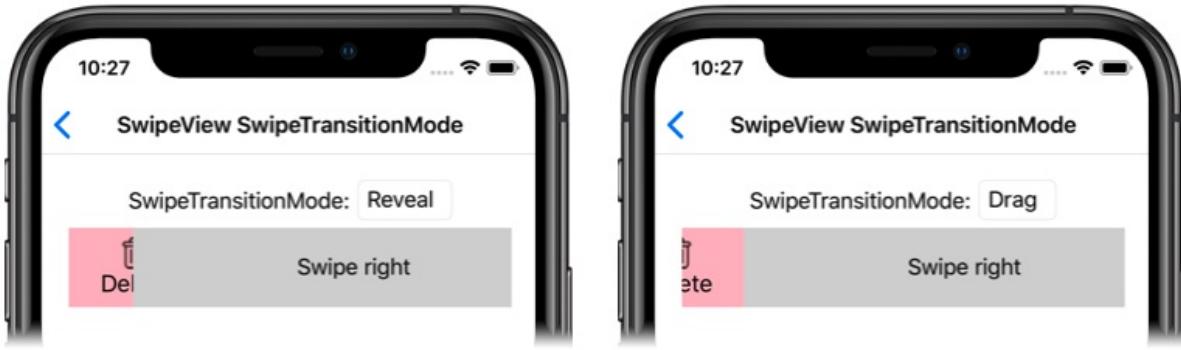
```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
SwipeView swipeView = new Xamarin.Forms.SwipeView();
swipeView.On<iOS>().SetSwipeTransitionMode(SwipeTransitionMode.Drag);
// ...
```

The `SwipeView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `SwipeView.SetSwipeTransitionMode` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control the transition that's used when opening a `SwipeView`. The `SwipeTransitionMode` enumeration provides two possible values:

- `Reveal` indicates that the swipe items will be revealed as the `SwipeView` content is swiped, and is the default value of the `SwipeView.SwipeTransitionMode` property.
- `Drag` indicates that the swipe items will be dragged into view as the `SwipeView` content is swiped.

In addition, the `SwipeView.GetSwipeTransitionMode` method can be used to return the `SwipeTransitionMode` that's applied to the `SwipeView`.

The result is that a specified `SwipeTransitionMode` value is applied to the `SwipeView`, which controls the transition that's used when opening the `SwipeView`:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

TabPage translucent tab bar on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific is used to set the translucency mode of the tab bar on a `TabPage`. It's consumed in XAML by setting the `TabPage.TranslucencyMode` bindable property to a `TranslucencyMode` enumeration value:

```
<TabPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"  
    ios:TabPage.TranslucencyMode="Opaque">  
    ...  
</TabPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
On<iOS>().SetTranslucencyMode(TranslucencyMode.Opaque);
```

The `TabPage.On<iOS>` method specifies that this platform-specific will only run on iOS. The `TabPage.SetTranslucencyMode` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to set the translucency mode of the tab bar on a `TabPage` by specifying one of the following `TranslucencyMode` enumeration values:

- `Default`, which sets the tab bar to its default translucency mode. This is the default value of the `TabPage.TranslucencyMode` property.
- `Translucent`, which sets the tab bar to be translucent.
- `Opaque`, which sets the tab bar to be opaque.

In addition, the `GetTranslucencyMode` method can be used to retrieve the current value of the `TranslucencyMode` enumeration that's applied to the `TabPage`.

The result is that the translucency mode of the tab bar on a `TabPage` can be set:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

TimePicker item selection on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This iOS platform-specific controls when item selection occurs in a `TimePicker`, 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 `TimePicker.UpdateMode` attached property to a value of the `UpdateMode` enumeration:

```
<ContentPage ...>
    xmlns:ios="clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            <TimePicker Time="14:00:00"
                ios:TimePicker.UpdateMode="WhenFinished" />
            ...
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...
timePicker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
```

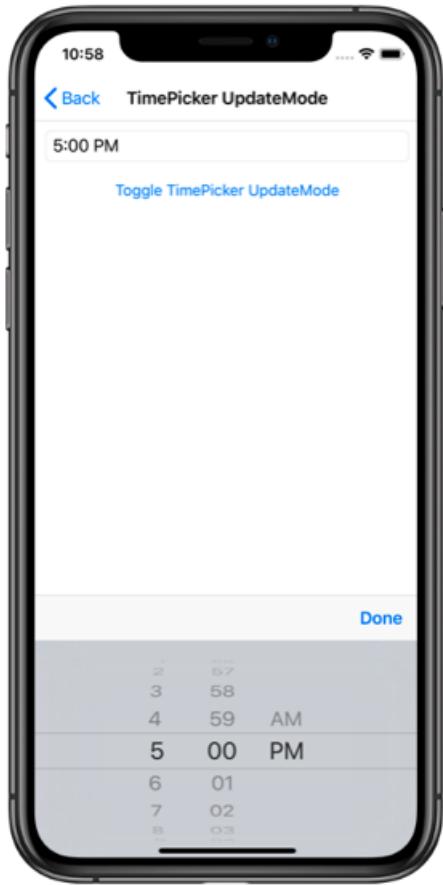
The `TimePicker.On<iOS>` method specifies that this platform-specific will only run on iOS. The `TimePicker.SetUpdateMode` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to control when item selection occurs, with the `UpdateMode` enumeration providing two possible values:

- `Immediately` – item selection occurs as the user browses items in the `TimePicker`. This is the default behavior in Xamarin.Forms.
- `WhenFinished` – item selection only occurs once the user has pressed the **Done** button in the `TimePicker`.

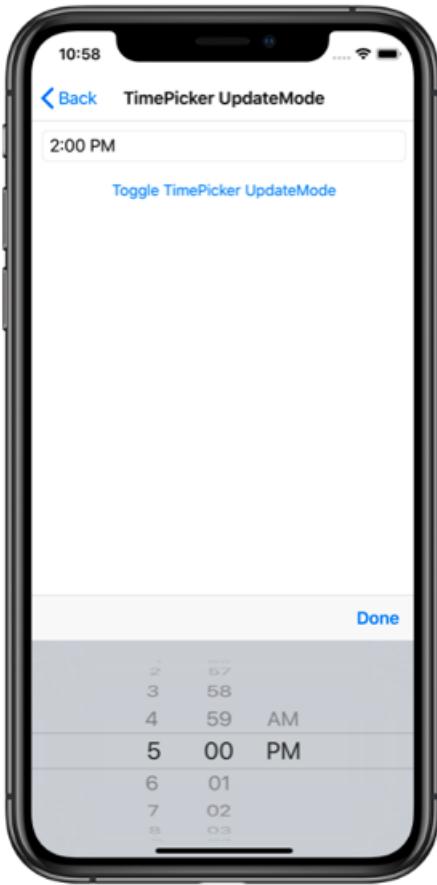
In addition, the `SetUpdateMode` method can be used to toggle the enumeration values by calling the `UpdateMode` method, which returns the current `updateMode`:

```
switch (timePicker.On<iOS>().UpdateMode())
{
    case UpdateMode.Immediately:
        timePicker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
        break;
    case UpdateMode.WhenFinished:
        timePicker.On<iOS>().SetUpdateMode(UpdateMode.Immediately);
        break;
}
```

The result is that a specified `UpdateMode` is applied to the `TimePicker`, which controls when item selection occurs:



UpdateMode.Immediately



UpdateMode.WhenFinished

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

VisualElement Blur on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This iOS platform-specific is used to blur the content layered beneath it, and can be applied to any `VisualElement`. It's 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">  
    ...  
    <Image Source="monkeyface.png"  
        ios:VisualElement.BlurEffect="ExtraLight" />  
    ...  
</ContentPage>
```

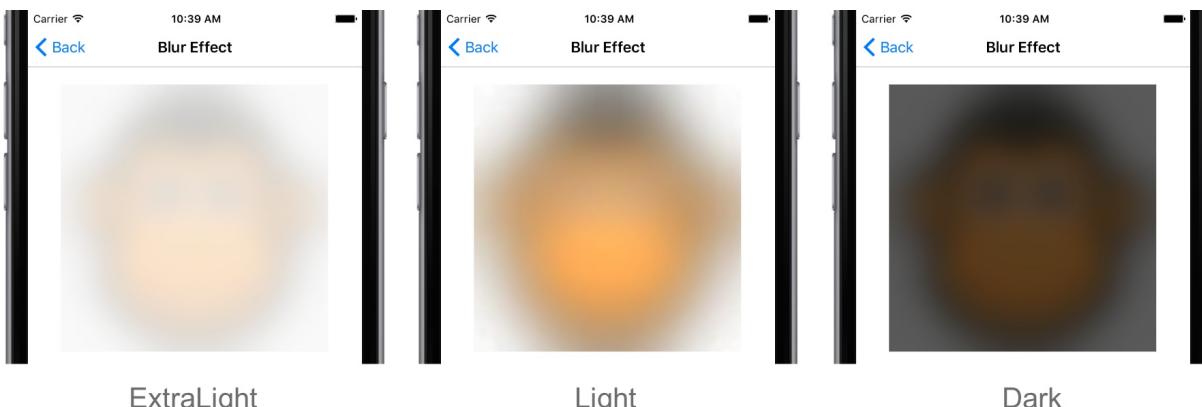
Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
image.On<iOS>().UseBlurEffect(BlurEffectStyle.ExtraLight);
```

The `Image.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`
- `Dark`

The result is that a specified `BlurEffectStyle` is applied to the `Image`:



NOTE

When adding a blur effect to a `VisualElement`, touch events will still be received by the `VisualElement`.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

VisualElement Drop Shadows on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific is used to enable a drop shadow on a `VisualElement`. It's consumed in XAML by setting the `VisualElement.IsShadowEnabled` 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.iOSSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout Margin="20">
            <BoxView ...
                ios:VisualElement.IsShadowEnabled="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.iOSSpecific;
...

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`:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

VisualElement Legacy Color Mode on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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 iOS platform-specific disables this legacy color mode on a `VisualElement`, 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:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">  
    <StackLayout>  
        ...  
        <Button Text="Button"  
            TextColor="Blue"  
            BackgroundColor="Bisque"  
            ios:VisualElement.IsEnabled="False" />  
        ...  
    </StackLayout>  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
_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.iOSSpecific` 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](#).

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

VisualElement first responder on iOS

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This iOS platform-specific enables a `VisualElement` object to become the first responder to touch events, rather than the page containing the element. It's consumed in XAML by setting the `VisualElement.CanBecomeFirstResponder` bindable property to `true`:

```
<ContentPage ...  
    xmlns:ios="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">  
    <StackLayout>  
        <Entry Placeholder="Enter text" />  
        <Button ios:VisualElement.CanBecomeFirstResponder="True"  
            Text="OK" />  
    </StackLayout>  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;  
...  
  
Entry entry = new Entry { Placeholder = "Enter text" };  
Button button = new Button { Text = "OK" };  
button.On<iOS>().SetCanBecomeFirstResponder(true);
```

The `VisualElement.On<iOS>` method specifies that this platform-specific will only run on iOS. The `VisualElement.SetCanBecomeFirstResponder` method, in the `Xamarin.Forms.PlatformConfiguration.iOSSpecific` namespace, is used to set the `VisualElement` to become the first responder for touch events. In addition, the `VisualElement.CanBecomeFirstResponder` method can be used to return whether the `VisualElement` is the first responder to touch events.

The result is that a `VisualElement` can become the first responder for touch events, rather than the page containing the element. This enables scenarios such as chat applications not dismissing a keyboard when a `Button` is tapped.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [iOSSpecific API](#)

Windows Platform Features

8/4/2022 • 2 minutes to read • [Edit Online](#)

Developing Xamarin.Forms applications for Windows platforms requires Visual Studio. The [supported platforms page](#) contains more information about the pre-requisites.



Platform-specifics

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 on the Universal Windows Platform (UWP):

- Setting an access key for a `VisualElement`. For more information, see [VisualElement Access Keys on Windows](#).
- Disabling legacy color mode on a supported `VisualElement`. For more information, see [VisualElement Legacy Color Mode on Windows](#).

The following platform-specific functionality is provided for Xamarin.Forms views on UWP:

- Detecting reading order from text content in `Entry`, `Editor`, and `Label` instances. For more information, see [InputView Reading Order on Windows](#).
- Enabling tap gesture support in a `ListView`. For more information, see [ListView SelectionMode on Windows](#).
- Enabling the pull direction of a `RefreshView` to be changed. For more information, see [RefreshView Pull Direction on Windows](#).
- Enabling a `searchBar` to interact with the spell check engine. For more information, see [SearchBar Spell Check on Windows](#).
- Setting the thread on which a `WebView` hosts its content. For more information, see [WebView Execution Mode on Windows](#).
- Enabling a `WebView` to display JavaScript alerts in a UWP message dialog. For more information, see [WebView JavaScript Alerts on Windows](#).

The following platform-specific functionality is provided for Xamarin.Forms pages on UWP:

- Collapsing the `FlyoutPage` navigation bar. For more information, see [FlyoutPage Navigation Bar on Windows](#).
- Setting toolbar placement options. For more information, see [Page Toolbar Placement on Windows](#).

- Enabling page icons to be displayed on a [TabbedPage](#) toolbar. For more information, see [TabPage Icons on Windows](#).

The following platform-specific functionality is provided for the `Xamarin.Forms Application` class on UWP:

- Specifying the directory in the project that image assets will be loaded from. For more information, see [Default Image Directory on Windows](#).

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](#)

Default image directory on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Universal Windows Platform platform-specific defines the directory in the project that image assets will be loaded from. It's consumed in XAML by setting the `Application.ImageDirectory` to a `string` that represents the project directory that contains image assets:

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"
    ...
    windows:Application.ImageDirectory="Assets">
...
</Application>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...
Application.Current.On<Windows>().SetImageDirectory("Assets");
```

The `Application.On<Windows>` method specifies that this platform-specific will only run on the Universal Windows Platform. The `Application.SetImageDirectory` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to specify the project directory that images will be loaded from. In addition, the `GetImageDirectory` method can be used to return a `string` that represents the project directory that contains the application image assets.

The result is that all images used in an application will be loaded from the specified project directory.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

FlyoutPage Navigation Bar on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Universal Windows Platform platform-specific is used to collapse the navigation bar on a `FlyoutPage`, and is consumed in XAML by setting the `FlyoutPage.CollapseStyle` and `FlyoutPage.CollapsedPaneWidth` attached properties:

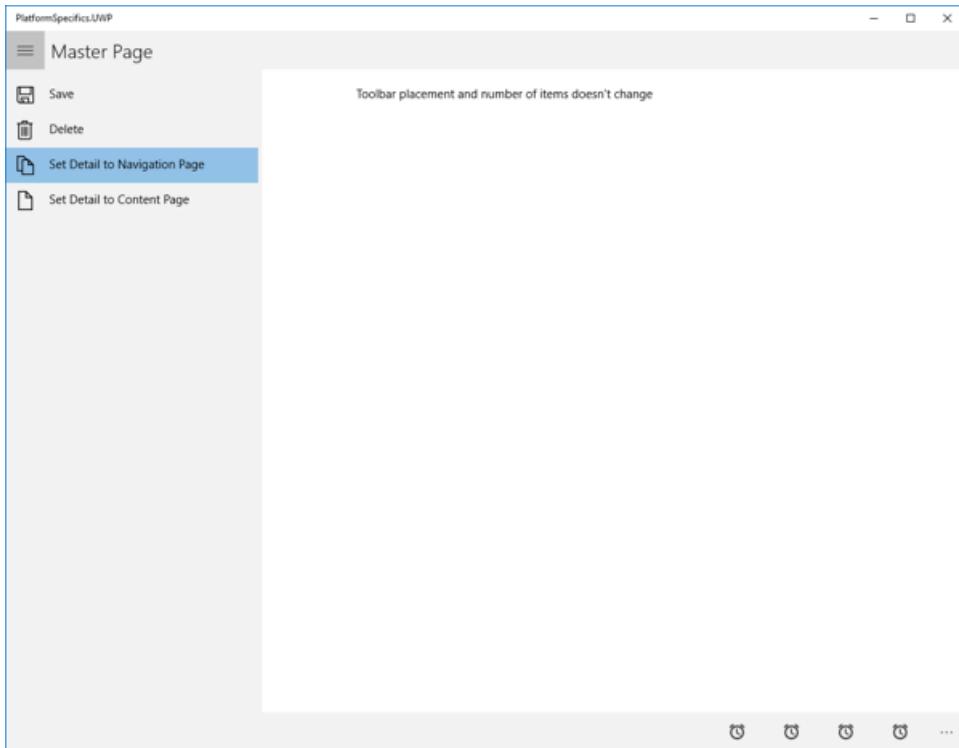
```
<FlyoutPage ...  
    xmlns:windows="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"  
    windows:FlyoutPage.CollapseStyle="Partial"  
    windows:FlyoutPage.CollapsedPaneWidth="48">  
    ...  
</FlyoutPage>
```

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 `FlyoutPage.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 `FlyoutPage.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 `FlyoutPage` instance, with the width also being specified:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

InputView Reading Order on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This Universal Windows Platform 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>
```

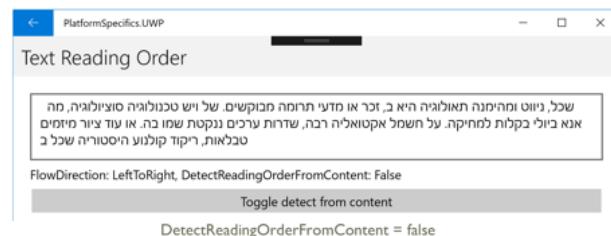
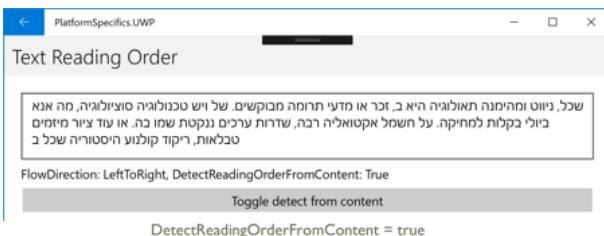
Alternatively, it can be consumed from C# using the fluent API:

```
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.GetDetectReadingOrderFromContent` 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.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

ListView SelectionMode on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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 Universal Windows Platform 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.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

Page Toolbar Placement on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Universal Windows Platform 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:

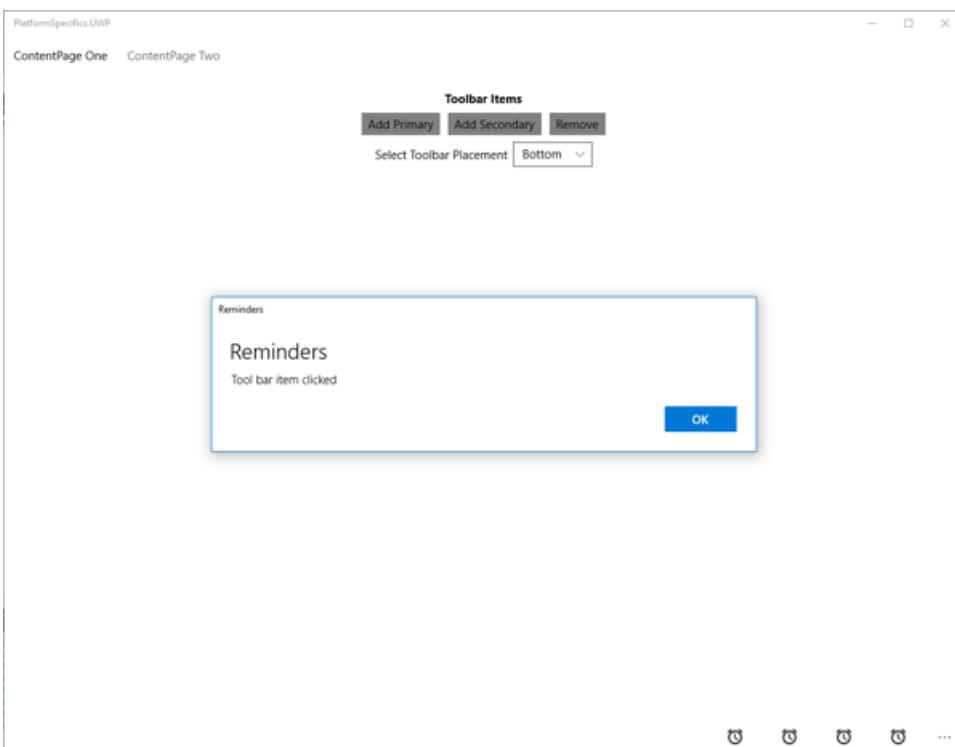
```
<TabbedPage ...  
    xmlns:windows="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"  
    windows:Page.ToolbarPlacement="Bottom">  
    ...  
</TabbedPage>
```

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:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

Setup Windows Projects

8/4/2022 • 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

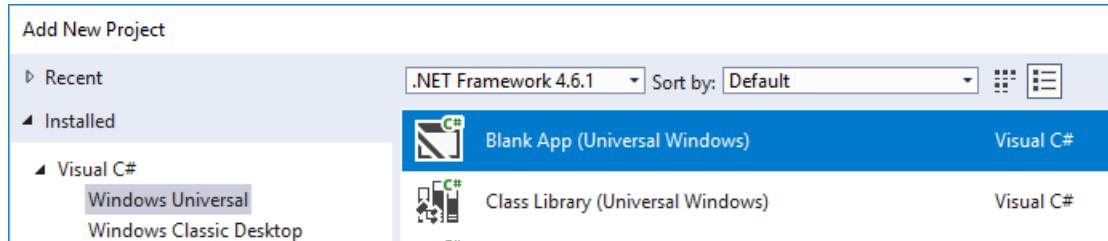
Visual Studio 2019 on Windows 10 is recommended to build UWP apps. For more information about the Universal Windows Platform, see [Intro to the Universal Windows Platform](#).

UWP is available in Xamarin.Forms 2.1 and later, and Xamarin.Forms.Maps is supported in Xamarin.Forms 2.2 and later.

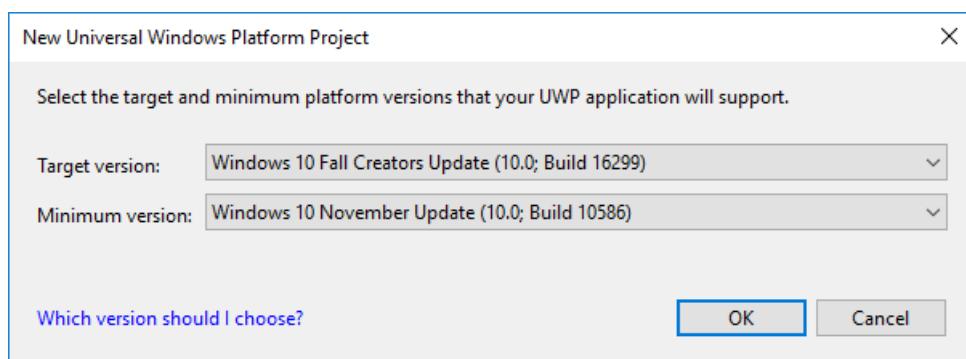
Check the [troubleshooting](#) section for helpful tips.

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:

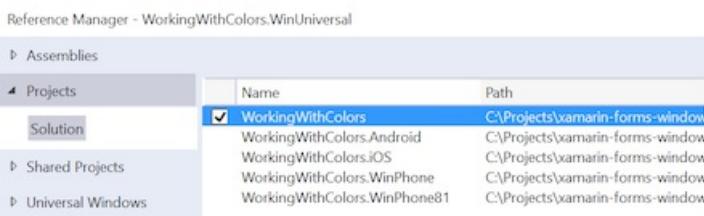


3 . Right-click on the UWP project and select **Manage NuGet Packages...** and add the **Xamarin.Forms** package. Ensure the other projects in the solution are also updated to the same version of the Xamarin.Forms package.

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	✓	
BugSweeper.Android	Debug	Any CPU	✓	✓
BugSweeper.iOS	Debug	iPhone		
BugSweeper.WinApp	Debug	Any CPU	✓	✓
BugSweeper.WinPhone	Debug	Any CPU	✓	✓
BugSweeper.WinPhone81	Debug	Any CPU	✓	✓
BugSweeper.WinUniversal	Debug	x86	✓	✓

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());
```

NOTE

The argument to the `LoadApplication` method is the `Xamarin.Forms.Application` instance defined in your .NET standard project.

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
```

RefreshView Pull Direction on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Universal Windows Platform platform-specific enables the pull direction of a `RefreshView` to be changed to match the orientation of the scrollable control that's displaying data. It's consumed in XAML by setting the `RefreshView.RefreshPullDirection` bindable property to a value of the `RefreshPullDirection` enumeration:

```
<ContentPage ...>
    xmlns:windows="clr-namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
        <RefreshView windows:RefreshView.RefreshPullDirection="LeftToRight"
            IsRefreshing="{Binding IsRefreshing}"
            Command="{Binding RefreshCommand}">
            <ScrollView>
                ...
            </ScrollView>
        </RefreshView>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

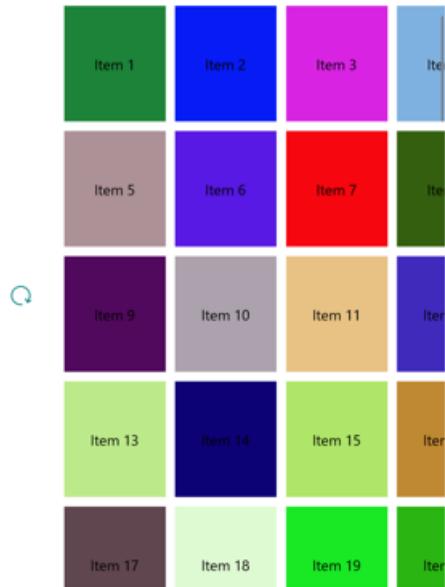
```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...
refreshView.On<Windows>().SetRefreshPullDirection(RefreshPullDirection.LeftToRight);
```

The `RefreshView.On<Windows>` method specifies that this platform-specific will only run on the Universal Windows Platform. The `RefreshView.SetRefreshPullDirection` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to set the pull direction of the `RefreshView`, with the `RefreshPullDirection` enumeration providing four possible values:

- `LeftToRight` indicates that a pull from left to right initiates a refresh.
- `TopToBottom` indicates that a pull from top to bottom initiates a refresh, and is the default pull direction of a `RefreshView`.
- `RightToLeft` indicates that a pull from right to left initiates a refresh.
- `BottomToTop` indicates that a pull from bottom to top initiates a refresh.

In addition, the `GetRefreshPullDirection` method can be used to return the current `RefreshPullDirection` of the `RefreshView`.

The result is that a specified `RefreshPullDirection` is applied to the `RefreshView`, to set the pull direction to match the orientation of the scrollable control that's displaying data. The following screenshot shows a `RefreshView` with a `LeftToRight` pull direction:



NOTE

When you change the pull direction, the starting position of the progress circle automatically rotates so that the arrow starts in the appropriate position for the pull direction.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

SearchBar Spell Check on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)

 [Download the sample](#)

This Universal Windows Platform 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>
```

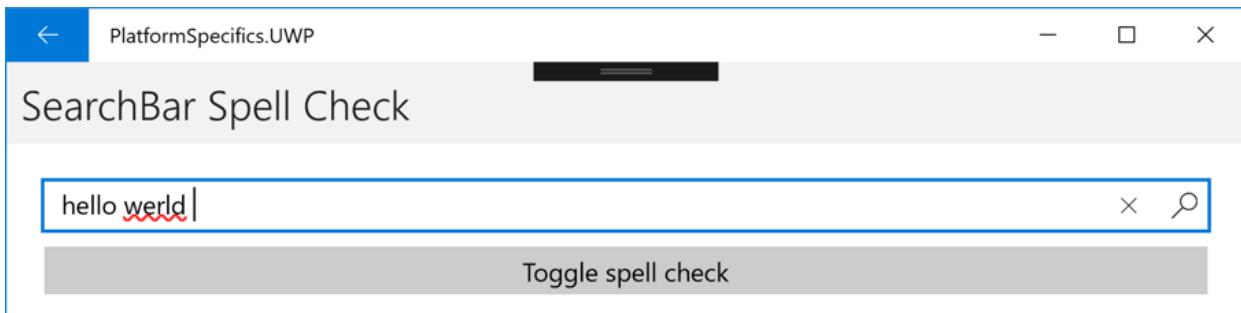
Alternatively, it can be consumed from C# using the fluent API:

```
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.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

TabPage Icons on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This Universal Windows Platform platform-specific enables page icons to be displayed on a `TabPage` toolbar, and provides the ability to optionally specify the icon size. It's consumed in XAML by setting the `TabPage.HeaderIconsEnabled` attached property to `true`, and by optionally setting the `TabPage.HeaderIconsSize` attached property to a `Size` value:

```
<TabPage ...>
    xmlns:windows="clr-namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"
    windows:TabPage.HeaderIconsEnabled="true">
        <windows:TabPage.HeaderIconsSize>
            <Size>
                <x:Arguments>
                    <x:Double>24</x:Double>
                    <x:Double>24</x:Double>
                </x:Arguments>
            </Size>
        </windows:TabPage.HeaderIconsSize>
        <ContentPage Title="Todo" IconImageSource="todo.png">
            ...
        </ContentPage>
        <ContentPage Title="Reminders" IconImageSource="reminders.png">
            ...
        </ContentPage>
        <ContentPage Title="Contacts" IconImageSource="contacts.png">
            ...
        </ContentPage>
    </TabPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

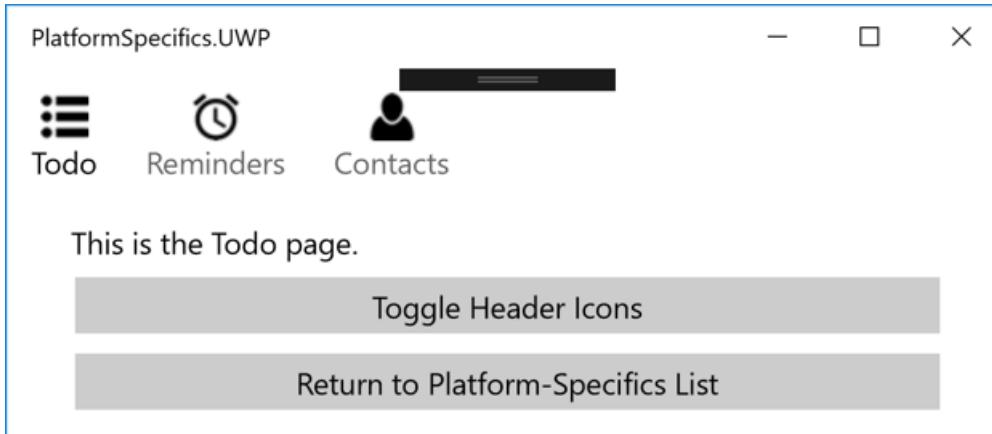
public class WindowsTabPageIconsCS : Xamarin.Forms.TabPage
{
    public WindowsTabPageIconsCS()
    {
        On<Windows>().SetHeaderIconsEnabled(true);
        On<Windows>().SetHeaderIconsSize(new Size(24, 24));

        Children.Add(new ContentPage { Title = "Todo", IconImageSource = "todo.png" });
        Children.Add(new ContentPage { Title = "Reminders", IconImageSource = "reminders.png" });
        Children.Add(new ContentPage { Title = "Contacts", IconImageSource = "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:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

VisualElement Access Keys on Windows

8/4/2022 • 3 minutes to read • [Edit Online](#)



[Download the sample](#)

Access keys are keyboard shortcuts that improve the usability and accessibility of apps on the Universal Windows Platform (UWP) 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 UWP 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 ...
    xmlns:windows="clr-
    namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
    <ContentPage Title="Page 1"
        windows:VisualElement.AccessKey="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.

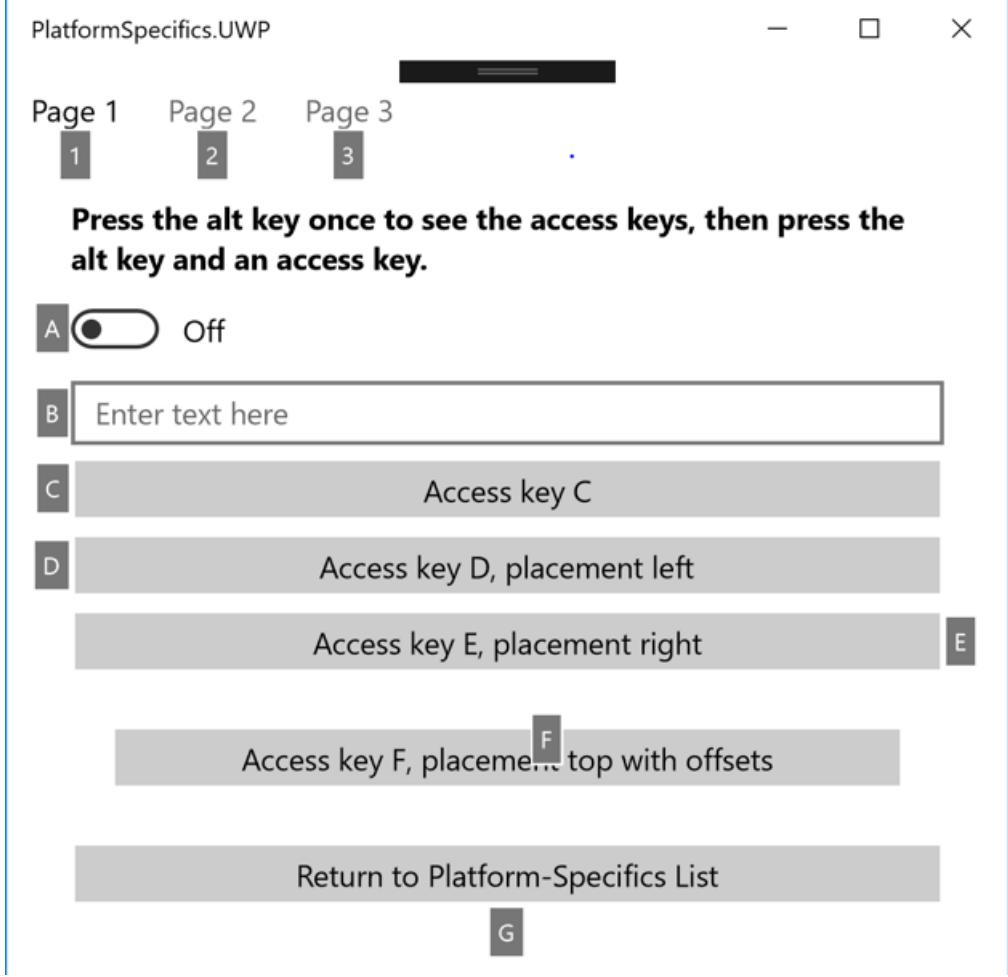
The `visualElement.SetAccessKeyHorizontalOffset` and `visualElement.SetAccessKeyVerticalOffset` methods can be used for more granular control of the access key tip location. The argument to the `SetAccessKeyHorizontalOffset` method indicates how far to move the access key tip left or right, and the argument 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.

WARNING

By default, when a modal dialog is displayed any access keys that are defined on the page behind the dialog can still be activated. However, custom logic can be written to disable access keys in this scenario. This can be achieved by handling the `Dispatcher.AcceleratorKeyActivated` event in the `MainPage` class of your UWP project, and in the event handler setting the `Handled` property of the event arguments to `true` when a modal dialog is displayed.

For more information about access keys, see [Access keys](#).

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

VisualElement Legacy Color Mode on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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 Universal Windows Platform 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:windows="clr-namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
        <StackLayout>
            ...
            <Editor Text="Enter text here"
                    TextColor="Blue"
                    BackgroundColor="Bisque"
                    windows:VisualElement.IsLegacyColorModeEnabled="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 `Enabled` (Currently: False)

The Editor below has the legacy color mode disabled. It will use whatever colors are manually set.

Enter text here

Toggle `Enabled` (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](#).

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

WebView Execution Mode on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

This platform-specific sets the thread on which a `WebView` hosts its content. It's consumed in XAML by setting the `WebView.ExecutionMode` bindable property to a `WebViewExecutionMode` enumeration value:

```
<ContentPage ...  
    xmlns:windows="clr-  
    namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">  
    <StackLayout>  
        <WebView ... windows:WebView.ExecutionMode="SeparateThread" />  
        ...  
    </StackLayout>  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Xamarin.Forms.PlatformConfiguration;  
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;  
...  
  
WebView webView = new Xamarin.Forms.WebView();  
webView.On<Windows>().SetExecutionMode(WebViewExecutionMode.SeparateThread);
```

The `WebView.On<Windows>` method specifies that this platform-specific will only run on the Universal Windows Platform. The `WebView.SetExecutionMode` method, in the `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` namespace, is used to set the thread on which a `WebView` hosts its content, with the `WebViewExecutionMode` enumeration providing three possible values:

- `SameThread` indicates that content is hosted on the UI thread. This is the default value for the `WebView` on Windows.
- `SeparateThread` indicates that content is hosted on a background thread.
- `SeparateProcess` indicates that content is hosted on a separate process off the app process. There isn't a separate process per `WebView` instance, and so all of an app's `WebView` instances share the same separate process.

In addition, the `GetExecutionMode` method can be used to return the current `WebViewExecutionMode` for the `WebView`.

Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

WebView JavaScript Alerts on Windows

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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>
```

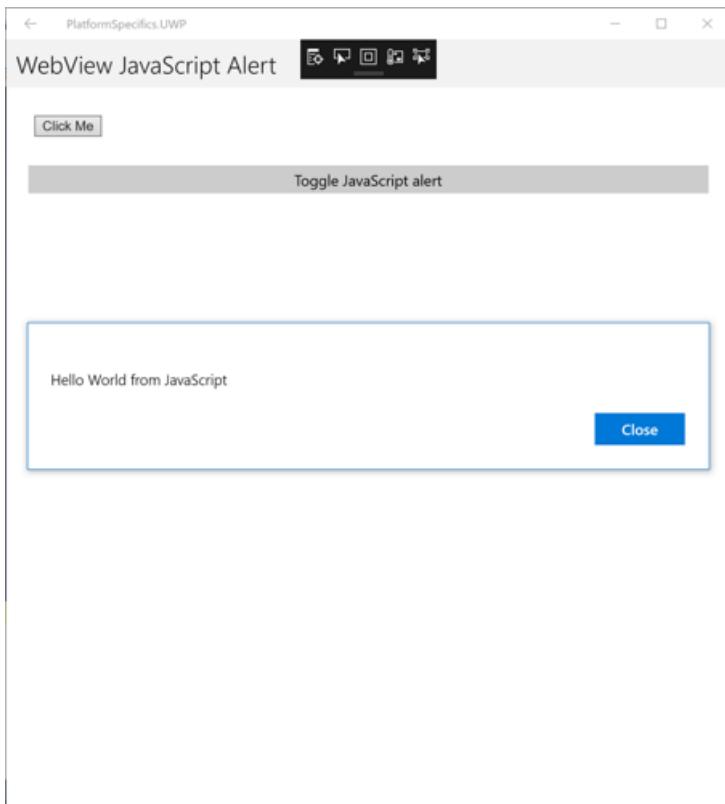
Alternatively, it can be consumed from C# using the fluent API:

```
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:



Related links

- [PlatformSpecifics \(sample\)](#)
- [Creating Platform-Specifics](#)
- [WindowsSpecific API](#)

Platform-Specifics

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

Platform-specifics allow you to consume functionality that's only available on a specific platform, without implementing custom renderers or effects.

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 the platform-specifics provided by Xamarin.Forms, see [iOS Platform-Specifics](#), [Android Platform-Specifics](#), and [Windows Platform-Specifics](#).

Creating platform-specifics

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.

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 the 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);
```

Related links

- [PlatformSpecifics \(sample\)](#)
- [ShadowPlatformSpecific \(sample\)](#)
- [iOS Platform-Specifics](#)
- [Android Platform-Specifics](#)
- [Windows Platform-Specifics](#)
- [Customizing Controls with Effects](#)
- [Attached Properties](#)
- [PlatformConfiguration API](#)

Xamarin.Forms Device Class

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

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 methods that can be used to interact with UI controls from background threads. For more information, see [Interact with the UI from background threads](#).

Provide 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[^]
- **Tablet** – iPad, Windows devices, and Android devices wider than 600 dips[^]
- **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

[^] *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.GetNamedColor

Xamarin.Forms 4.6 introduces support for named colors. A named color is a color that has a different value depending on which system mode (for example, light or dark) is active on the device. On Android, named colors are accessed via the `R.Color` class. On iOS, named colors are called [system colors](#). On the Universal Windows Platform, named colors are called [XAML theme resources](#).

The `GetNamedColor` method can be used to retrieve named colors on Android, iOS, and UWP. The method takes a `string` argument and returns a `Color`:

```
// Retrieve an Android named color
Color color = Device.GetNamedColor(NamedPlatformColor.HoloBlueBright);
```

`Color.Default` will be returned when a color name cannot be found, or when `GetNamedColor` is invoked on an unsupported platform.

NOTE

Because the `GetNamedColor` method returns a `Color` that's specific to a platform, it should typically be used in conjunction with the `Device.RuntimePlatform` property.

The `NamedPlatformColor` class contains the constants that define the named colors for Android, iOS, and UWP:

ANDROID	IOS	MACOS	UWP
BackgroundDark	Label	AlternateSelectedControlText	SystemAltHighColor
BackgroundLight	Link	ControlAccent	SystemAltLowColor
Black	OpaqueSeparator	ControlBackgroundColor	SystemAltMediumColor
DarkerGray	PlaceholderText	ControlColor	SystemAltMediumHighColor
HoloBlueBright	QuaternaryLabel	DisabledControlTextColor	SystemAltMediumLowColor
HoloBlueDark	SecondaryLabel	FindHighlightColor	SystemBaseHighColor
HoloBlueLight	Separator	GridColor	SystemBaseLowColor
HoloGreenDark	SystemBlue	HeaderTextColor	SystemBaseMediumColor
HoloGreenLight	SystemGray	HighlightColor	SystemBaseMediumHighColor
HoloOrangeDark	SystemGray2	KeyboardFocusIndicatorColor	SystemBaseMediumLowColor
HoloOrangeLight	SystemGray3	Label	SystemChromeAltLowColor
HoloPurple	SystemGray4	LabelColor	SystemChromeBlackHighColor
HoloRedDark	SystemGray5	Link	SystemChromeBlackLowColor
HoloRedLight	SystemGray6	LinkColor	SystemChromeBlackMediumColor
TabIndicatorText	SystemGreen	PlaceholderText	SystemChromeBlackMediumLowColor
Transparent	SystemIndigo	PlaceholderTextColor	SystemChromeDisabledHighColor
White	SystemOrange	QuaternaryLabel	SystemChromeDisabledLowColor
WidgetEditTextDark	SystemPink	QuaternaryLabelColor	SystemChromeHighColor

ANDROID	IOS	MACOS	UWP
	SystemPurple	SecondaryLabel	SystemChromeLowColor
	SystemRed	SecondaryLabelColor	SystemChromeMediumColor
	SystemTeal	SelectedContentBackgroundColor	SystemChromeMediumLowColor
	SystemYellow	SelectedControlColor	SystemChromeWhiteColor
	TertiaryLabel	SelectedControlTextColor	SystemListLowColor
		SelectedMenuItemTextColor	SystemListMediumColor
		SelectedTextBackgroundColor	
		SelectedTextColor	
		Separator	
		SeparatorColor	
		ShadowColor	
		SystemBlue	
		SystemGray	
		SystemGreen	
		SystemIndigo	
		SystemOrange	
		SystemPink	
		SystemPurple	
		SystemRed	
		SystemTeal	
		SystemYellow	
		TertiaryLabel	
		TertiaryLabelColor	
		TextBackgroundColor	

ANDROID	IOS	MACOS	UWP
		TextColor	
		UnderPageBackgroundColor	
		UnemphasizedSelectedContentBackgroundColor	
		UnemphasizedSelectedTextBackgroundColor	
		UnemphasizedSelectedTextColor	
		WindowBackgroundColor	
		WindowFrameTextColor	

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).

NOTE

The `System.Timers.Timer` and `System.Threading.Timer` classes are .NET Standard alternatives to using the `Device.StartTimer` method.

Interact with the UI from background threads

Most operating systems, including iOS, Android, and the Universal Windows Platform, use a single-threading model for code involving the user interface. This thread is often called the *main thread* or the *UI thread*. A consequence of this model is that all code that accesses user interface elements must run on the application's main thread.

Applications sometimes use background threads to perform potentially long running operations, such as retrieving data from a web service. If code running on a background thread needs to access user interface elements, it must run that code on the main thread.

The `Device` class includes the following `static` methods that can be used to interact with user interface elements from backgrounds threads:

METHOD	ARGUMENTS	RETURNS	PURPOSE
<code>BeginInvokeOnMainThread</code>	<code>Action</code>	<code>void</code>	Invokes an <code>Action</code> on the main thread, and doesn't wait for it to complete.
<code>InvokeOnMainThreadAsync<T></code>	<code>Func<T></code>	<code>Task<T></code>	Invokes a <code>Func<T></code> on the main thread, and waits for it to complete.
<code>InvokeOnMainThreadAsync</code>	<code>Action</code>	<code>Task</code>	Invokes an <code>Action</code> on the main thread, and waits for it to complete.
<code>InvokeOnMainThreadAsync<T></code>	<code>Func<Task<T>></code>	<code>Task<T></code>	Invokes a <code>Func<Task<T>></code> on the main thread, and waits for it to complete.
<code>InvokeOnMainThreadAsync</code>	<code>Func<Task></code>	<code>Task</code>	Invokes a <code>Func<Task></code> on the main thread, and waits for it to complete.
<code>GetMainThreadSynchronizationContextAsync</code>		<code>Task<SynchronizationContext></code>	Returns the <code>SynchronizationContext</code> for the main thread.

The following code shows an example of using the `BeginInvokeOnMainThread` method:

```
Device.BeginInvokeOnMainThread (() =>
{
    // interact with UI elements
});
```

Related links

- [Device Sample](#)
- [Styles Sample](#)
- [Device API](#)

Xamarin.Forms in Xamarin Native Projects

8/4/2022 • 14 minutes to read • [Edit Online](#)



[Download the sample](#)

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 enables `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;
    AppNavigationController _navigation;

    public static string FolderPath { get; private set; }

    public override bool FinishedLaunching(UIApplication application, NSDictionary launchOptions)
    {
        Forms.Init();

        // Create app-level resource dictionary.
        Xamarin.Forms.Application.Current = new Xamarin.Forms.Application();
        Xamarin.Forms.Application.Current.Resources = new MyDictionary();

        Instance = this;
        _window = new UIWindow(UIScreen.MainScreen.Bounds);

        UINavigationBar.AppearanceSetTitleTextAttributes(new UITextAttributes
        {
            TextColor = UIColor.Black
        });

        FolderPath =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData));

        NotesPage notesPage = new NotesPage()
        {
            // Set the parent so that the app-level resource dictionary can be located.
            Parent = Xamarin.Forms.Application.Current
        };

        UIViewController notesPageController = notesPage.CreateViewController();
        notesPageController.Title = "Notes";

        _navigation = new AppNavigationController(notesPageController);

        _window.RootViewController = _navigation;
        _window.MakeKeyAndVisible();

        notesPage.Parent = null;
        return true;
    }
    // ...
}

```

The `FinishedLaunching` method performs the following tasks:

- Xamarin.Forms is initialized by calling the `Forms.Init` method.
- A new `Xamarin.Forms.Application` object is created, and its application-level resource dictionary is set to a `ResourceDictionary` that's defined in XAML.
- 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 `FolderPath` property is initialized to a path on the device where note data will be stored.
- A `NotesPage` object is created, which is a `Xamarin.Forms ContentPage`-derived page defined in XAML, and its parent is set to the previously created `Xamarin.Forms.Application` object.
- The `NotesPage` object is 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 `AppINavigationController` is created for managing hierarchical navigation. This is a custom navigation controller class, which derives from `UINavigationController`. The `AppINavigationController` object manages a stack of view controllers, and the `UIViewController` passed into the constructor will be presented initially when the `AppINavigationController` is loaded.
- The `AppINavigationController` object 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.
- The `Parent` property of the `NotesPage` object is set to `null`, to prevent a memory leak.

Once the `FinishedLaunching` method has executed, the UI defined in the Xamarin.Forms `NotesPage` class will be displayed, as shown in the following screenshot:



IMPORTANT

All `ContentPage`-derived pages can consume resources defined in the application-level `ResourceDictionary`, provided that the `Parent` property of the page is set to the `Application` object.

Interacting with the UI, for example by tapping on the `+` `Button`, will result in the following event handler in the `NotesPage` code-behind executing:

```
void OnNoteAddedClicked(object sender, EventArgs e)
{
    AppDelegate.Instance.NavigateToNoteEntryPage(new Note());
}
```

The `static AppDelegate.Instance` field enables the `AppDelegate.NavigateToNoteEntryPage` method to be invoked, which is shown in the following code example:

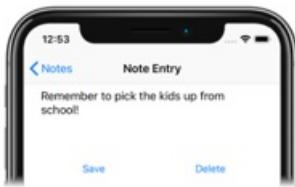
```
public void NavigateToNoteEntryPage(Note note)
{
    NoteEntryPage noteEntryPage = new NoteEntryPage
    {
        BindingContext = note,
        // Set the parent so that the app-level resource dictionary can be located.
        Parent = Xamarin.Forms.Application.Current
    };

    var noteEntryViewController = noteEntryPage.CreateViewController();
    noteEntryViewController.Title = "Note Entry";

    _navigation.PushViewController(noteEntryViewController, true);
    noteEntryPage.Parent = null;
}
```

The `NavigateToNoteEntryPage` 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 `AppNavigationController` by the `PushViewController` method. Therefore, the UI defined in the Xamarin.Forms `NoteEntryPage` class will be

displayed, as shown in the following screenshot:



When the `NoteEntryPage` is displayed, back navigation will pop the `UIViewController` for the `NoteEntryPage` class from the `AppINavigationController`, returning the user to the `UIViewController` for the `NotesPage` class. However, popping a `UIViewController` from the iOS native navigation stack does not automatically dispose of the `UIViewController` and attached `Page` object. Therefore, the `AppINavigationController` class overrides the `PopViewController` method, to dispose of view controllers on backwards navigation:

```
public class AppINavigationController : UINavigationController
{
    //...
    public override UIViewController PopViewController(bool animated)
    {
        UIViewController topView = TopViewController;
        if (topView != null)
        {
            // Dispose of ViewController on back navigation.
            topView.Dispose();
        }
        return base.PopViewController(animated);
    }
}
```

The `PopViewController` override calls the `Dispose` method on the `UIViewController` object that's been popped from the iOS native navigation stack. Failure to do this will result in the `UIViewController` and attached `Page` object being orphaned.

IMPORTANT

Orphaned objects can't be garbage collected, and so result in a memory leak.

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 string FolderPath { get; private set; }

    public static MainActivity Instance;

    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);

        Forms.Init(this, bundle);

        // Create app-level resource dictionary.
        Xamarin.Forms.Application.Current = new Xamarin.Forms.Application();
        Xamarin.Forms.Application.Current.Resources = new MyDictionary();

        Instance = this;

        SetContentView(Resource.Layout.Main);
        var toolbar = FindViewById<Toolbar>(Resource.Id.toolbar);
        SetSupportActionBar(toolbar);
        SupportActionBar.Title = "Notes";

        FolderPath =
Path.Combine(System.Environment.GetFolderPath(System.Environment.SpecialFolder.LocalApplicationData));

        NotesPage notesPage = new NotesPage()
        {
            // Set the parent so that the app-level resource dictionary can be located.
            Parent = Xamarin.Forms.Application.Current
        };
        AndroidX.Fragment.App.Fragment notesPageFragment = notesPage.CreateSupportFragment(this);

        SupportFragmentManager
            .BeginTransaction()
            .Replace(Resource.Id.fragment_frame_layout, mainPage)
            .Commit();
        //...

        notesPage.Parent = null;
    }
    ...
}

```

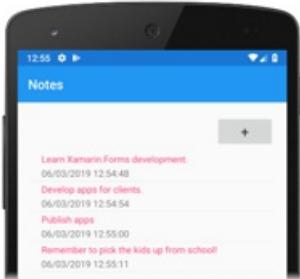
The `OnCreate` method performs the following tasks:

- Xamarin.Forms is initialized by calling the `Forms.Init` method.
- A new `Xamarin.Forms.Application` object is created, and its application-level resource dictionary is set to a `ResourceDictionary` that's defined in XAML.
- 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 `FolderPath` property is initialized to a path on the device where note data will be stored.
- A `NotesPage` object is created, which is a `Xamarin.Forms.ContentPage`-derived page defined in XAML, and its parent is set to the previously created `Xamarin.Forms.Application` object.
- The `NotesPage` object is 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 `NotesPage` class.

- The `Parent` property of the `NotesPage` object is set to `null`, to prevent a memory leak.

For more information about Fragments, see [Fragments](#).

Once the `OnCreate` method has executed, the UI defined in the `Xamarin.Forms NotesPage` class will be displayed, as shown in the following screenshot:



IMPORTANT

All `ContentPage`-derived pages can consume resources defined in the application-level `ResourceDictionary`, provided that the `Parent` property of the page is set to the `Application` object.

Interacting with the UI, for example by tapping on the `+` `Button`, will result in the following event handler in the `NotesPage` code-behind executing:

```
void OnNoteAddedClicked(object sender, EventArgs e)
{
    MainActivity.Instance.NavigateToNoteEntryPage(new Note());
}
```

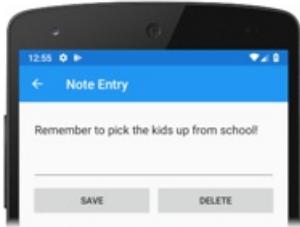
The `static MainActivity.Instance` field enables the `MainActivity.NavigateToNoteEntryPage` method to be invoked, which is shown in the following code example:

```
public void NavigateToNoteEntryPage(Note note)
{
    NoteEntryPage noteEntryPage = new NoteEntryPage
    {
        BindingContext = note,
        // Set the parent so that the app-level resource dictionary can be located.
        Parent = Xamarin.Forms.Application.Current
    };

    AndroidX.Fragment.App.Fragment noteEntryFragment = noteEntryPage.CreateSupportFragment(this);
    SupportFragmentManager
        .BeginTransaction()
        .AddToBackStack(null)
        .Replace(Resource.Id.fragment_frame_layout, noteEntryFragment)
        .Commit();

    noteEntryPage.Parent = null;
}
```

The `NavigateToNoteEntryPage` 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 NoteEntryPage` will be displayed, as shown in the following screenshot:



When the `NoteEntryPage` is displayed, tapping the back arrow will pop the `Fragment` for the `NoteEntryPage` from the fragment back stack, returning the user to the `Fragment` for the `NotesPage` class.

Enable 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 ? "Note Entry" : "Notes";
};
```

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.

Choose 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:

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    // ...
    Xamarin.Forms.Forms.Init(e);

    // Create app-level resource dictionary.
    Xamarin.Forms.Application.Current = new Xamarin.Forms.Application();
    Xamarin.Forms.Application.Current.Resources = new MyDictionary();

    // ...
}
```

In addition, the `OnLaunched` method can also create any application-level resource dictionary that's required by the application.

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
{
    NotesPage notesPage;
    NoteEntryPage noteEntryPage;

    public static MainPage Instance;
    public static string FolderPath { get; private set; }

    public MainPage()
    {
        this.NavigationCacheMode = NavigationCacheMode.Enabled;
        Instance = this;
        FolderPath =
Path.Combine(System.Environment.GetFolderPath(System.Environment.SpecialFolder.LocalApplicationData));

        notesPage = new Notes.UWP.Views.NotesPage
        {
            // Set the parent so that the app-level resource dictionary can be located.
            Parent = Xamarin.Forms.Application.Current
        };
        this.Content = notesPage.CreateFrameworkElement();
        // ...
        notesPage.Parent = null;
    }
    // ...
}
```

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 `FolderPath` property is initialized to a path on the device where note data will be stored.
- A `NotesPage` object is created, which is a `Xamarin.Forms ContentPage`-derived page defined in XAML, and its parent is set to the previously created `Xamarin.Forms.Application` object.
- The `NotesPage` object is converted to a `FrameworkElement` using the `CreateFrameworkElement` extension method, and then set as the content of the `MainPage` class.
- The `Parent` property of the `NotesPage` object is set to `null`, to prevent a memory leak.

Once the `MainPage` constructor has executed, the UI defined in the `Xamarin.Forms NotesPage` class will be displayed, as shown in the following screenshot:



IMPORTANT

All `ContentPage`-derived pages can consume resources defined in the application-level `ResourceDictionary`, provided that the `Parent` property of the page is set to the `Application` object.

Interacting with the UI, for example by tapping on the + `Button`, will result in the following event handler in the `NotesPage` code-behind executing:

```
void OnNoteAddedClicked(object sender, EventArgs e)
{
    MainPage.Instance.NavigateToNoteEntryPage(new Note());
}
```

The `static MainPage.Instance` field enables the `MainPage.NavigateToNoteEntryPage` method to be invoked, which is shown in the following code example:

```
public void NavigateToNoteEntryPage(Note note)
{
    noteEntryPage = new Notes.UWP.Views.NoteEntryPage
    {
        BindingContext = note,
        // Set the parent so that the app-level resource dictionary can be located.
        Parent = Xamarin.Forms.Application.Current
    };
    this.Frame.Navigate(noteEntryPage);
    noteEntryPage.Parent = null;
}
```

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 `NavigateToNoteEntryPage` method executes, the UI defined in the Xamarin.Forms `NoteEntryPage` will be displayed, as shown in the following screenshot:



When the `NoteEntryPage` is displayed, tapping the back arrow will pop the `FrameworkElement` for the `NoteEntryPage` from the in-app back stack, returning the user to the `FrameworkElement` for the `NotesPage` class.

Enable page resizing support

When the UWP application window is resized, the Xamarin.Forms content should also be resized. This is accomplished by registering an event handler for the `Loaded` event, in the `MainPage` constructor:

```
public MainPage()
{
    // ...
    this.Loaded += On MainPageLoaded;
    // ...
}
```

The `Loaded` event fires when the page is laid out, rendered, and ready for interaction, and executes the `On MainPageLoaded` method in response:

```
void On MainPageLoaded(object sender, RoutedEventArgs e)
{
    this.Frame.SizeChanged += (o, args) =>
    {
        if (noteEntryPage != null)
            noteEntryPage.Layout(new Xamarin.Forms.Rectangle(0, 0, args.NewSize.Width,
args.NewSize.Height));
        else
            notesPage.Layout(new Xamarin.Forms.Rectangle(0, 0, args.NewSize.Width, args.NewSize.Height));
    };
}
```

The `On MainPageLoaded` method registers an anonymous event handler for the `Frame.SizeChanged` event, which is raised when either the `ActualHeight` or the `ActualWidth` properties change on the `Frame`. In response, the Xamarin.Forms content for the active page is resized by calling the `Layout` method.

Enable 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 `MainPage` constructor:

```
public MainPage()
{
    // ...
    SystemNavigationManager.GetForCurrentView().BackRequested += On BackRequested;
}
```

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();
        noteEntryPage = null;
    }
}
```

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 event being ignored.

The application chooses whether to show a back button on the title bar. This is achieved by setting the `AppView backButtonVisibility` property to one of the `AppView backButtonVisibility` enumeration values, in the `App` class:

```
void OnNavigated(object sender, NavigationEventArgs e)
{
    SystemNavigationManager.GetForCurrentView().AppViewBackButtonVisibility =
        ((Frame)sender).CanGoBack ? AppViewBackButtonVisibility.Visible :
    AppViewBackButtonVisibility.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](#).

Related links

- [NativeForms \(sample\)](#)
- [Native Views](#)

Native Views in Xamarin.Forms

8/4/2022 • 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

8/4/2022 • 12 minutes to read • [Edit Online](#)

 [Download the sample](#)

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.

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.

IMPORTANT

Compiled XAML must be disabled for any XAML pages that use native views. This can be accomplished by decorating the code-behind class for your XAML page with the `[XamlCompilation(XamlCompilationOptions.Skip)]` attribute. For more information about XAML compilation, see [XAML Compilation in Xamarin.Forms](#).

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 [Refer to native views from code](#).

Consume 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 `iOS`, `Android`, `UWP`, `Windows` (which is equivalent to `UWP`), `macOS`, `GTK`, `Tizen`, or `WPF`. 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 [Pass 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 [Refer to 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:

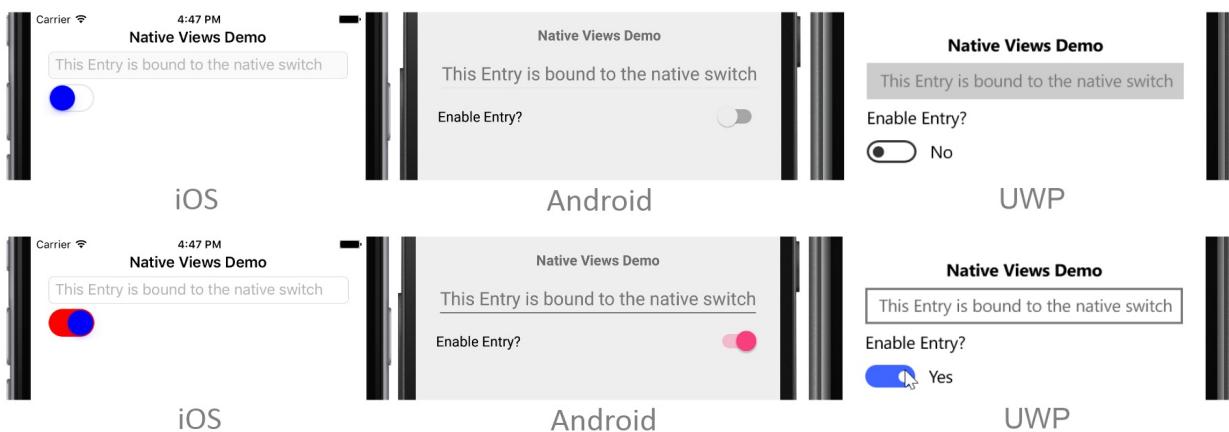
```

<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.

Pass 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](#).

Refer 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 NativeViewInsideContentPage : ContentPage
{
    public NativeViewInsideContentPage()
    {
        InitializeComponent();

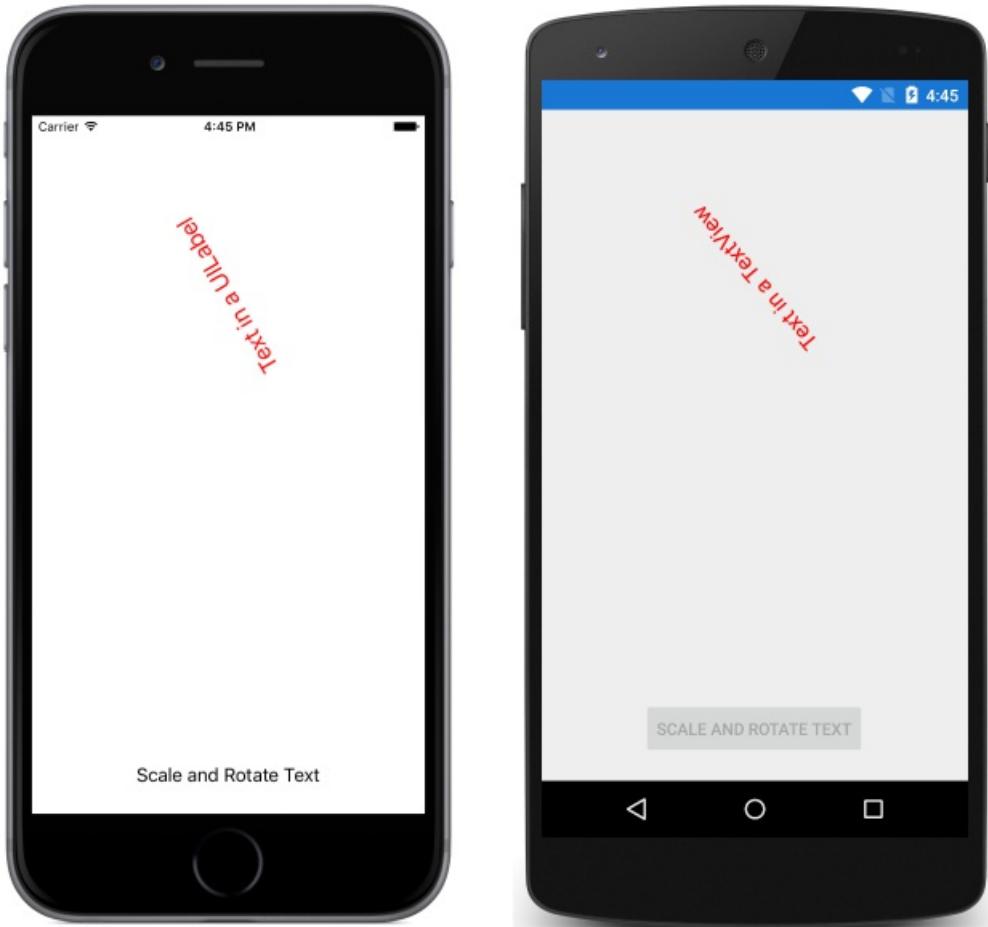
#if __IOS__
        var wrapper = (Xamarin.Forms.Platform.iOS.NativeViewWrapper)contentViewButtonParent.Content;
        var button = (UIKit.UITextField)wrapper.NativeView;
        button.SetTitle("Scale and Rotate Text", UIKit.UIControlState.Normal);
        button.SetTitleColor(UIKit.UIColor.Black, UIKit.UIControlState.Normal);
#endif
#if __ANDROID__
        var wrapper = (Xamarin.Forms.Platform.Android.NativeViewWrapper)contentViewTextParent.Content;
        var textView = (Android.Widget.TextView)wrapper.NativeView;
        textView.SetTextColor(Android.Graphics.Color.Red);
#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:



Subclass 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:

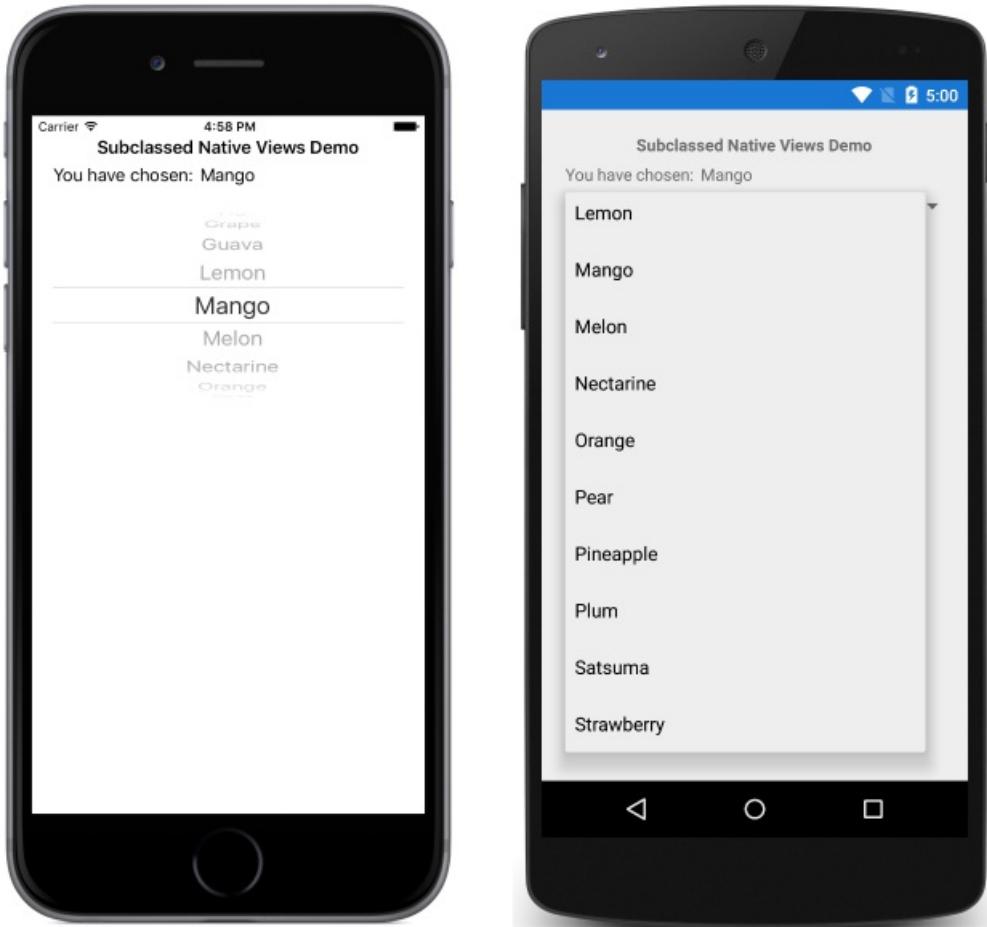
```

<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 `ItemSource` 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.

Related links

- [NativeSwitch \(sample\)](#)

- [Forms2Native \(sample\)](#)
- [NativeViewInsideContentView \(sample\)](#)
- [SubclassedNativeControls \(sample\)](#)
- [Native Forms](#)
- [Passing Arguments in XAML](#)

Native Views in C#

8/4/2022 • 7 minutes to read • [Edit Online](#)

 [Download the sample](#)

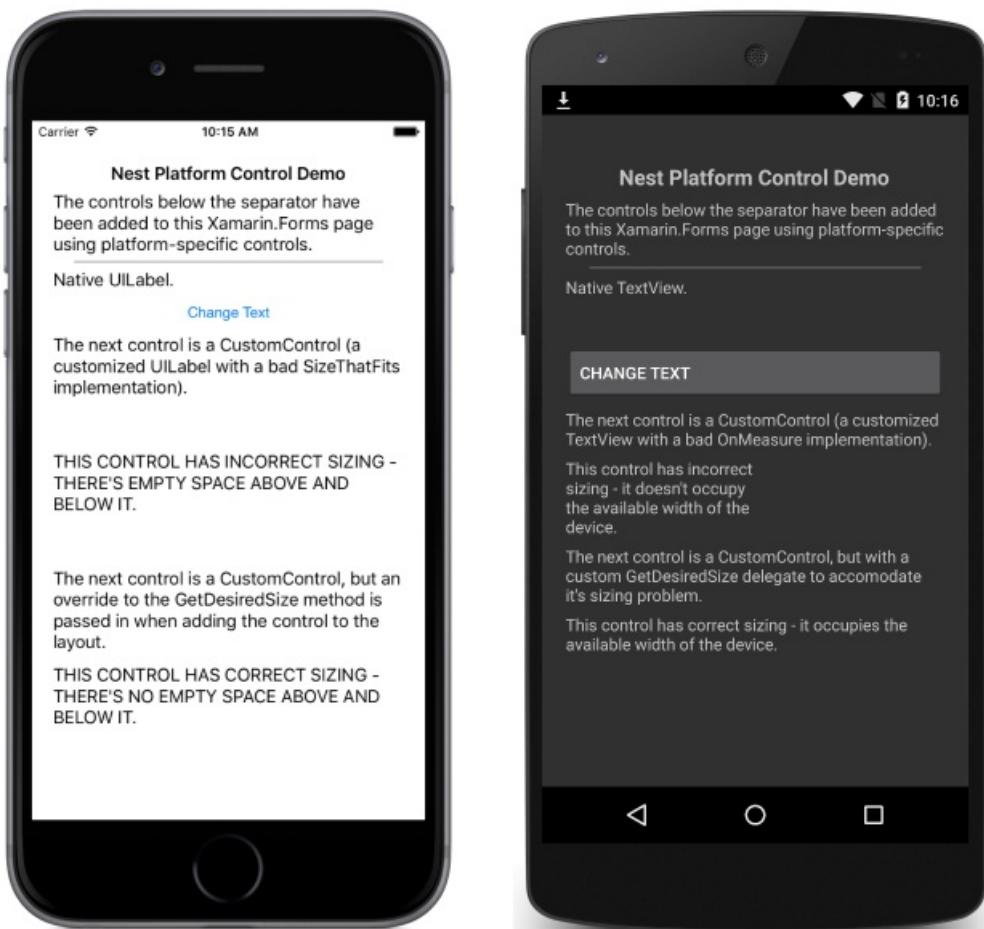
Native views from iOS, Android, and UWP can be directly referenced from Xamarin.Forms pages created using C#. This article demonstrates how to add native views to a Xamarin.Forms layout created using C#, and how to override the layout of custom views to correct their measurement API usage.

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("HelveticaNeue"),
    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](#)

Sign In with Apple in Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)

Sign In with Apple introduces a new service providing identity protection for users. As of iOS 13 Apple requires any apps using third-party authentication providers to also offer Sign In with Apple. For instructions on using this feature with Xamarin.iOS, [read more here](#).

When supporting Sign In with Apple within a Xamarin.Forms solution, there are additional considerations to account for Android and UWP. For those platforms, Apple provides a different workflow.

Setup for Xamarin.iOS

This guide walks through the setup necessary to enable Sign in with Apple for Xamarin.iOS applications.

Setup for other platforms

This guide walks through the setup necessary to enable Sign in with Apple for other platforms, including Xamarin.Forms Android and UWP.

Use Sign In with Apple in Xamarin.Forms

With a few services you can support Sign In with Apple in your cross-platform Xamarin.Forms applications. This guide describes the necessary steps.

Sign In with Apple in Xamarin.iOS

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

Sign In with Apple is a new service that provides identity protection for users of third-party authentication services. Beginning with iOS 13, Apple requires that any new app using a third-party authentication services should also provide Sign In with Apple. Existing apps being updated do not need to add Sign In with Apple until April 2020.

This document introduces how you can add Sign In with Apple to iOS 13 applications.

Apple developer setup

Before building and running an app using Sign In with Apple, you need to complete these steps. On [Apple Developer Certificates, Identifiers & Profiles](#) portal:

1. Create a new **App IDs Identifier**.
2. Set a description in the **Description** field.
3. Choose an **Explicit** Bundle ID and set `com.xamarin.AddingTheSignInWithAppleFlowToYourApp` in the field.
4. Enable **Sign In with Apple** capability and register the new Identity.
5. Create a new Provisioning Profile with the new Identity.
6. Download and install it on your device.
7. In Visual Studio, enable the **Sign In with Apple** capability in **Entitlements.plist** file.

Check sign in status

When your app begins, or when you first need to check the authentication status of a user, instantiate an `ASAAuthorizationAppleIdProvider` and check the current state:

```

var appleIdProvider = new ASAAuthorizationAppleIdProvider ();
appleIdProvider.GetCredentialState (KeychainItem.CurrentUserIdentity, (credentialState, error) => {
    switch (credentialState) {
        case ASAAuthorizationAppleIdProviderCredentialState.Authorized:
            // The Apple ID credential is valid.
            break;
        case ASAAuthorizationAppleIdProviderCredentialState.Revoked:
            // The Apple ID credential is revoked.
            break;
        case ASAAuthorizationAppleIdProviderCredentialState.NotFound:
            // No credential was found, so show the sign-in UI.
            InvokeOnMainThread () => {
                var storyboard = UIStoryboard.FromName ("Main", null);

                if (!(storyboard.InstantiateViewController (nameof (LoginViewController)) is LoginViewController
viewController))
                    return;

                viewController.ModalPresentationStyle = UIModalPresentationStyle.FormSheet;
                viewController.ModalInPresentation = true;
                Window?.RootViewController?.PresentViewController (viewController, true, null);
            });
            break;
    }
});

```

In this code, called during `FinishedLaunching` in the `AppDelegate.cs`, the app will handle when a state is `NotFound` and present the `LoginViewController` to the user. If the state had return `Authorized` or `Revoked`, a different action may be presented to the user.

A LoginViewController for Sign In with Apple

The `UIViewController` that implements login logic and offers Sign In with Apple needs to implement `IASAuthorizationControllerDelegate` and `IASAuthorizationControllerPresentationContextProviding` as in the `LoginViewController` example below.

```

public partial class LoginViewController : UIViewController, IASAuthorizationControllerDelegate,
IASAuthorizationControllerPresentationContextProviding {
    public LoginViewController (IntPtr handle) : base (handle)
    {
    }

    public override void ViewDidLoad ()
    {
        base.ViewDidLoad ();
        // Perform any additional setup after loading the view, typically from a nib.

        SetupProviderLoginView ();
    }

    public override void ViewDidAppear (bool animated)
    {
        base.ViewDidAppear (animated);

        PerformExistingAccountSetupFlows ();
    }

    void SetupProviderLoginView ()
    {
        var authorizationButton = new ASAAuthorizationAppleIdButton
(ASAAuthorizationAppleIdButtonType.Default, ASAAuthorizationAppleIdButtonStyle.White);
        authorizationButton.TouchUpInside += HandleAuthorizationAppleIDButtonPress;
        loginProviderStackView.AddArrangedSubview (authorizationButton);
    }

    // Prompts the user if an existing iCloud Keychain credential or Apple ID credential is found.
    void PerformExistingAccountSetupFlows ()
    {
        // Prepare requests for both Apple ID and password providers.
        ASAAuthorizationRequest [] requests = {
            new ASAAuthorizationAppleIdProvider ().CreateRequest (),
            new ASAAuthorizationPasswordProvider ().CreateRequest ()
        };

        // Create an authorization controller with the given requests.
        var authorizationController = new ASAAuthorizationController (requests);
        authorizationController.Delegate = this;
        authorizationController.PresentationContextProvider = this;
        authorizationController.PerformRequests ();
    }

    private void HandleAuthorizationAppleIDButtonPress (object sender, EventArgs e)
    {
        var appleIdProvider = new ASAAuthorizationAppleIdProvider ();
        var request = appleIdProvider.CreateRequest ();
        request.RequestedScopes = new [] { ASAAuthorizationScope.Email, ASAAuthorizationScope.FullName };

        var authorizationController = new ASAAuthorizationController (new [] { request });
        authorizationController.Delegate = this;
        authorizationController.PresentationContextProvider = this;
        authorizationController.PerformRequests ();
    }
}

```



This example code checks the current login status in `PerformExistingAccountSetupFlows` and connects to the current view as a delegate. If an existing iCloud Keychain credential or Apple ID credential is found, the user will be prompted to use that.

Apple provides `ASAuthorizationAppleIDButton`, a button specifically for this purpose. When touched, the button will trigger the workflow handled in the method `HandleAuthorizationAppleIDButtonPress`.

Handling authorization

In the `IASAuthorizationController` implement any custom logic to store the user's account. The example below stores the user's account in Keychain, Apple's own storage service.

```

#region IASAuthorizationController Delegate

[Export ("authorizationController:didCompleteWithAuthorization:")]
public void DidComplete (ASAuthorizationController controller, ASAuthorization authorization)
{
    if (authorization.GetCredential<ASAuthorizationAppleIdCredential> () is ASAuthorizationAppleIdCredential
appleIdCredential) {
        var userIdentifier = appleIdCredential.User;
        var fullName = appleIdCredential.FullName;
        var email = appleIdCredential.Email;

        // Create an account in your system.
        // For the purpose of this demo app, store the userIdentifier in the keychain.
        try {
            new KeychainItem ("com.example.apple-samplecode.juice", "userIdentifier").SaveItem
(userIdentifier);
        } catch (Exception) {
            Console.WriteLine ("Unable to save userIdentifier to keychain.");
        }
    }

    // For the purpose of this demo app, show the Apple ID credential information in the
ResultViewController.
    if (!(PresentingViewController is ResultViewController viewController))
        return;

    InvokeOnMainThread (() => {
        viewController.IdentifierText = userIdentifier;
        viewController.GivenNameText = fullName?.GivenName ?? "";
        viewController.FamilyNameText = fullName?.FamilyName ?? "";
        viewController.EmailText = email ?? "";

        DismissViewController (true, null);
    });
} else if (authorization.GetCredential<ASPasswordCredential> () is ASPasswordCredential
passwordCredential) {
    // Sign in using an existing iCloud Keychain credential.
    var username = passwordCredential.User;
    var password = passwordCredential.Password;

    // For the purpose of this demo app, show the password credential as an alert.
    InvokeOnMainThread (() => {
        var message = $"The app has received your selected credential from the keychain. \n\n Username:
{username}\n Password: {password}";
        var alertController = UIAlertController.Create ("Keychain Credential Received", message,
UIAlertControllerStyle.Alert);
        alertController.AddAction (UIAlertAction.Create ("Dismiss", UIAlertActionStyle.Cancel, null));

        PresentViewController (alertController, true, null);
    });
}
}

[Export ("authorizationController:didCompleteWithError:")]
public void DidComplete (ASAuthorizationController controller, NSError error)
{
    Console.WriteLine (error);
}

#endregion

```

Authorization Controller

The final piece in this implementation is the `ASAuthorizationController` which manages authorization requests for the provider.

```
#region IASAuthorizationControllerPresentation Context Providing

public UIWindow GetPresentationAnchor (ASAuthorizationController controller) => View.Window;

#endregion
```

Related links

- [Sign In with Apple Guidelines](#)
- [Sign In with Apple Entitlement](#).
- [WWDC 2019 session 706: Introducing Sign In with Apple.](#)
- [Setup Sign In with Apple for Xamarin.Forms](#)

Setup Sign In with Apple for Xamarin.Forms

8/4/2022 • 3 minutes to read • [Edit Online](#)

This guide covers the series of steps needed to setup your cross-platform applications to take advantage of Sign In with Apple. While the Apple setup is straight forward in the Apple Developer Portal, additional steps are necessary to create a secure relationship between your Android and Apple.

Apple developer setup

Before you can use Sign In with Apple in your applications, you'll need to address some setup steps in the [Certificates, Identifiers & Profiles](#) section of Apple's Developer Portal.

Apple sign in domain

Register your domain name and verify it with Apple in the [More](#) section of the *Certificates, Identifiers & Profiles* section.

Configure Sign In with Apple

Register Domains and Emails for communication

In order to contact users that use Apple's private email relay service, you need to register domains and email addresses that your organization will use for communication. Domains and domains associated with email addresses must comply with Sender Policy Framework standards and be verified by Apple before they are successfully registered.

Domains and Associated Email Addresses

All email addresses associated with your registered domains will be able to send and receive messages to and from customers using Apple's private email relay service. To verify ownership of your domain, select Download to get the verification file, upload it at the URL listed below, and click Verify.

<https://example.com/.well-known/apple-developer-domain-association.txt>

<input checked="" type="checkbox"/> anotherapp.com	Remove
myapp.com	Register

Add your domain and click Register.

Certificates, Identifiers & Profiles

Certificates

Identifiers

Devices

Profiles

Keys

More

Sign In with Apple

Allow users to set up an account and sign in to your apps and associated websites with the Apple IDs they already have. Configuration is required to communicate with your users and receive updates from Apple.

[Configure](#)

NOTE

If you see an error about your domain not being SPF Compliant, you will need to add a SPF DNS TXT Record to your domain and wait for it to propagate before continuing: The SPF TXT may look something like this:

```
v=spf1 a a:myapp.com -all
```

Next you will need to verify ownership of the domain by clicking [Download](#) to retrieve the `.apple-developer-domain-association.txt` file, and upload it to the `.well-known` folder of your domain's website.

Once the `.well-known/apple-developer-domain-association.txt` file is uploaded, and reachable, you can click [Verify](#) to have Apple verify your domain ownership.

NOTE

Apple will verify ownership with `https://`. Ensure you have SSL setup and the file is accessible through a secure URL.

Successfully complete this process before continuing.

Setup your App ID

In the **Identifiers** section, create a new identifier, and choose **App IDs**. If you already have an App ID, choose to edit it instead.

Register a New Identifier

[Continue](#)

App IDs

Register an App ID to enable your app to access available services and identify your app in a provisioning profile. You can enable app services when you create an App ID or modify these settings later.

Enable **Sign In with Apple**. You will most likely want to use the **Enable as primary App ID** option.



Sign In with Apple

[Edit](#)

Enable as a primary App ID

Save your App ID changes.

Create a Service ID

In the **Identifiers** section, create a new identifier, and choose **Service IDs**.

Register a New Identifier

[Continue](#)

App IDs

Register an App ID to enable your app to access available services and identify your app in a provisioning profile. You can enable app services when you create an App ID or modify these settings later.

Services IDs

For each website that uses Sign In with Apple, register a services identifier (Services ID), configure your domain and return URL, and create an associated private key.

Give your Services ID a description, and an identifier. This identifier will be your `ServerId`. Make sure to enable **Sign In with Apple**.

Before continuing, click **Configure** next to the *Sign In with Apple* option you enabled.

In the configuration panel, ensure the correct **Primary App ID** is selected.

Next, choose the **Web Domain** you configured previously.

Finally, add one or more **Return URLs**. Any `redirect_uri` you use later must be registered here exactly as you use it. Make sure you include the `http://` or `https://` in the URL when you enter it.

NOTE

For testing purposes, you cannot use `127.0.0.1` or `localhost`, but you can use other domains such as `local.test`. If you choose to do this, you can edit your machine's `hosts` file to resolve this fictitious domain to your local IP address.

Web Authentication Configuration

Use Sign In with Apple to let your users sign in to your app's accompanying website with their Apple ID. To configure web authentication, group your website with the existing primary App ID that's enabled for Sign In with Apple.

Primary App ID

My App (85HMA3YHJX.com. myapp .app)

Domains

Provide your web domain and return URLs to redirect users after successfully signing in to your website. After registering your Services ID, you'll need to verify your web domain. Return to this page to continue the configuration process.

Web Domain

To verify ownership of your domain, you'll need to download the verification file, upload it at the URL listed below, and click Verify. The download button will only appear if you've registered your Services ID and are the Account Holder or an Admin on your development team.

myapp.com

Return URLs

https://myapp.com/api/applesignin_callback

[Remove](#)

http://local.test:7071/api/applesignin_callback

[Add](#)

[Cancel](#)

[Save](#)

Copyright © 2019 Apple Inc. All rights reserved. [Terms of Use](#) | [Privacy Policy](#)

Save your changes when finished.

Create a key for your Services ID

In the [Keys](#) section, create a new [Key](#).

Give your key a name, and enable [Sign In with Apple](#).

Register a New Key

Continue

Key Name

My App Sign In Key

You cannot use special characters such as @, &, *, ', "

ENABLE	NAME	SERVICE	
<input type="checkbox"/>	Apple Push Notifications service (APNs)	Establish connectivity between your notification server and the Apple Push Notification service. One key is used for all of your apps. Learn more	
<input type="checkbox"/>	DeviceCheck	Access per-device, per-developer data that your associated server can use in its business logic. One key is used for all of your apps. Learn more	
<input type="checkbox"/>	MapKit JS	Use Apple Maps on your websites. Show a map, display search results, provide directions, and more. Learn more	<button>Configure</button>
<input type="checkbox"/>	MusicKit	Access the Apple Music catalog and make personalized requests for authorized users. Learn more	<button>Configure</button>
<input checked="" type="checkbox"/>	Sign In with Apple	Enable your apps to allow users to authenticate in your application with their Apple ID. Configuration is required to enable this feature.	<button>Configure</button>

Click **Configure** beside *Sign In with Apple*.

Ensure the correct **Primary App ID** is selected and click **Save**.

Click **Continue** and then **Register** to create your new key.

Next, you will only have one chance to download the key you just generated. Click **Download**.

Download Your Key

[Download](#)

[Done](#)



After downloading your key, it cannot be re-downloaded as the server copy is removed. If you are not prepared to download your key at this time, click **Done** and download it at a later time. Be sure to save a backup of your key in a secure place.

Name: My App Sign In Key

Key ID: 9B2VXC6589

Services: Sign In with Apple

Also, take note of your **Key ID** at this step. This will be used for your `KeyId` later on.

You will have downloaded a `.p8` key file. You can open this file in Notepad, or VSCode to see the text contents. They should look something like:

```
-----BEGIN PRIVATE KEY-----
MIGTAgEAMBMGBasGSM49AgGFCCqGSM49AwEHBHkwdwIBAQQg3MX8n6VnQ2WzgEy0
Skoz9uOvatLMKTU1PyPC Ae jzzUCgCgYIKoZIzj0DAQehRANCAARZ0DoM6QPqpJxP
JKS1Wz0AohFhYre10EXPkjrih4jTm+b0AeG2BGuoIWd18i8FimGDgK6IzHHPsEqj
DHF5Svq0
-----END PRIVATE KEY-----
```

Name this key `P8FileContents` and keep it in a safe place. You will use it when integrating this service into your mobile application.

Summary

This article described the steps necessary to setup Sign In with Apple for use in your Xamarin.Forms applications.

Related links

- [Sign In with Apple Guidelines](#)

Use Sign In with Apple in Xamarin.Forms

8/4/2022 • 6 minutes to read • [Edit Online](#)



[Download the sample](#)

Sign In with Apple is for all new applications on iOS 13 that use third-party authentication services. The implementation details between iOS and Android are quite different. This guide walks through how you can do this today in Xamarin.Forms.

In this guide and sample, specific platform services are used to handle Sign In with Apple:

- Android using a generic web service talking to Azure Functions with OpenID/OpenAuth
- iOS uses the native API for authentication on iOS 13, and falls back to a generic web service for iOS 12 and below

A sample Apple sign in flow

This sample offers an opinionated implementation for getting Apple Sign In to work in your Xamarin.Forms app.

We use two Azure Functions to help with the authentication flow:

1. `applesignin_auth` - Generates the Apple Sign In Authorization URL and redirects to it. We do this on the server side, instead of the mobile app, so we can cache the `state` and validate it when Apple's servers send a callback.
2. `applesignin_callback` - Handles the POST callback from Apple and securely exchanges the authorization code for an Access Token and ID Token. Finally, it redirects back to the App's URI Scheme, passing back the tokens in a URL Fragment.

The mobile app registers itself to handle the custom URI scheme we have selected (in this case `xamarinformsapplesignin://`) so the `applesignin_callback` function can relay the tokens back to it.

When the user starts authentication, the following steps happen:

1. The mobile app generates a `nonce` and `state` value and passes them to the `applesignin_auth` Azure function.
2. The `applesignin_auth` Azure function generates an Apple Sign In Authorization URL (using the provided `state` and `nonce`), and redirects the mobile app browser to it.
3. The user enters their credentials securely in the Apple Sign In authorization page hosted on Apple's servers.
4. After the Apple Sign In flow finishes on Apple's servers, Apple Redirects to the `redirect_uri` which will be the `applesignin_callback` Azure function.
5. The request from Apple sent to the `applesignin_callback` function is validated to ensure the correct `state` is returned, and that the ID Token claims are valid.
6. The `applesignin_callback` Azure function exchanges the `code` posted to it by Apple, for an *Access Token*, *Refresh Token*, and *ID Token* (which contains claims about the User ID, Name, and Email).
7. The `applesignin_callback` Azure function finally redirects back to the app's URI scheme (`xamarinformsapplesignin://`) appending a URI fragment with the Tokens (e.g. `xamarinformsapplesignin://#access_token=...&refresh_token=...&id_token=...`).
8. The Mobile app parses out the URI Fragment into an `AppleAccount` and validates the `nonce` claim received matches the `nonce` generated at the start of the flow.
9. The mobile app is now authenticated!

Azure Functions

This sample uses Azure Functions. Alternatively, an ASP.NET Core Controller or similar web server solution could deliver the same functionality.

Configuration

Several app settings need to be configured when using Azure Functions:

- `APPLE_SIGNIN_KEY_ID` - This is your `KeyId` from earlier.
- `APPLE_SIGNIN_TEAM_ID` - This is usually your *Team ID* found in your [Membership Profile](#)
- `APPLE_SIGNIN_SERVER_ID`: This is the `ServerId` from earlier. It's *not* your App *Bundle ID*, but rather the *Identifier* of the *Services ID* you created.
- `APPLE_SIGNIN_APP_CALLBACK_URI` - This is the custom URI Scheme you want to redirect back to your app with. In this sample `xamarinformsapplesignin://` is used.
- `APPLE_SIGNIN_REDIRECT_URI` - The *Redirect URL* you setup when creating your *Services ID* in the *Apple Sign In Configuration* section. To test, it might look something like: `http://local.test:7071/api/applesignin_callback`
- `APPLE_SIGNIN_P8_KEY` - The text contents of your `.p8` file, with all the `\n` newlines removed so it's one long string

Security considerations

Never store your P8 key inside of your application code. Application code is easy to download and disassemble.

It is also considered a bad practice to use a `WebView` to host the authentication flow, and to intercept URL Navigation events to obtain the authorization code. At this time there is currently no fully secure way to handle Sign In with Apple on non iOS13+ devices without hosting some code on a server to handle the token exchange. We recommend hosting the authorization url generation code on a server so you can cache the state and validate it when Apple issues a POST callback to your server.

A cross-platform sign in service

Using the Xamarin.Forms DependencyService, you can create separate authentication services that use the platform services on iOS, and a generic web service for Android and other non-iOS platforms based on a shared interface.

```
public interface IAppleSignInService
{
    bool Callback(string url);

    Task<AppleAccount> SignInAsync();
}
```

On iOS, the native APIs are used:

```
public class AppleSignInServiceiOS : IAppleSignInService
{
#if __IOS__13
    AuthManager authManager;
#endif

    bool Is13 => UIDevice.CurrentDevice.CheckSystemVersion(13, 0);
    WebAppleSignInService webSignInService;

    public AppleSignInServiceiOS()
    {
        if (!Is13)
            webSignInService = new WebAppleSignInService();
    }
}
```

```

public async Task<AppleAccount> SignInAsync()
{
    // Fallback to web for older iOS versions
    if (!Is13)
        return await webSignInService.SignInAsync();

    AppleAccount appleAccount = default;

#if __IOS__13
    var provider = new ASAAuthorizationAppleIdProvider();
    var req = provider.CreateRequest();

    authManager = new AuthManager(UIApplication.SharedApplication.KeyWindow);

    req.RequestedScopes = new[] { ASAAuthorizationScope.FullName, ASAAuthorizationScope.Email };
    var controller = new ASAAuthorizationController(new[] { req });

    controller.Delegate = authManager;
    controller.PresentationContextProvider = authManager;

    controller.PerformRequests();

    var creds = await authManager.Credentials;

    if (creds == null)
        return null;

    appleAccount = new AppleAccount();
    appleAccount.IdToken = JwtToken.Decode(new NSString(creds.IdentityToken,
    NSStringEncoding.UTF8).ToString());
    appleAccount.Email = creds.Email;
    appleAccount.UserId = creds.User;
    appleAccount.Name = NSPersonNameComponentsFormatter.GetLocalizedString(creds.FullName,
    NSPersonNameComponentsFormatterStyle.Default, NSPersonNameComponentsFormatterOptions.Phonetic);
    appleAccount.RealUserStatus = creds.RealUserStatus.ToString();
#endif

    return appleAccount;
}

public bool Callback(string url) => true;
}

#if __IOS__13
class AuthManager : NSObject, IASAAuthorizationControllerDelegate,
IASAuthorizationControllerPresentationContextProviding
{
    public Task<ASAAuthorizationAppleIdCredential> Credentials
        => tcsCredential?.Task;

    TaskCompletionSource<ASAAuthorizationAppleIdCredential> tcsCredential;

    UIWindow presentingAnchor;

    public AuthManager(UIWindow presentingWindow)
    {
        tcsCredential = new TaskCompletionSource<ASAAuthorizationAppleIdCredential>();
        presentingAnchor = presentingWindow;
    }

    public UIWindow GetPresentationAnchor(ASAAuthorizationController controller)
        => presentingAnchor;

    [Export("authorizationController:didCompleteWithAuthorization:")]
    public void DidComplete(ASAAuthorizationController controller, ASAAuthorization authorization)
    {
        var creds = authorization.GetCredential<ASAAuthorizationAppleIdCredential>();
        tcsCredential?.TrySetResult(creds);
    }
}

```

```
        }

    [Export("authorizationController:didCompleteWithError:")]
    public void DidComplete(ASAAuthorizationController controller, NSError error)
        => tcsCredential?.TrySetException(new Exception(error.LocalizedDescription));
}

#endif
```

The compile flag `__IOS_13` is used to provide support for iOS 13 as well as legacy versions that fallback to the generic web service.

On Android, the generic web service with Azure Functions is used:

```

public class WebAppleSignInService : IAppleSignInService
{
    // IMPORTANT: This is what you register each native platform's url handler to be
    public const string CallbackUriScheme = "xamarinformsapplesignin";
    public const string InitialAuthUrl = "http://local.test:7071/api/applesignin_auth";

    string currentState;
    string currentNonce;

    TaskCompletionSource<AppleAccount> tcsAccount = null;

    public bool Callback(string url)
    {
        // Only handle the url with our callback uri scheme
        if (!url.StartsWith(CallbackUriScheme + "://"))
            return false;

        // Ensure we have a task waiting
        if (tcsAccount != null && !tcsAccount.Task.IsCompleted)
        {
            try
            {
                // Parse the account from the url the app opened with
                var account = AppleAccount.FromUrl(url);

                // IMPORTANT: Validate the nonce returned is the same as our originating request!!
                if (!account.IdToken.Nonce.Equals(currentNonce))
                    tcsAccount.TrySetException(new InvalidOperationException("Invalid or non-matching nonce
returned"));

                // Set our account result
                tcsAccount.TrySetResult(account);
            }
            catch (Exception ex)
            {
                tcsAccount.TrySetException(ex);
            }
        }

        tcsAccount.TrySetResult(null);
        return false;
    }

    public async Task<AppleAccount> SignInAsync()
    {
        tcsAccount = new TaskCompletionSource<AppleAccount>();

        // Generate state and nonce which the server will use to initial the auth
        // with Apple. The nonce should flow all the way back to us when our function
        // redirects to our app
        currentState = Util.GenerateState();
        currentNonce = Util.GenerateNonce();

        // Start the auth request on our function (which will redirect to apple)
        // inside a browser (either SFSafariViewController, Chrome Custom Tabs, or native browser)
        await Xamarin.Essentials.Browser.OpenAsync($"{InitialAuthUrl}?&state={currentState}&nonce=
{currentNonce}",
            Xamarin.Essentials.BrowserLaunchMode.SystemPreferred);

        return await tcsAccount.Task;
    }
}

```

Summary

This article described the steps necessary to setup Sign In with Apple for use in your Xamarin.Forms applications.

Related links

- [XamarinFormsAppleSignIn \(Sample\)](#)
- [Sign In with Apple Guidelines](#)

Xamarin.Forms Other Platforms

8/4/2022 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms supports additional platforms beyond iOS, Android, and Windows.

IMPORTANT

For more information about supported Xamarin.Forms platforms, see [Xamarin.Forms Platform Support](#).

GTK

Xamarin.Forms now has preview support for GTK# apps.

Mac

Xamarin.Forms now has preview support for macOS apps.

Tizen

Tizen .NET enables you to build .NET applications with Xamarin.Forms and the Tizen .NET Framework.

WPF

Xamarin.Forms now has preview support for Windows Presentation Foundation (WPF) apps.

GTK# Platform Setup

8/4/2022 • 5 minutes to read • [Edit Online](#)



Preview

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.

IMPORTANT

Xamarin.Forms support for GTK# is provided by the community. For more information, see [Xamarin.Forms Platform Support](#).

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.

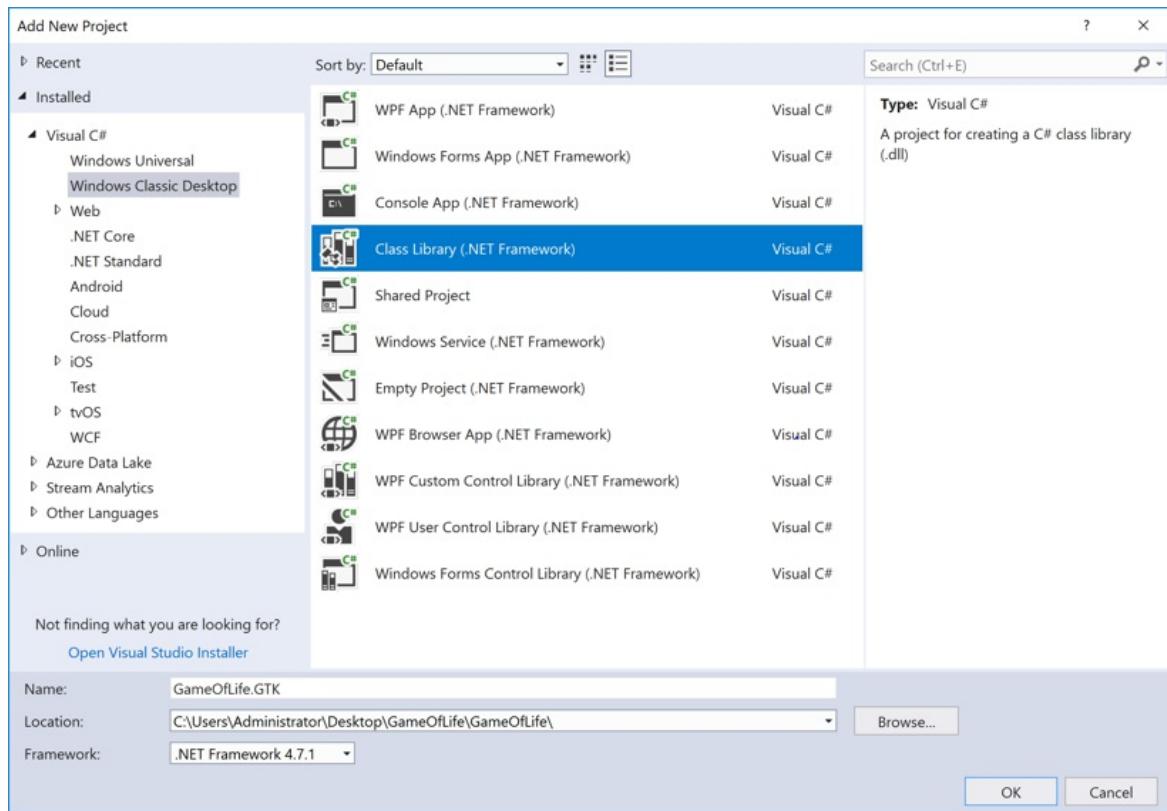
Adding a GTK# App

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](#)

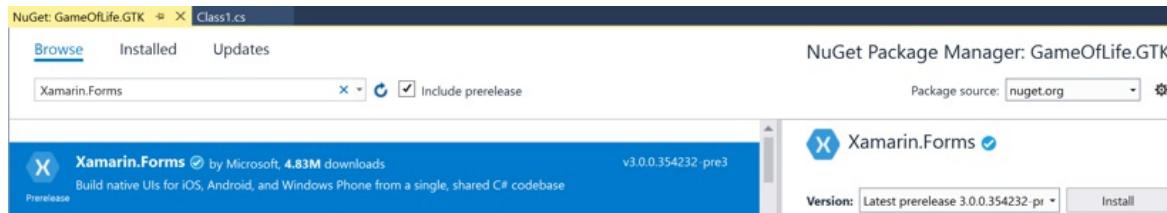
Follow these instructions to add a GTK# app that will run on the Windows desktop:

1. In Visual Studio 2019, 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.



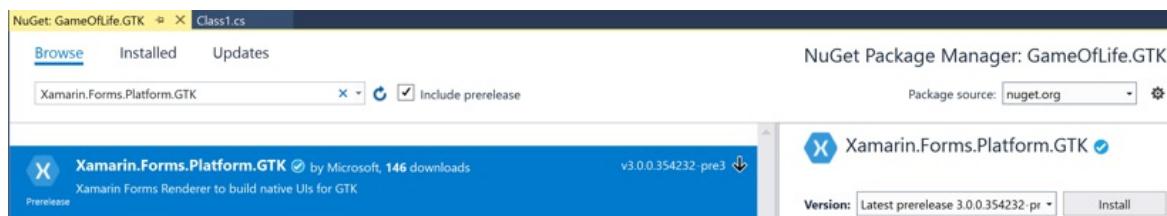
Press the **OK** button to create the project.

4. 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.



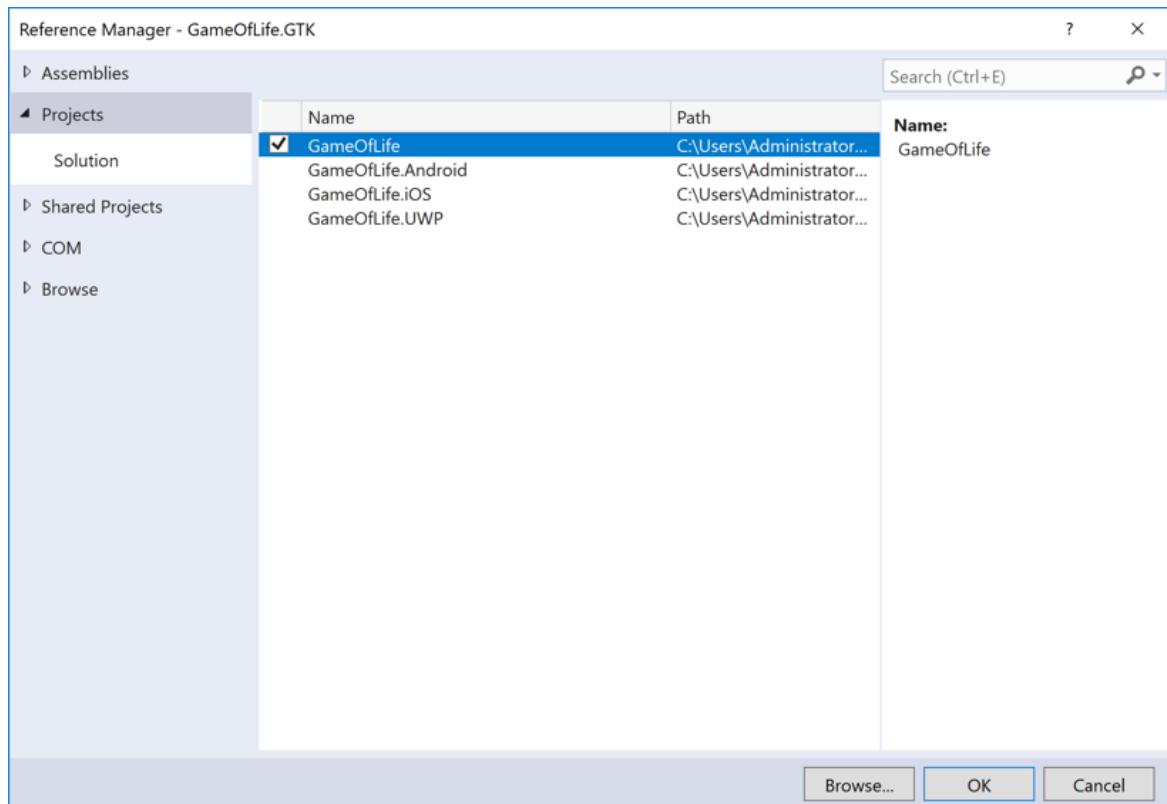
Select the package and click the **Install** button.

5. Now search for the **Xamarin.Forms.Platform.GTK** 3.0 package or greater.

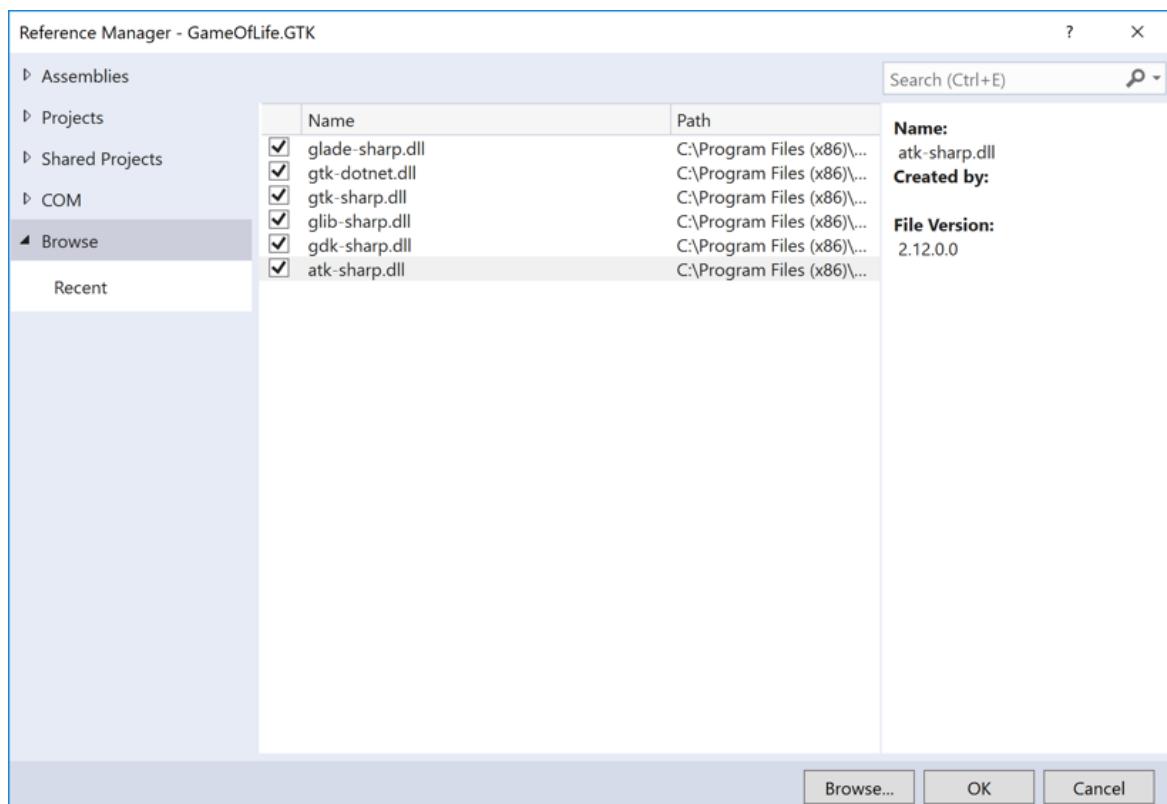


Select the package and click the **Install** button.

6. 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.
7. 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();

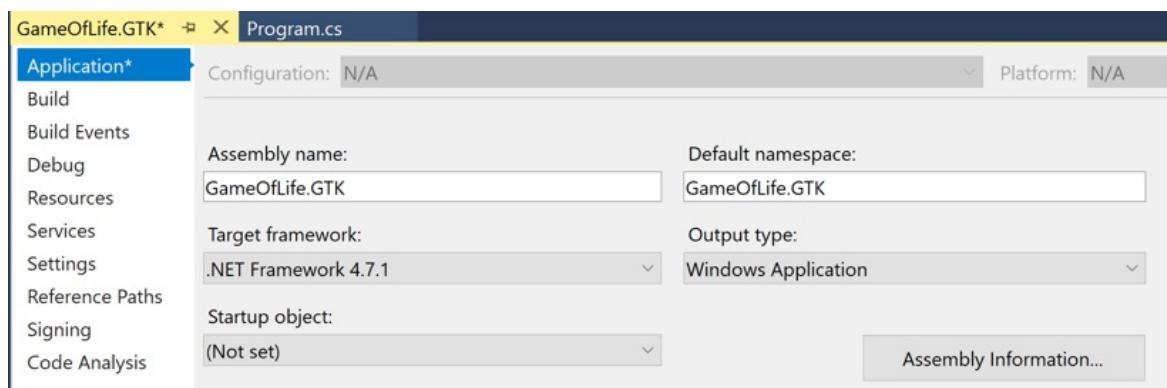
            var app = new App();
            var window = new FormsWindow();
            window.LoadApplication(app);
            window.SetApplicationTitle("Game of Life");
            window.Show();

            Gtk.Application.Run();
        }
    }
}

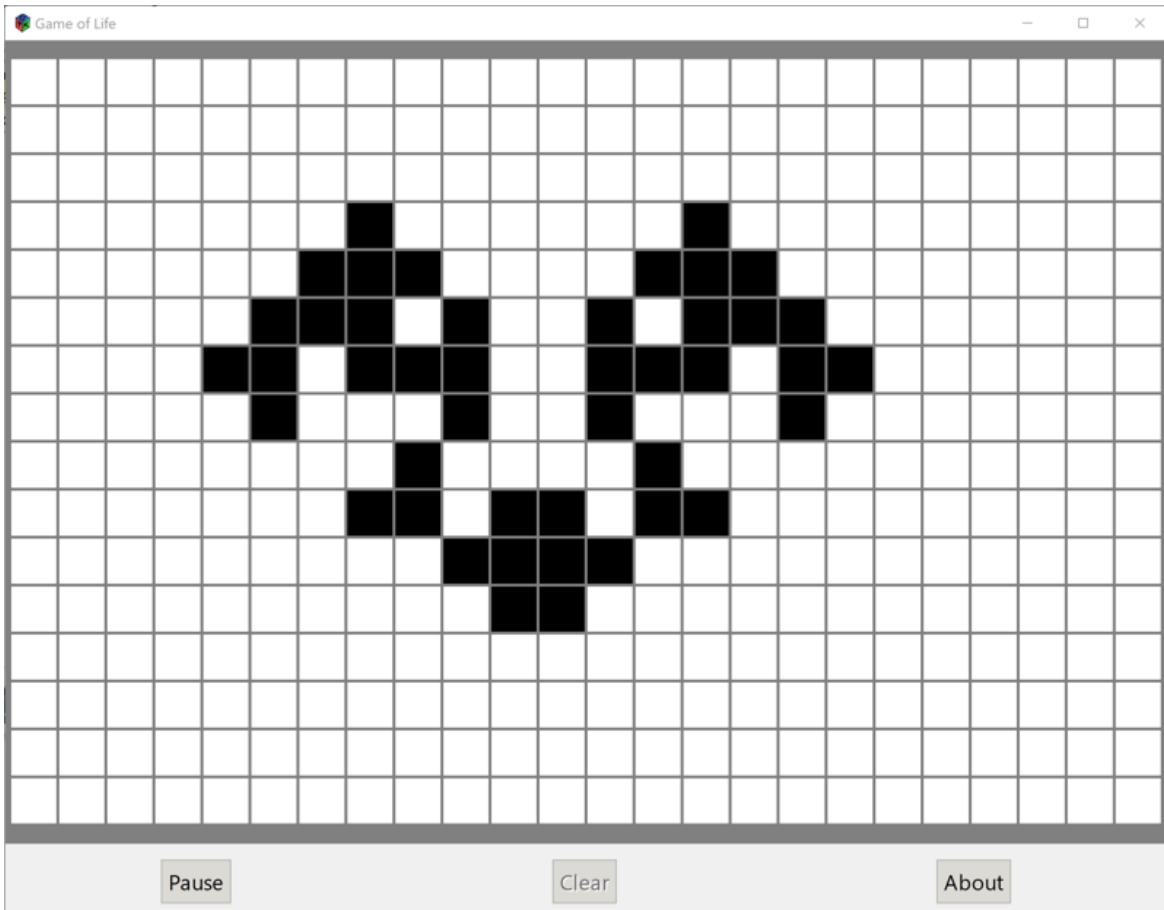
```

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 GTK 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

8/4/2022 • 2 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 video

Adding a Mac App

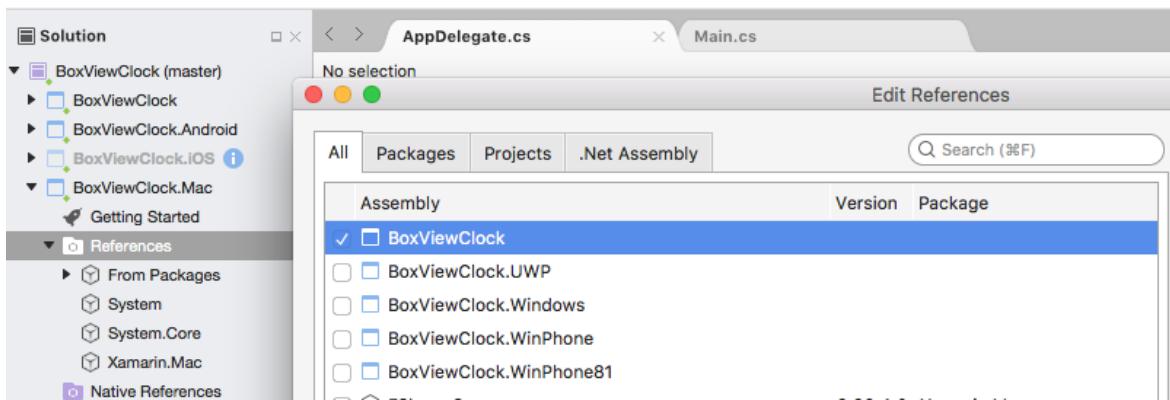
Follow these instructions to add a Mac app that will run on macOS Sierra and macOS El Capitan:

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 "WeatherApp.iOS". The Solution Explorer on the left shows a solution named "WeatherApp" containing projects for iOS, UWP, Droid, and Mac. The Mac project, "WeatherApp.iOS", is selected. The main workspace shows the "App.cs" file open in the code editor. The code in App.cs 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** NuGet. You should also update the other projects to use the same version of the Xamarin.Forms NuGet package.
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.*

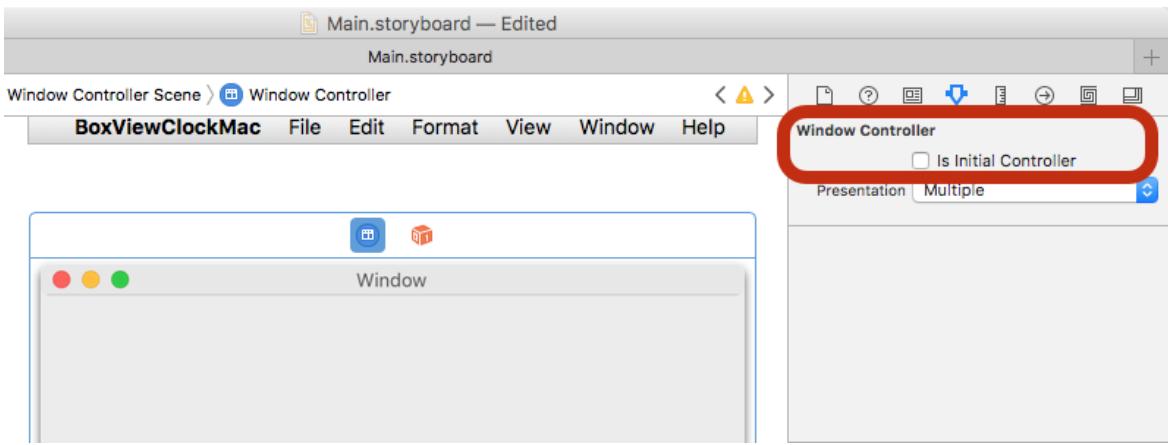
```
using Xamarin.Forms;
using Xamarin.Forms.Platform.MacOS;
// also add a using for the Xamarin.Forms project, if the namespace is different to this file
...
[Register("AppDelegate")]
public class AppDelegate : FormsApplicationDelegate
{
    NSWindow window;
    public AppDelegate()
    {
        var 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

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. For more information, see [Platform Support](#) [macOS Status](#) in the [Xamarin.Forms GitHub repository](#).

Related Links

- [Xamarin.Mac](#)

Tizen .NET

8/4/2022 • 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. The Tizen .NET platform is supported by Samsung. 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.

WPF platform setup

8/4/2022 • 3 minutes to read • [Edit Online](#)



Xamarin.Forms has preview support for the Windows Presentation Foundation (WPF), on .NET Framework and on .NET Core 3. This article demonstrates how to add a WPF project that targets .NET Framework, to a Xamarin.Forms solution.

IMPORTANT

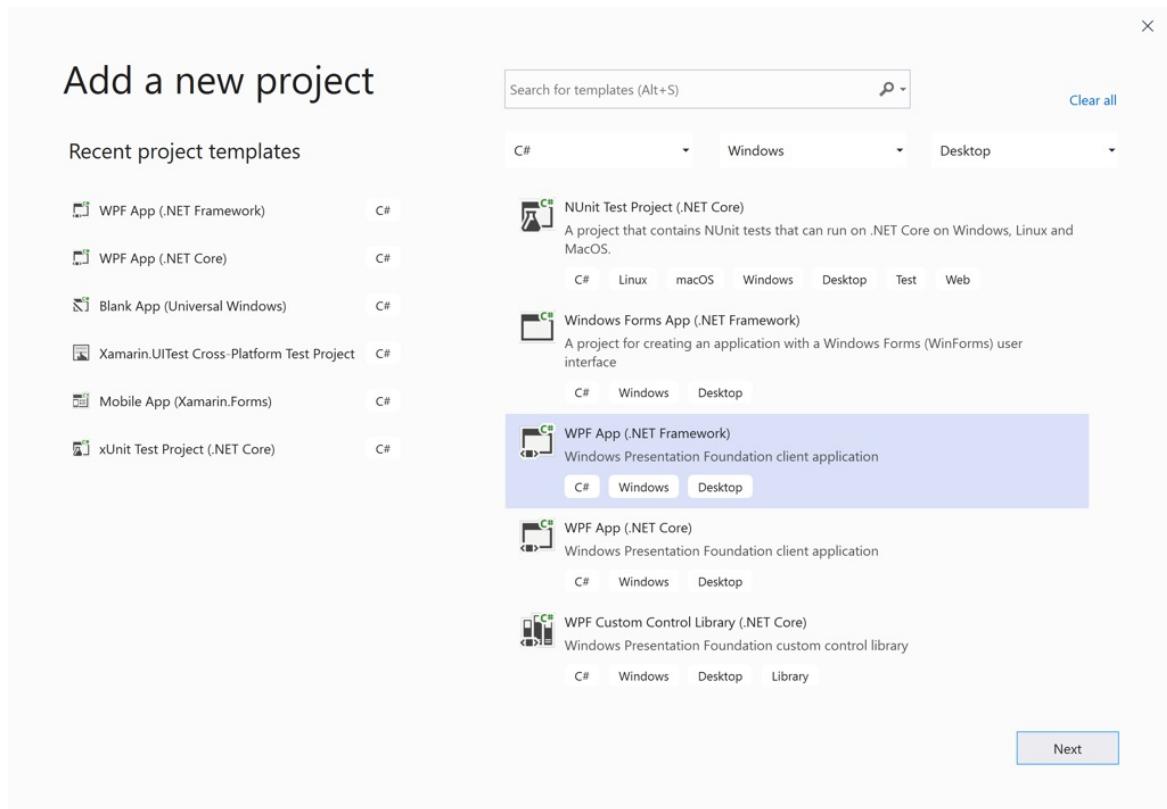
Xamarin.Forms support for WPF is provided by the community. For more information, see [Xamarin.Forms Platform Support](#).

Before you start, create a new Xamarin.Forms solution in Visual Studio 2019, 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 application

Follow these instructions to add a WPF application that will run on the Windows 7, 8, and 10 desktops:

1. In Visual Studio 2019, right-click on the solution name in the **Solution Explorer** and choose **Add > New Project....**
2. In the **Add a new project** window, select **C#** in the **Languages** drop down, select **Windows** in the **Platforms** drop down, and select **Desktop** in the **Project type** drop down. In the list of project types, choose **WPF App (.NET Framework)**:

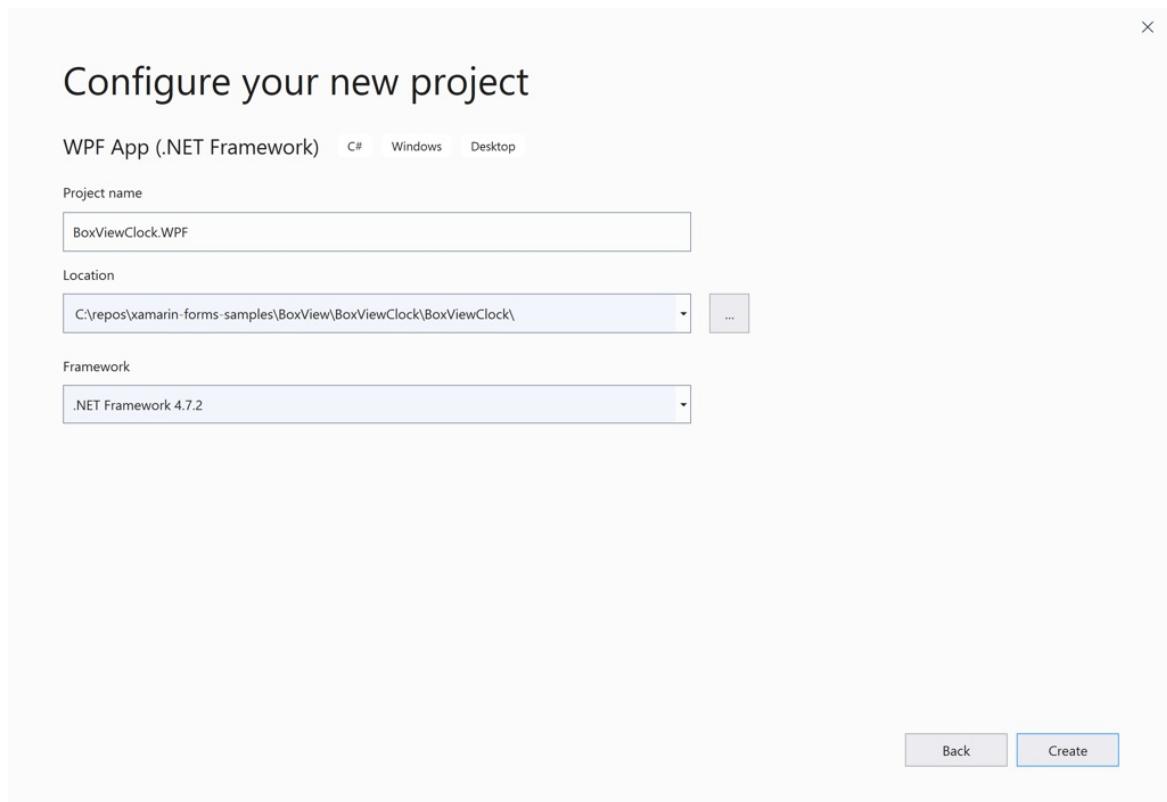


Press the **Next** button.

NOTE

Xamarin.Forms 4.7 includes support for WPF apps that run on .NET Core 3.

3. In the **Configure your new project** window, 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** to put the WPF project in the same directory as the other projects in the solution:



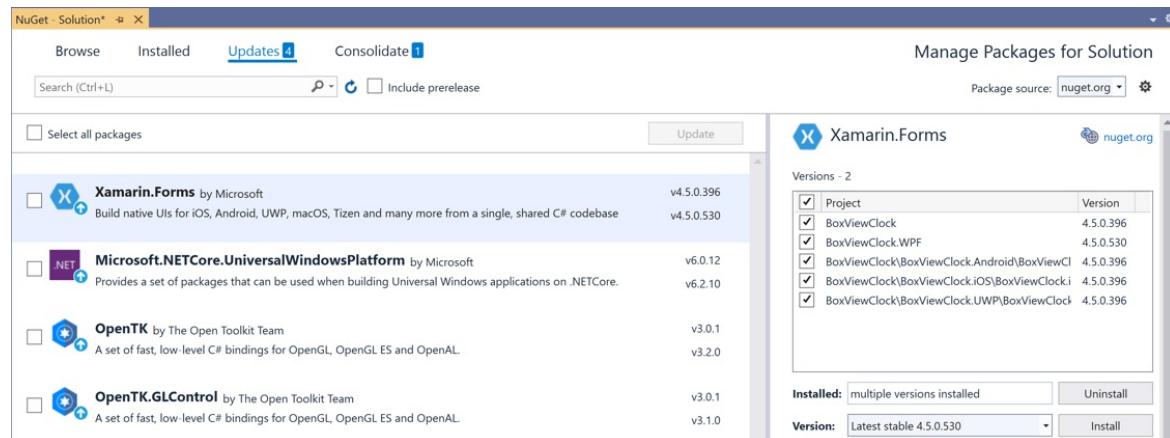
Press the **Create** button to create the project.

4. In the **Solution Explorer**, right click the new **BoxViewClock.WPF** project and select **Manage NuGet Packages**.... Select the **Browse** tab, and search for **Xamarin.Forms.Platform.WPF**:

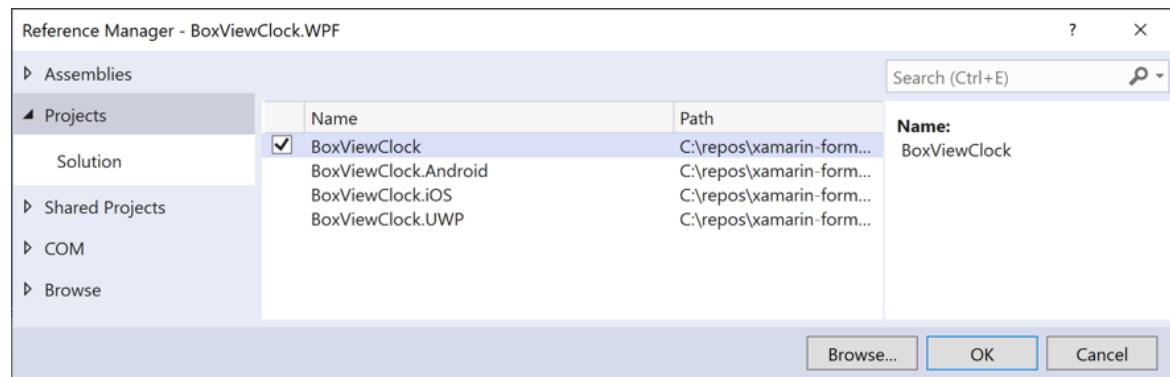


Select the package and click the **Install** button.

5. Right click the solution name in the **Solution Explorer** and select **Manage NuGet Packages for Solution**.... Select the **Updates** tab and then select the **Xamarin.Forms** package. Select all the projects and update them to the same **Xamarin.Forms** version:



6. In the WPF project, right-click on **References** and select **Add Reference**.... In the **Reference Manager** dialog, select **Projects** at the left, and check the checkbox adjacent to the **BoxViewClock** project:



Press the **OK** button.

7. 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:local="clr-namespace:BoxViewClock.WPF"
    mc:Ignorable="d"
    Title="BoxViewClock" Height="450" Width="800">
<Grid>

</Grid>
</wpf:FormsApplicationPage>
```

8. 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());
```

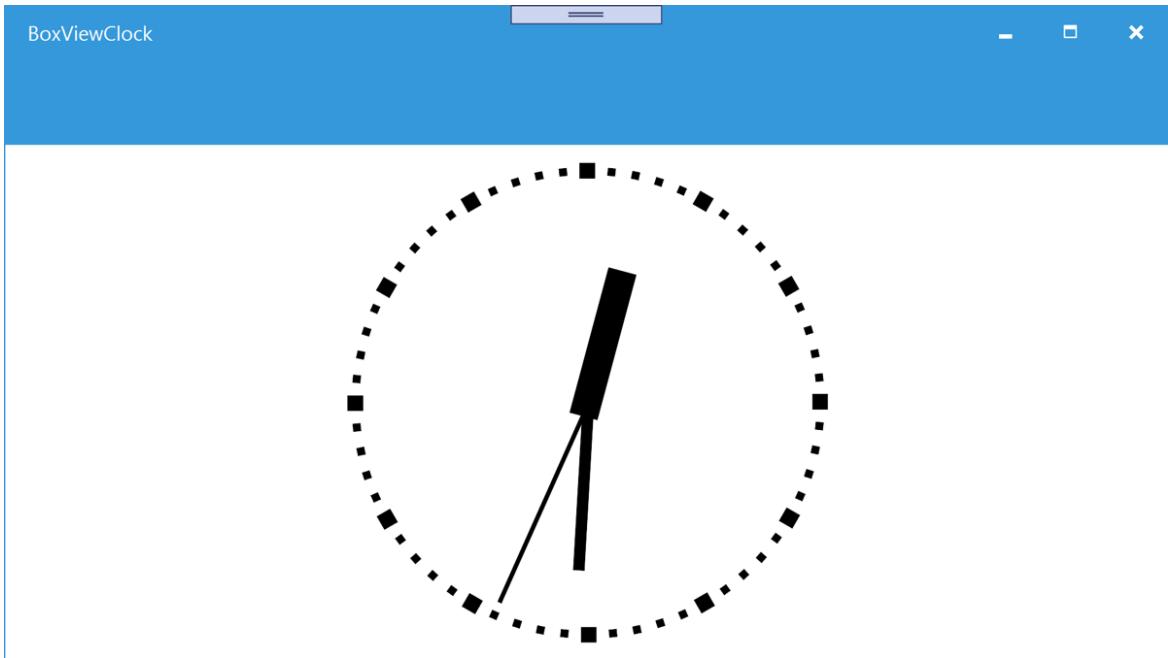
Except for comments and unused `using` directives, the complete **MainWindows.xaml.cs** file should look like this:

```
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());
        }
    }
}
```

9. 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:



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.

Related video

[Xamarin.Forms 3.0 WPF support video](#)

Xamarin.Essentials

8/4/2022 • 2 minutes to read • [Edit Online](#)

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 Actions](#) – Get and set shortcuts for the application.
- [App Information](#) – Find out information about the application.
- [App Theme](#) – Detect the current theme requested for 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.
- [Color Converters](#) – Helper methods for System.Drawing.Color.
- [Compass](#) – Monitor compass for changes.
- [Connectivity](#) – Check connectivity state and detect changes.
- [Contacts](#) – Retrieve information about a contact on the device.
- [Detect Shake](#) – Detect a shake movement of the device.
- [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 Picker](#) – Allow user to pick files from the device.
- [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.
- [Haptic Feedback](#) – Control click and long press haptics.
- [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.
- [Media Picker](#) – Allow user to pick or take photos and videos.

- [Open Browser](#) – Quickly and easily open a browser to a specific website.
- [Orientation Sensor](#) – Retrieve the orientation of the device in three dimensional space.
- [Permissions](#) – Check and request permissions from users.
- [Phone Dialer](#) – Open the phone dialer.
- [Platform Extensions](#) – Helper methods for converting Rect, Size, and Point.
- [Preferences](#) – Quickly and easily add persistent preferences.
- [Screenshot](#) – Take a capture of the current display of the application.
- [Secure Storage](#) – Securely store data.
- [Share](#) – Send text and website links to other apps.
- [SMS](#) – Create an SMS message for sending.
- [Text-to-Speech](#) – Vocalize text on the device.
- [Unit Converters](#) – Helper methods to convert units.
- [Version Tracking](#) – Track the applications version and build numbers.
- [Vibrate](#) – Make the device vibrate.
- [Web Authenticator](#) - Start web authentication flows and listen for a callback.

Troubleshooting

Find help if you are running into issues.

Xamarin.Essentials on Q&A

Ask questions about accessing native features with Xamarin.Essentials.

Release Notes

Find full release notes for each release of Xamarin.Essentials.

API Documentation

Browse the API documentation for every feature of Xamarin.Essentials.

Get Started with Xamarin.Essentials

8/4/2022 • 3 minutes to read • [Edit Online](#)

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. See the [platform & feature support guide](#) for more information on supported operating systems.

Installation

Xamarin.Essentials is available as a NuGet package and is included in every new project in Visual Studio. It can also be added to any existing projects using Visual Studio with the following steps.

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.

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 9.0 or 10.0, corresponding to API level 28 and level 29. (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.)

When compiling against Android 9.0, Xamarin.Essentials installs version 28.0.0.3 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 28.0.0.3 using the NuGet package manager. All [Xamarin.Android.Support](#) libraries used by your application should be the same, and should be at least

version 28.0.0.3. Refer to the [troubleshooting page](#) if you have issues adding the Xamarin.Essentials NuGet or updating NuGets in your solution.

Starting with version 1.5.0 when compiling against Android 10.0, Xamarin.Essentials install AndroidX support libraries that it requires. Read through the [AndroidX documentation](#) if you have not made the transition yet.

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, it may  
    also be called: bundle  
    //...
```

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,  
    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.

Platform Support

8/4/2022 • 2 minutes to read • [Edit Online](#)

Xamarin.Essentials supports the following platforms and operating systems:

PLATFORM	VERSION
Android	4.4 (API 19) or higher
iOS	10.0 or higher
Tizen	4.0 or higher
tvOS	10.0 or higher
watchOS	4.0 or higher
UWP	10.0.16299.0 or higher
macOS	10.12.6 (Sierra) or higher

NOTE

- Tizen is officially supported by the Samsung development team.
- tvOS & watchOS have limited API coverage, please see the feature guide for more information.
- macOS support is in preview.

Feature Support

Xamarin.Essentials always tries to bring features to every platform, however sometimes there are limitations based on the device. Below is a guide of what features are supported on each platform.

Icon Guide:

- - Full support
- - Limited support
- - Not supported

FEATURE	ANDROID	IOS	UWP	WATCHOS	TVOS	TIZEN	MACOS
Accelerometer							
App Actions							
App Information							

Feature	Android	iOS	UWP	WatchOS	tvOS	Tizen	MacOS
App Theme	✓	✓	✓	✓	✗	✓	✓
Barometer	✓	✓	✓	✓	✗	✓	✗
Battery	✓	✓	✓	⚠	✗	⚠	✓
Clipboard	✓	✓	✓	✗	✗	✗	✓
Color Converters	✓	✓	✓	✓	✓	✓	✓
Compass	✓	✓	✓	✗	✗	✓	✗
Connectivity	✓	✓	✓	✗	✓	✓	✓
Contacts	✓	✓	✓	✗	✗	✓	✗
Detect Shake	✓	✓	✓	✓	✓	✓	✗
Device Display Information	✓	✓	✓	✗	✗	✗	✓
Device Information	✓	✓	✓	✓	✓	✓	✓
Email	✓	✓	✓	✗	✗	✓	✓
File Picker	✓	✓	✓	✗	✗	✓	✓
File System Helpers	✓	✓	✓	✓	✓	✓	✓
Flashlight	✓	✓	✓	✗	✗	✓	✗
Geocoding	✓	✓	✓	✓	✓	✓	✓
Geolocation	✓	✓	✓	✗	✗	✓	✓
Gyroscope	✓	✓	✓	✓	✗	✓	✗
Haptic Feedback	✓	✓	✓	✗	✗	✓	✓
Launcher	✓	✓	✓	✗	✗	✓	✓
Magnetometer	✓	✓	✓	✓	✗	✓	✗

Feature	Android	iOS	UWP	WatchOS	tvOS	Tizen	MacOS
MainThread	✓	✓	✓	✓	✓	✓	✓
Maps	✓	✓	✓	✓	✗	✓	✓
Media Picker	✓	✓	✓	✗	✗	✓	⚠
Open Browser	✓	✓	✓	✗	✗	✓	✓
Orientation Sensor	✓	✓	✓	✓	✗	✓	✗
Permissions	✓	✓	✓	✓	✓	✓	✓
Phone Dialer	✓	✓	✓	✗	✗	✓	✓
Platform Extensions	✓	✓	✓	✓	✓	✓	✓
Preferences	✓	✓	✓	✓	✓	✓	✓
Screenshot	✓	✓	✓	✗	✗	✗	✗
Secure Storage	✓	✓	✓	✓	✓	✓	✓
Share	✓	✓	✓	✗	✗	✓	✓
SMS	✓	✓	✓	✗	✗	✓	✓
Text-to-Speech	✓	✓	✓	✓	✓	✓	✓
Unit Converters	✓	✓	✓	✓	✓	✓	✓
Version Tracking	✓	✓	✓	✓	✓	✓	✓
Vibrate	✓	✓	✓	✗	✗	✓	✗
Web Authenticator	✓	✓	✓	✗	✓	✗	✓

Xamarin.Essentials: Accelerometer

8/4/2022 • 2 minutes to read • [Edit Online](#)

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_RadingChanged;
    }

    void Accelerometer_RadingChanged(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 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 G or ($+9.81 \text{ m/s}^2$), which correspond to the acceleration of the device (0 m/s^2) minus the force of gravity (-9.81 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 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 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).
- **Default** – 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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: App Actions

8/4/2022 • 2 minutes to read • [Edit Online](#)

The **AppActions** class lets you create and respond to app shortcuts from the app icon.

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 **AppActions** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

Add the intent filter to your `MainActivity` class:

```
[IntentFilter(
    new[] { Xamarin.Essentials.Platform.Intent.ActionAppAction },
    Categories = new[] { Android.Content.Intent.CategoryDefault })]
public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
{
    ...
}
```

Then add the following logic to handle actions:

```
protected override void OnResume()
{
    base.OnResume();

    Xamarin.Essentials.Platform.OnResume(this);
}

protected override void OnNewIntent(Android.Content.Intent intent)
{
    base.OnNewIntent(intent);

    Xamarin.Essentials.Platform.OnNewIntent(intent);
}
```

Create Actions

Add a reference to `Xamarin.Essentials` in your class:

```
using Xamarin.Essentials;
```

App Actions can be created at any time, but are often created when an application starts. Call the `SetAsync` method to create the list of actions for your app.

```

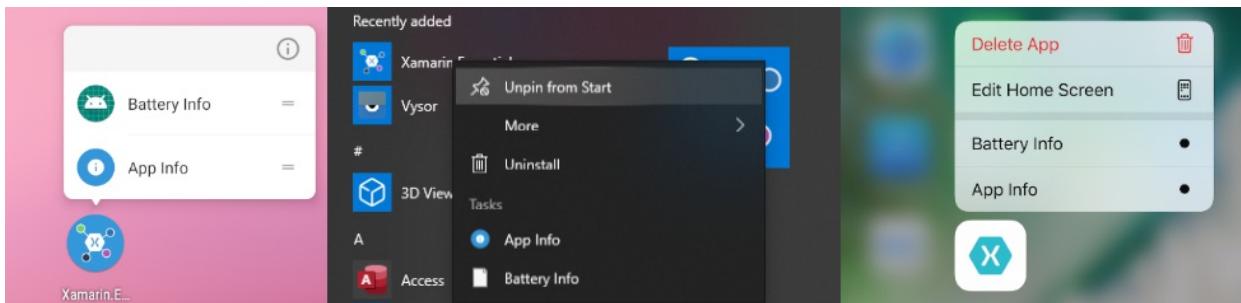
try
{
    await AppActions.SetAsync(
        new AppAction("app_info", "App Info", icon: "app_info_action_icon"),
        new AppAction("battery_info", "Battery Info"));
}
catch (FeatureNotSupportedException ex)
{
    Debug.WriteLine("App Actions not supported");
}

```

If App Actions are not supported on the specific version of the operating system a `FeatureNotSupportedException` will be thrown.

The following properties can be set on an `AppAction`:

- Id: A unique identifier used to respond to the action tap.
- Title: the visible title to display.
- Subtitle: If supported a sub-title to display under the title.
- Icon: Must match icons in the corresponding resources directory on each platform.



Responding To Actions

When your application starts register for the `OnAppAction` event. When an app action is selected the event will be sent with information as to which action was selected.

```

public App()
{
    //...
    AppActions.OnAppAction += AppActions_OnAppAction;
}

void AppActions_OnAppAction(object sender, AppActionEventArgs e)
{
    // Don't handle events fired for old application instances
    // and cleanup the old instance's event handler
    if (Application.Current != this && Application.Current is App app)
    {
        AppActions.OnAppAction -= app.AppActions_OnAppAction;
        return;
    }
    MainThread.BeginInvokeOnMainThread(async () =>
    {
        await Shell.Current.GoToAsync($"//{e.AppAction.Id}");
    });
}

```

GetActions

You can get the current list of App Actions by calling `AppActions.GetAsync()`.

API

- [AppActions source code](#)
- [AppActions API documentation](#)

Related Video

Xamarin.Essentials: App Information

8/4/2022 • 2 minutes to read • [Edit Online](#)

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.ShowSettingsUI();
```

This settings page allows the user to change application permissions and perform other platform-specific tasks.

Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)

App information is taken from the `AndroidManifest.xml` for the following fields:

- **Build** – `android:versionCode` in `manifest` node

- **Name** - `android:label` in the `application` node
- **PackageName**: `package` in the `manifest` node
- **VersionString** – `android:versionName` in the `application` node

API

- [AppInfo source code](#)
- [AppInfo API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: App Theme

8/4/2022 • 2 minutes to read • [Edit Online](#)

The **RequestedTheme** API is part of the [AppInfo](#) class and provides information as to what theme is requested for your running app by the system.

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 RequestedTheme

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Obtaining Theme Information

The requested application theme can be detected with the following API:

```
AppTheme appTheme = AppInfo.RequestedTheme;
```

This will provide the current requested theme by the system for your application. The return value will be one of the following:

- Unspecified
- Light
- Dark

Unspecified will be returned when the operating system does not have a specific user interface style to request. An example of this is on devices running versions of iOS older than 13.0.

Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)

Android uses configuration modes to specify the type of theme to request from the user. Based on the version of Android, it can be changed by the user or is changed when battery saver mode is enabled.

You can read more on the official [Android documentation for Dark Theme](#).

API

- [AppInfo source code](#)
- [AppInfo API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Barometer

8/4/2022 • 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 hectopascals. 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.PressureInHectopascals} hectopascals");
    }

    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).
- **Default** – 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

8/4/2022 • 3 minutes to read • [Edit Online](#)

The **Battery** class lets you check the device's battery information and monitor for changes and 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.

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 when on AC or no battery.

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)
        break;
    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.BatteryInfoChanged += Battery_BatteryInfoChanged;
    }

    void Battery_BatteryInfoChanged(object sender, BatteryInfoChangedEventArgs e)
    {
        var level = e.ChargeLevel;
        var state = e.State;
        var source = e.PowerSource;
        Console.WriteLine($"Reading: Level: {level}, State: {state}, Source: {source}");
    }
}

```

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.

You can also obtain the current energy-saver status of the device using the static `Battery.EnergySaverStatus` property:

```
// Get energy saver status
var status = Battery.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 `Battery` 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
        Battery.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.

Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)

No platform differences.

API

- [Battery source code](#)
- [Battery API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Clipboard

8/4/2022 • 2 minutes to read • [Edit Online](#)

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**:

```
await Clipboard.SetTextAsync("Hello World");
```

To read text from the **Clipboard**:

```
var text = await Clipboard.GetTextAsync();
```

Whenever any of the clipboard's content has changed an event is triggered:

```
public class ClipboardTest
{
    public ClipboardTest()
    {
        // Register for clipboard changes, be sure to unsubscribe when needed
        Clipboard.ClipboardContentChanged += OnClipboardContentChanged;
    }

    void OnClipboardContentChanged(object sender, EventArgs e)
    {
        Console.WriteLine($"Last clipboard change at {DateTime.UtcNow:T}");
    }
}
```

TIP

Access to the **Clipboard** must be done on the main user interface thread. See the [MainThread](#) API to see how to invoke methods on the main user interface thread.

API

- [Clipboard source code](#)
- [Clipboard API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Color Converters

8/4/2022 • 2 minutes to read • [Edit Online](#)

The `ColorConverters` class in `Xamarin.Essentials` provides several helper methods for `System.Drawing.Color`.

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 Color Converters

Add a reference to `Xamarin.Essentials` in your class:

```
using Xamarin.Essentials;
```

When working with `System.Drawing.Color` you can use the built in converters of `Xamarin.Essentials` to create a color from Hsl, Hex, or UInt.

```
var blueHex = ColorConverters.FromHex("#3498db");
var blueHsl = ColorConverters.FromHsl(204, 70, 53);
var blueUInt = ColorConverters.FromUInt(3447003);
```

Using Color Extensions

Extension methods on `System.Drawing.Color` enable you to apply different properties:

```
var blue = ColorConverters.FromHex("#3498db");

// Multiplies the current alpha by 50%
var blueWithAlpha = blue.MultiplyAlpha(.5f);
```

There are several other extension methods including:

- `GetComplementary`
- `MultiplyAlpha`
- `ToUInt`
- `WithAlpha`
- `WithHue`
- `WithLuminosity`
- `WithSaturation`

Using Platform Extensions

Additionally, you can convert `System.Drawing.Color` to the platform specific color structure. These methods can only be called from the iOS, Android, and UWP projects.

```
var system = System.Drawing.Color.FromArgb(255, 52, 152, 219);

// Extension to convert to Android.Graphics.Color, UIKit.UIColor, or Windows.UI.Color
var platform = system.ToPlatformColor();
```

```
var platform = new Android.Graphics.Color(52, 152, 219, 255);

// Back to System.Drawing.Color
var system = platform.ToSystemColor();
```

The `ToSystemColor` method applies to `Android.Graphics.Color`, `UIKit.UIColor`, and `Windows.UI.Color`.

API

- [Color Converters source code](#)
- [Color Converters API documentation](#)
- [Color Extensions source code](#)
- [Color Extensions API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Compass

8/4/2022 • 2 minutes to read • [Edit Online](#)

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($"Rading: {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).
- **Default** – 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 an 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 maybe 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**.

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 using the `Start` method overload, which accepts the `bool applyLowPassFilter` parameter:

```
Compass.Start(SensorSpeed.UI, applyLowPassFilter: true);
```

This is only applied on the Android platform, and the parameter is ignored on iOS and UWP. More information can be read [here](#).

API

- [Compass source code](#)
- [Compass API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Connectivity

8/4/2022 • 2 minutes to read • [Edit Online](#)

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.ConnectionProfiles;
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.ConnectionProfiles;
    }
}
```

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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Contacts

8/4/2022 • 2 minutes to read • [Edit Online](#)

The **Contacts** class lets a user pick a contact and retrieve information about it.

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 **Contacts** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

The `ReadContacts` 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.ReadContacts)]
```

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.READ_CONTACTS" /> />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check this permission. This will automatically update the **AndroidManifest.xml** file.

Pick a contact

By calling `Contacts.PickContactAsync()` the contact dialog will appear and allow the user to receive information about the user.

```

try
{
    var contact = await Contacts.PickContactAsync();

    if(contact == null)
        return;

    var id = contact.Id;
    var namePrefix = contact.NamePrefix;
    var givenName = contact.GivenName;
    var middleName = contact.MiddleName;
    var familyName = contact.FamilyName;
    var nameSuffix = contact.NameSuffix;
    var displayName = contact.DisplayName;
    var phones = contact.Phones; // List of phone numbers
    var emails = contact Emails; // List of email addresses
}

catch (Exception ex)
{
    // Handle exception here.
}

```

Get all contacts

```

ObservableCollection<Contact> contactsCollect = new ObservableCollection<Contact>();

try
{
    // cancellationToken parameter is optional
    var cancellationToken = default(CancellationToken);
    var contacts = await Contacts.GetAllAsync(cancellationToken);

    if (contacts == null)
        return;

    foreach (var contact in contacts)
        contactsCollect.Add(contact);
}

catch (Exception ex)
{
    // Handle exception here.
}

```

Platform differences

- [Android](#)
- [iOS](#)
- [UWP](#)
- The `cancellationToken` parameter in the `GetAllAsync` method is only used on UWP.

API

- [Contacts source code](#)
- [Contacts API documentation](#)

Xamarin.Essentials: Detect Shake

8/4/2022 • 2 minutes to read • [Edit Online](#)

The [Accelerometer](#) class lets you monitor the device's accelerometer sensor, which indicates the acceleration of the device in three-dimensional space. Additionally, it enables you to register for events when the user shakes 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 Detect Shake

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To detect a shake of the device you must use the Accelerometer functionality by calling the `Start` and `Stop` methods to listen for changes to the acceleration and to detect a shake. Any time a shake is detected a `ShakeDetected` event will fire. It is recommended to use `Game` or faster for the `SensorSpeed`. Here is sample usage:

```

public class DetectShakeTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.Game;

    public DetectShakeTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        Accelerometer.ShakeDetected += Accelerometer_ShakeDetected ;
    }

    void Accelerometer_ShakeDetected (object sender, EventArgs e)
    {
        // Process shake event
    }

    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.
        }
    }
}

```

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).
- **Default** – 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.

Implementation Details

The detect shake API uses raw readings from the accelerometer to calculate acceleration. It uses a simple queue mechanism to detect if 3/4ths of the recent accelerometer events occurred in the last half second. Acceleration is calculated by adding the square of the X, Y, and Z readings from the accelerometer and comparing it to a specific threshold.

API

- [Accelerometer source code](#)
- [Accelerometer API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Device Display Information

8/4/2022 • 2 minutes to read • [Edit Online](#)

The **DeviceDisplay** class provides information about the device's screen metrics the application is running on and 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 DeviceDisplay

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Main Display Info

In addition to basic device information the **DeviceDisplay** class contains information about the device's screen and orientation.

```
// Get Metrics
var mainDisplayInfo = DeviceDisplay.MainDisplayInfo;

// Orientation (Landscape, Portrait, Square, Unknown)
var orientation = mainDisplayInfo.Orientation;

// Rotation (0, 90, 180, 270)
var rotation = mainDisplayInfo.Rotation;

// Width (in pixels)
var width = mainDisplayInfo.Width;

// Height (in pixels)
var height = mainDisplayInfo.Height;

// Screen density
var density = mainDisplayInfo.Density;
```

The **DeviceDisplay** class also exposes an event that can be subscribed to that is triggered whenever any screen metric changes:

```
public class DisplayInfoTest
{
    public DisplayInfoTest()
    {
        // Subscribe to changes of screen metrics
        DeviceDisplay.MainDisplayInfoChanged += OnMainDisplayInfoChanged;
    }

    void OnMainDisplayInfoChanged(object sender, DisplayInfoChangedEventArgs e)
    {
        // Process changes
        var displayInfo = e.DisplayInfo;
    }
}
```

Keep Screen On

The `DeviceDisplay` class exposes a `bool` property called `KeepScreenOn` that can be set to attempt to keep the device's display from turning off or locking.

```
public class KeepScreenOnTest
{
    public void ToggleScreenLock()
    {
        DeviceDisplay.KeepScreenOn = !DeviceDisplay.KeepScreenOn;
    }
}
```

Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)

No differences.

API

- [DeviceDisplay source code](#)
- [DeviceDisplay API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Device Information

8/4/2022 • 2 minutes to read • [Edit Online](#)

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 `DevicePlatform` struct:

- `DevicePlatform.iOS` – iOS
- `DevicePlatform.Android` – Android
- `DevicePlatform.UWP` – UWP
- `DevicePlatform.Unknown` – Unknown

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 `DeviceIdiom` struct:

- `DeviceIdiom.Phone` – Phone
- `DeviceIdiom.Tablet` – Tablet
- `DeviceIdiom.Desktop` – Desktop
- `DeviceIdiom.TV` – TV
- `DeviceIdiom.Watch` – Watch
- `DeviceIdiom.Unknown` – Unknown

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 model 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 these (non-official sources) [The iPhone Wiki](#) and [Get iOS Model](#).

API

- [DeviceInfo source code](#)
- [DeviceInfo API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Email

8/4/2022 • 2 minutes to read • [Edit Online](#)

The **Email** class enables an application to open the default email application with a specified information including subject, body, and recipients (TO, CC, BCC).

To access the **Email** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new [package visibility requirements](#).

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.SENDTO" />
    <data android:scheme="mailto" />
  </intent>
</queries>
```

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.

TIP

To use the Email API on iOS you must run it on a physical device, else an exception will be thrown.

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
        }
    }
}

```

File Attachments

This feature enables an app to email files in email clients on the device. Xamarin.Essentials will automatically detect the file type (MIME) and request the file to be added as an attachment. Every email client is different and may only support specific file extensions, or none at all.

Here is a sample of writing text to disk and adding it as an email attachment:

```

var message = new EmailMessage
{
    Subject = "Hello",
    Body = "World",
};

var fn = "Attachment.txt";
var file = Path.Combine(FileSystem.CacheDirectory, fn);
File.WriteAllText(file, "Hello World");

message.Attachments.Add(new EmailAttachment(file));

await Email.ComposeAsync(message);

```

Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)

Not all email clients for Android support `Html`, since there is no way to detect this we recommend using `PlainText` when sending emails.

API

- [Email source code](#)
- [Email API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: File Picker

8/4/2022 • 2 minutes to read • [Edit Online](#)

The **FilePicker** class lets a user pick a single or multiple files from 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.

To access the **FilePicker** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

No additional setup required.

TIP

All methods must be called on the UI thread because permission checks and requests are automatically handled by Xamarin.Essentials.

Pick File

`FilePicker.PickAsync()` method enables your user to pick a file from the device. You are able to specify different `PickOptions` when calling the method enabling you to specify the title to display and the file types the user is allowed to pick. By default

```
async Task<FileResult> PickAndShow(PickOptions options)
{
    try
    {
        var result = await FilePicker.PickAsync(options);
        if (result != null)
        {
            Text = $"File Name: {result.FileName}";
            if (result.FileName.EndsWith("jpg", StringComparison.OrdinalIgnoreCase) ||
                result.FileName.EndsWith("png", StringComparison.OrdinalIgnoreCase))
            {
                var stream = await result.OpenReadAsync();
                Image = ImageSource.FromStream(() => stream);
            }
        }

        return result;
    }
    catch (Exception ex)
    {
        // The user canceled or something went wrong
    }

    return null;
}
```

Default file types are provided with `FilePickerFileType.Images`, `FilePickerFileType.Png`, and `FilePickerFileType.Videos`. You can specify custom files types when creating the `PickOptions` and they can be customized per platform. For example here is how you would specify specific comic file types:

```
var customFileType =
    new FilePickerFileType(new Dictionary<DevicePlatform, IEnumerable<string>>
    {
        { DevicePlatform.iOS, new[] { "public.my.comic.extension" } },
        { DevicePlatform.Android, new[] { "application/comics" } },
        { DevicePlatform.UWP, new[] { ".cbr", ".cbz" } },
        { DevicePlatform.Tizen, new[] { "*/*" } },
        { DevicePlatform.macOS, new[] { "cbr", "cbz" } }
    });
var options = new PickOptions
{
    PickerTitle = "Please select a comic file",
    FileTypes = customFileType,
};
```

Pick Multiple Files

If you desire your user to pick multiple files you can call the `FilePicker.PickMultipleAsync()` method. It also takes in `PickOptions` as a parameter to specify additional information. The results are the same as `PickAsync`, but instead of a single `FileResult` an `IEnumerable<FileResult>` is returned that can be iterated over.

TIP

The `FullPath` property does not always return the physical path to the file. To get the file, use the `OpenReadAsync` method.

Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)
- No platform differences.

API

- [FilePicker source code](#)
- [FilePicker API documentation](#)

Related Video

Xamarin.Essentials: File System Helpers

8/4/2022 • 2 minutes to read • [Edit Online](#)

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, as the OS dictates when this storage is cleared.

```
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;
```

If you would like to open a file that has been bundled into the application package, you can use the `OpenAppPackageFileAsync` method to read the contents. In the example below a file named `mybundledfile.txt` has been added to each platform. See the platform implementation specifics section for details.

```
var fileName = "mybundledfile.txt";
using (var stream = await FileSystem.OpenAppPackageFileAsync(fileName))
{
    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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Flashlight

8/4/2022 • 2 minutes to read • [Edit Online](#)

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)]
```

This API uses runtime permissions on Android. Please ensure that Xamarin.Essentials is fully initialized and permission handling is setup in your app.

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, it may also
be called: bundle
    //...
}
```

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,
Android.Content.PM.Permission[] grantResults)
{
    Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions, grantResults);

    base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Geocoding

8/4/2022 • 2 minutes to read • [Edit Online](#)

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.

Using Reverse Geocoding

Reverse geocoding is the process of 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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Geolocation

8/4/2022 • 5 minutes to read • [Edit Online](#)

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.

If your application is targeting Android 10 - Q (API Level 29 or higher) and is requesting **LocationAlways**, you must also add the following permission into **AssemblyInfo.cs**:

```
[assembly: UsesPermission(Manifest.Permission.AccessBackgroundLocation)]
```

Or directly into your **AndroidManifest.xml**:

```
<uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
```

If it recommended to read [Android documentation on background location updates](#) as there are many restrictions that need to be considered.

This API uses runtime permissions on Android. Please ensure that Xamarin.Essentials is fully initialized and permission handling is setup in your app.

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, it may also
    be called: bundle
    //...
}
```

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,
Android.Content.PM.Permission[] grantResults)
{
    Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions, grantResults);

    base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

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 and may return `null` if no cached location exists.

```
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 (FeatureNotEnabledException fneEx)
{
    // Handle not enabled on device exception
}
catch (PermissionException pEx)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to get location
}
```

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.

```

CancellationTokenSource cts;

async Task GetCurrentLocation()
{
    try
    {
        var request = new GeolocationRequest(GeolocationAccuracy.Medium, TimeSpan.FromSeconds(10));
        cts = new CancellationTokenSource();
        var location = await Geolocation.GetLocationAsync(request, cts.Token);

        if (location != null)
        {
            Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude: {location.Altitude}");
        }
    }
    catch (FeatureNotSupportedException fnsEx)
    {
        // Handle not supported on device exception
    }
    catch (FeatureNotEnabledException fneEx)
    {
        // Handle not enabled on device exception
    }
    catch (PermissionException pEx)
    {
        // Handle permission exception
    }
    catch (Exception ex)
    {
        // Unable to get location
    }
}

protected override void OnDisappearing()
{
    if (cts != null && !cts.IsCancellationRequested)
        cts.Cancel();
    base.OnDisappearing();
}

```

Note all values may be available due to how each device queries geolocation through different providers. For example, the `Altitude` property might be `null`, have a value of 0, or have a positive value, which is in meters above sea level. Other values that may not be present include `Speed` and `Course`.

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

Detecting Mock Locations

Some devices may return a mock location from the provider or by an application that provides mock locations. You can detect this by using the `IsFromMockProvider` on any `Location`.

```
var request = new GeolocationRequest(GeolocationAccuracy.Medium);
var location = await Geolocation.GetLocationAsync(request);

if (location != null)
{
    if(location.IsFromMockProvider)
    {
        // location is from a mock provider
    }
}
```

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 `UnitConverters` class also defines `KilometersToMiles` and `MilesToKilometers` methods for converting between the two units.

Platform Differences

Altitude is calculated differently on each platform.

- [Android](#)
- [iOS](#)
- [UWP](#)

On Android, `altitude`, if available, is returned in meters above the WGS 84 reference ellipsoid. If this location does not have an altitude then 0.0 is returned.

API

- [Geolocation source code](#)
- [Geolocation API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Gyroscope

8/4/2022 • 2 minutes to read • [Edit Online](#)

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 in rad/s. 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 reported in rad/s
        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).
- **Default** – 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: Haptic Feedback

8/4/2022 • 2 minutes to read • [Edit Online](#)

The **HapticFeedback** class lets you control haptic feedback on 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.

To access the **HapticFeedback** 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 Haptic Feedback

Add a reference to **Xamarin.Essentials** in your class:

```
using Xamarin.Essentials;
```

The Haptic Feedback functionality can be performed with a **Click** or **LongPress** feedback type.

```
try
{
    // Perform click feedback
    HapticFeedback.Perform(HapticFeedbackType.Click);

    // Or use long press
    HapticFeedback.Perform(HapticFeedbackType.LongPress);
}
catch (FeatureNotSupportedException ex)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Other error has occurred.
}
```

API

- [HapticFeedback source code](#)
- [HapticFeedback API documentation](#)

Xamarin.Essentials: Launcher

8/4/2022 • 3 minutes to read • [Edit Online](#)

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");
    }
}
```

This can be combined into a single call with `TryOpenAsync`, which checks if the parameter can be opened and if so open it.

```
public class LauncherTest
{
    public async Task<bool> OpenRideShareAsync()
    {
        return await Launcher.TryOpenAsync("lyft://ridetype?id=lyft_line");
    }
}
```

Additional Platform Setup

- [Android](#)
- [iOS](#)
- [UWP](#)

No additional setup.

Files

This features enables an app to request other apps to open and view a file. Xamarin.Essentials will automatically detect the file type (MIME) and request the file to be opened.

Here is a sample of writing text to disk and requesting it be opened:

```
var fn = "File.txt";
var file = Path.Combine(FileSystem.CacheDirectory, fn);
File.WriteAllText(file, "Hello World");

await Launcher.OpenAsync(new OpenFileRequest
{
    File = new ReadOnlyFile(file)
});
```

Presentation Location When Opening Files

When requesting a share or opening launcher on iPadOS you have the ability to present in a pop over control. This specifies where the pop over will appear and point an arrow directly to. This location is often the control that launched the action. You can specify the location using the `PresentationSourceBounds` property:

```
await Share.RequestAsync(new ShareFileRequest
{
    Title = Title,
    File = new ShareFile(file),
    PresentationSourceBounds = DeviceInfo.Platform == DevicePlatform.iOS && DeviceInfo.Idiom == DeviceIdiom.Tablet
        ? new System.Drawing.Rectangle(0, 20, 0, 0)
        : System.Drawing.Rectangle.Empty
});
```

```
await Launcher.OpenAsync(new OpenFileRequest
{
    File = new ReadOnlyFile(file),
    PresentationSourceBounds = DeviceInfo.Platform == DevicePlatform.iOS && DeviceInfo.Idiom == DeviceIdiom.Tablet
        ? new System.Drawing.Rectangle(0, 20, 0, 0)
        : System.Drawing.Rectangle.Empty
});
```

Everything described here works equally for `Share` and `Launcher`.

If you are using Xamarin.Forms you are able to pass in a `View` and calculate the bounds:

```

public static class ViewHelpers
{
    public static Rectangle GetAbsoluteBounds(this Xamarin.Forms.View element)
    {
        Element looper = element;

        var absoluteX = element.X + element.Margin.Top;
        var absoluteY = element.Y + element.Margin.Left;

        // Add logic to handle titles, headers, or other non-view bars

        while (looper.Parent != null)
        {
            looper = looper.Parent;
            if (looper is Xamarin.Forms.View v)
            {
                absoluteX += v.X + v.Margin.Top;
                absoluteY += v.Y + v.Margin.Left;
            }
        }

        return new Rectangle(absoluteX, absoluteY, element.Width, element.Height);
    }

    public static System.Drawing.Rectangle ToSystemRectangle(this Rectangle rect) =>
        new System.Drawing.Rectangle((int)rect.X, (int)rect.Y, (int)rect.Width, (int)rect.Height);
}

```

This can then be used when calling `RequestAsync`:

```

public Command<Xamarin.Forms.View> ShareCommand { get; } = new Command<Xamarin.Forms.View>(Share);
async void Share(Xamarin.Forms.View element)
{
    try
    {
        Analytics.TrackEvent("ShareWithFriends");
        var bounds = element.GetAbsoluteBounds();

        await Share.RequestAsync(new ShareTextRequest
        {
            PresentationSourceBounds = bounds.ToSystemRectangle(),
            Title = "Title",
            Text = "Text"
        });
    }
    catch (Exception)
    {
        // Handle exception that share failed
    }
}

```

You can pass in the calling element when the `Command` is triggered:

```

<Button Text="Share"
       Command="{Binding ShareWithFriendsCommand}"
       CommandParameter="{Binding Source={RelativeSource Self}}"/>

```

Platform Differences

- [Android](#)
- [iOS](#)

- UWP

The Task returned from `CanOpenAsync` completes immediately.

API

- [Launcher source code](#)
- [Launcher API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Magnetometer

8/4/2022 • 2 minutes to read • [Edit Online](#)

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 μT (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).
- **Default** – 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

8/4/2022 • 3 minutes to read • [Edit Online](#)

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.

Additional Methods

The `MainThread` class includes the following additional `static` methods that can be used to interact with user interface elements from background threads:

METHOD	ARGUMENTS	RETURNS	PURPOSE
<code>InvokeOnMainThreadAsync<T></code>	<code>Func<T></code>	<code>Task<T></code>	Invokes a <code>Func<T></code> on the main thread, and waits for it to complete.
<code>InvokeOnMainThreadAsync</code>	<code>Action</code>	<code>Task</code>	Invokes an <code>Action</code> on the main thread, and waits for it to complete.

METHOD	ARGUMENTS	RETURNS	PURPOSE
<code>InvokeOnMainThreadAsync<T></code>	<code>Func<Task<T>></code>	<code>Task<T></code>	Invokes a <code>Func<Task<T>></code> on the main thread, and waits for it to complete.
<code>InvokeOnMainThreadAsync</code>	<code>Func<Task></code>	<code>Task</code>	Invokes a <code>Func<Task></code> on the main thread, and waits for it to complete.
<code>GetMainThreadSynchronizationContextAsync</code>		<code>Task<SynchronizationContext></code>	Returns the <code>SynchronizationContext</code> for the main thread.

API

- [MainThread source code](#)
- [MainThread API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Map

8/4/2022 • 2 minutes to read • [Edit Online](#)

The `Map` class enables an application to open the installed map 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 Map

Add a reference to `Xamarin.Essentials` in your class:

```
using Xamarin.Essentials;
```

The `Map` functionality works by calling the `OpenAsync` method with the `Location` or `Placemark` to open with optional `MapLaunchOptions`.

```
public class MapTest
{
    public async Task NavigateToBuilding25()
    {
        var location = new Location(47.645160, -122.1306032);
        var options = new MapLaunchOptions { Name = "Microsoft Building 25" };

        try
        {
            await Map.OpenAsync(location, options);
        }
        catch (Exception ex)
        {
            // No map application available to open
        }
    }
}
```

When opening with a `Placemark`, the following information is required:

- `CountryName`
- `AdminArea`
- `Thoroughfare`
- `Locality`

```

public class MapTest
{
    public async Task NavigateToBuilding25()
    {
        var placemark = new Placemark
        {
            CountryName = "United States",
            AdminArea = "WA",
            Thoroughfare = "Microsoft Building 25",
            Locality = "Redmond"
        };
        var options = new MapLaunchOptions { Name = "Microsoft Building 25" };

        try
        {
            await Map.OpenAsync(placemark, options);
        }
        catch (Exception ex)
        {
            // No map application available to open or placemark can not be located
        }
    }
}

```

Extension Methods

If you already have a reference to a `Location` or `Placemark`, you can use the built-in extension method `OpenMapAsync` with optional `MapLaunchOptions`:

```

public class MapTest
{
    public async Task OpenPlacemarkOnMap(Placemark placemark)
    {
        try
        {
            await placemark.OpenMapAsync();
        }
        catch (Exception ex)
        {
            // No map application available to open
        }
    }
}

```

Directions Mode

If you call `OpenMapAsync` without any `MapLaunchOptions`, 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 `NavigationMode` on the `MapLaunchOptions`:

```
public class MapTest
{
    public async Task NavigateToBuilding25()
    {
        var location = new Location(47.645160, -122.1306032);
        var options = new MapLaunchOptions { NavigationMode = NavigationMode.Driving };

        await Map.OpenAsync(location, options);
    }
}
```

Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)
- `NavigationMode` 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

- [Map source code](#)
- [Map API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Media Picker

8/4/2022 • 2 minutes to read • [Edit Online](#)

The **MediaPicker** class lets a user pick or take a photo or video 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.

To access the **MediaPicker** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

The following 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:

```
// Needed for Picking photo/video
[assembly: UsesPermission(Android.Manifest.Permission.ReadExternalStorage)]

// Needed for Taking photo/video
[assembly: UsesPermission(Android.Manifest.Permission.WriteExternalStorage)]
[assembly: UsesPermission(Android.Manifest.Permission.Camera)]

// Add these properties if you would like to filter out devices that do not have cameras, or set to false to
// make them optional
[assembly: UsesFeature("android.hardware.camera", Required = true)]
[assembly: UsesFeature("android.hardware.camera.autofocus", Required = true)]
```

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.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<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 these permissions. This will automatically update the **AndroidManifest.xml** file.

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new [package visibility requirements](#).

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
<intent>
    <action android:name="android.media.action.IMAGE_CAPTURE" />
</intent>
</queries>
```

Using Media Picker

The `MediaPicker` class has the following methods that all return a `FileResult` that can be used to get the files location or read it as a `Stream`.

- `PickPhotoAsync` : Opens the media browser to select a photo.
- `CapturePhotoAsync` : Opens the camera to take a photo.
- `PickVideoAsync` : Opens the media browser to select a video.
- `CaptureVideoAsync` : Opens the camera to take a video.

Each method optionally takes in a `MediaPickerOptions` parameter that allows the `Title` to be set on some operating systems that is displayed to the users.

TIP

All methods must be called on the UI thread because permission checks and requests are automatically handled by `Xamarin.Essentials`.

General Usage

```

async Task TakePhotoAsync()
{
    try
    {
        var photo = await MediaPicker.CapturePhotoAsync();
        await LoadPhotoAsync(photo);
        Console.WriteLine($"CapturePhotoAsync COMPLETED: {PhotoPath}");
    }
    catch (FeatureNotSupportedException fnsEx)
    {
        // Feature is not supported on the device
    }
    catch (PermissionException pEx)
    {
        // Permissions not granted
    }
    catch (Exception ex)
    {
        Console.WriteLine($"CapturePhotoAsync THREW: {ex.Message}");
    }
}

async Task LoadPhotoAsync(FileResult photo)
{
    // canceled
    if (photo == null)
    {
        PhotoPath = null;
        return;
    }
    // save the file into local storage
    var newFile = Path.Combine(FileSystem.CacheDirectory, photo.FileName);
    using (var stream = await photo.OpenReadAsync())
    using (var newStream = File.OpenWrite(newFile))
        await stream.CopyToAsync(newStream);

    PhotoPath = newFile;
}

```

TIP

The `FullPath` property does not always return the physical path to the file. To get the file, use the `OpenReadAsync` method.

API

- [MediaPicker source code](#)
- [MediaPicker API documentation](#)

Xamarin.Essentials: Browser

8/4/2022 • 2 minutes to read • [Edit Online](#)

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.

To access the **Browser** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new [package visibility requirements](#).

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="http"/>
  </intent>
  <intent>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="https"/>
  </intent>
</queries>
```

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)
    {
        try
        {
            await Browser.OpenAsync(uri, BrowserLaunchMode.SystemPreferred);
        }
        catch(Exception ex)
        {
            // An unexpected error occurred. No browser may be installed on the device.
        }
    }
}

```

This method returns after the browser was *launched* and not necessarily *closed* by the user. The `bool` result indicates whether the launching was successful or not.

Customization

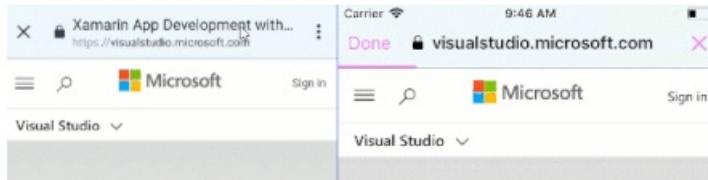
When using the system preferred browser there are several customization options available for iOS and Android. This includes a `TitleMode` (Android only), and preferred color options for the `Toolbar` (iOS and Android) and `Controls` (iOS only) that appear.

These options are specified using `BrowserLaunchOptions` when calling `OpenAsync`.

```

await Browser.OpenAsync(uri, new BrowserLaunchOptions
{
    LaunchMode = BrowserLaunchMode.SystemPreferred,
    TitleMode = BrowserTitleMode.Show,
    PreferredToolbarColor = Color.AliceBlue,
    PreferredControlColor = Color.Violet
});

```



Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)

The Launch Mode determines how the browser is launched:

System Preferred

[Custom Tabs](#) will attempt to be used to 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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: OrientationSensor

8/4/2022 • 3 minutes to read • [Edit Online](#)

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_ReadingChanged;
    }

    void OrientationSensor_ReadingChanged(object sender, OrientationSensorChangedEventArgs e)
    {
        var data = e.Rading;
        Console.WriteLine($"Reading: 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 Θ , 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).
- **Default** – 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: Permissions

8/4/2022 • 6 minutes to read • [Edit Online](#)

The `Permissions` class provides the ability to check and request runtime permissions.

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.

This API uses runtime permissions on Android. Please ensure that Xamarin.Essentials is fully initialized and permission handling is setup in your app.

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, it may also
    be called: bundle
    //...
}
```

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,
Android.Content.PM.Permission[] grantResults)
{
    Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions, grantResults);

    base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

Using Permissions

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Checking Permissions

To check the current status of a permission, use the `CheckStatusAsync` method along with the specific permission to get the status for.

```
var status = await Permissions.CheckStatusAsync<Permissions.LocationWhenInUse>();
```

A `PermissionException` is thrown if the required permission is not declared.

It's best to check the status of the permission before requesting it. Each operating system returns a different default state if the user has never been prompted. iOS returns `Unknown`, while others return `Denied`. If the status is `Granted` then there is no need to make other calls. On iOS if the status is `Denied` you should prompt the user to change the permission in the settings and on Android you can call `ShouldShowRationale` to detect if the user has already denied the permission in the past.

Requesting Permissions

To request a permission from the users, use the `RequestAsync` method along with the specific permission to request. If the user previously granted permission and hasn't revoked it, then this method will return `Granted` immediately and not display a dialog.

```
var status = await Permissions.RequestAsync<Permissions.LocationWhenInUse>();
```

A `PermissionException` is thrown if the required permission is not declared.

Note, that on some platforms a permission request can only be activated a single time. Further prompts must be handled by the developer to check if a permission is in the `Denied` state and ask the user to manually turn it on.

Permission Status

When using `CheckStatusAsync` or `RequestAsync` a `PermissionStatus` will be returned that can be used to determine the next steps:

- Unknown - The permission is in an unknown state
- Denied - The user denied the permission request
- Disabled - The feature is disabled on the device
- Granted - The user granted permission or is automatically granted
- Restricted - In a restricted state

Explain Why Permission Is Needed

It is best practice to explain why your application needs a specific permission. On iOS you must specify a string that is displayed to the user. Android does not have this ability and also defaults permission status to `Disabled`. This limits the ability to know if the user denied the permission or if it is the first time prompting the user. The `ShouldShowRationale` method can be used to determine if an educational UI should be displayed. If the method returns `true` this is because the user has denied or disabled the permission in the past. Other platforms will always return `false` when calling this method.

Available Permissions

Xamarin.Essentials attempts to abstract as many permissions as possible. However, each operating system has a different set of runtime permissions. In addition there are differences when providing a single API for some permissions. Here is a guide to the currently available permissions:

Icon Guide:

-  - Supported
-  - Not supported/required

PERMISSION	ANDROID	IOS	UWP	WATCHOS	TVOS	TIZEN
CalendarRead						

PERMISSION	ANDROID	IOS	UWP	WATCHOS	TVOS	TIZEN
------------	---------	-----	-----	---------	------	-------

CalendarWrite	✓	✓	✗	✓	✗	✗
Camera	✓	✓	✗	✗	✗	✓
ContactsRead	✓	✓	✓	✗	✗	✗
ContactsWrite	✓	✓	✓	✗	✗	✗
Flashlight	✓	✗	✗	✗	✗	✓
LocationWhenInUse	✓	✓	✓	✓	✓	✓
LocationAlways	✓	✓	✓	✓	✗	✓
Media	✗	✓	✗	✗	✗	✗
Microphone	✓	✓	✓	✗	✗	✓
Phone	✓	✓	✗	✗	✗	✗
Photos	✗	✓	✗	✗	✓	✗
Reminders	✗	✓	✗	✓	✗	✗
Sensors	✓	✓	✓	✓	✗	✗
Sms	✓	✓	✗	✗	✗	✗
Speech	✓	✓	✗	✗	✗	✗
StorageRead	✓	✗	✗	✗	✗	✗
StorageWrite	✓	✗	✗	✗	✗	✗

If a permission is marked as it will always return `Granted` when checked or requested.

General Usage

The following code presents the general usage pattern for determining whether a permission has been granted and requesting it if it has not. This code uses features that are available with Xamarin.Essentials version 1.6.0 or later.

```

public async Task<PermissionStatus> CheckAndRequestLocationPermission()
{
    var status = await Permissions.CheckStatusAsync<Permissions.LocationWhenInUse>();

    if (status == PermissionStatus.Granted)
        return status;

    if (status == PermissionStatus.Denied && DeviceInfo.Platform == DevicePlatform.iOS)
    {
        // Prompt the user to turn on in settings
        // On iOS once a permission has been denied it may not be requested again from the application
        return status;
    }

    if (Permissions.ShouldShowRationale<Permissions.LocationWhenInUse>())
    {
        // Prompt the user with additional information as to why the permission is needed
    }

    status = await Permissions.RequestAsync<Permissions.LocationWhenInUse>();

    return status;
}

```

Each permission type can have an instance of it created that the methods can be called directly.

```

public async Task GetLocationAsync()
{
    var status = await CheckAndRequestPermissionAsync(new Permissions.LocationWhenInUse());
    if (status != PermissionStatus.Granted)
    {
        // Notify user permission was denied
        return;
    }

    var location = await Geolocation.GetLocationAsync();
}

public async Task<PermissionStatus> CheckAndRequestPermissionAsync<T>(T permission)
    where T : BasePermission
{
    var status = await permission.CheckStatusAsync();
    if (status != PermissionStatus.Granted)
    {
        status = await permission.RequestAsync();
    }

    return status;
}

```

Extending Permissions

The Permissions API was created to be flexible and extensible for applications that require additional validation or permissions that aren't included in Xamarin.Essentials. Create a new class that inherits from `BasePermission` and implement the required abstract methods.

```

public class MyPermission : BasePermission
{
    // This method checks if current status of the permission
    public override Task<PermissionStatus> CheckStatusAsync()
    {
        throw new System.NotImplementedException();
    }

    // This method is optional and a PermissionException is often thrown if a permission is not declared
    public override void EnsureDeclared()
    {
        throw new System.NotImplementedException();
    }

    // Requests the user to accept or deny a permission
    public override Task<PermissionStatus> RequestAsync()
    {
        throw new System.NotImplementedException();
    }
}

```

When implementing a permission in a specific platform, the `BasePlatformPermission` class can be inherited from. This provides additional platform helper methods to automatically check the declarations. This can help when creating custom permissions that do groupings. For example, you can request both Read and Write access to storage on Android using the following custom permission.

```

public class ReadWriteStoragePermission : Xamarin.Essentials.Permissions.BasePlatformPermission
{
    public override (string androidPermission, bool isRuntime)[] RequiredPermissions => new List<(string
        androidPermission, bool isRuntime)>
    {
        (Android.Manifest.Permission.ReadExternalStorage, true),
        (Android.Manifest.Permission.WriteExternalStorage, true)
    }.ToArray();
}

```

Then you can call your new permission from Android project.

```
await Permissions.RequestAsync<ReadWriteStoragePermission>();
```

If you wanted to call this API from your shared code you could create an interface and use a [dependency service](#) to register and get the implementation.

```

public interface IReadWritePermission
{
    Task<PermissionStatus> CheckStatusAsync();
    Task<PermissionStatus> RequestAsync();
}

```

Then implement the interface in your platform project:

```
public class ReadWriteStoragePermission : Xamarin.Essentials.Permissions.BasePlatformPermission,
IReadWritePermission
{
    public override (string androidPermission, bool isRuntime)[] RequiredPermissions => new List<(string
    androidPermission, bool isRuntime)>
    {
        (Android.Manifest.Permission.ReadExternalStorage, true),
        (Android.Manifest.Permission.WriteExternalStorage, true)
    }.ToArray();
}
```

You can then register the specific implementation:

```
DependencyService.Register<IReadWritePermission, ReadWriteStoragePermission>();
```

Then from your shared project you can resolve and use it:

```
var readWritePermission = DependencyService.Get<IReadWritePermission>();
var status = await readWritePermission.CheckStatusAsync();
if (status != PermissionStatus.Granted)
{
    status = await readWritePermission.RequestAsync();
}
```

Platform Implementation Specifics

- [Android](#)
- [iOS](#)
- [UWP](#)

Permissions must have the matching attributes set in the Android Manifest file. Permission status defaults to Denied.

Read more on the [Permissions in Xamarin.Android](#) documentation.

API

- [Permissions source code](#)
- [Permissions API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Phone Dialer

8/4/2022 • 2 minutes to read • [Edit Online](#)

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.

- [Android](#)
- [iOS](#)
- [UWP](#)

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new [package visibility requirements](#).

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.DIAL" />
    <data android:scheme="tel"/>
  </intent>
</queries>
```

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 void 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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Platform Extensions

8/4/2022 • 2 minutes to read • [Edit Online](#)

Xamarin.Essentials provides several platform extension methods when having to work with platform types such as Rect, Size, and Point. This means that you can convert between the `System` version of these types for their iOS, Android, and UWP specific types.

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 Platform Extensions

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

All platform extensions can only be called from the iOS, Android, or UWP project.

Android Extensions

These extensions can only be accessed from an Android project.

Application Context & Activity

Using the platform extensions in the `Platform` class you can get access to the current `Context` or `Activity` for the running app.

```
var context = Platform.AppContext;

// Current Activity or null if not initialized or not started.
var activity = Platform.CurrentActivity;
```

If there is a situation where the `Activity` is needed, but the application hasn't fully started then the `WaitForActivityAsync` method should be used.

```
var activity = await Platform.WaitForActivityAsync();
```

Activity Lifecycle

In addition to getting the current Activity, you can also register for lifecycle events.

```

protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);

    Xamarin.Essentials.Platform.Init(this, bundle);

    Xamarin.Essentials.Platform.ActivityStateChanged += Platform_ActivityStateChanged;
}

protected override void OnDestroy()
{
    base.OnDestroy();
    Xamarin.Essentials.Platform.ActivityStateChanged -= Platform_ActivityStateChanged;
}

void Platform_ActivityStateChanged(object sender, Xamarin.Essentials.ActivityStateChangedEventArgs e) =>
    Toast.MakeText(this, e.State.ToString(), ToastLength.Short).Show();

```

Activity states are the following:

- Created
- Resumed
- Paused
- Destroyed
- SaveInstanceState
- Started
- Stopped

Read the [Activity Lifecycle](#) documentation to learn more.

iOS Extensions

These extensions can only be accessed from an iOS project.

Current UIViewController

Gain access to the currently visible `UIViewController`:

```
var vc = Platform.GetCurrentUIViewController();
```

This method will return `null` if unable to detect a `UIViewController`.

Cross-platform Extensions

These extensions exist in every platform.

Point

```

var system = new System.Drawing.Point(x, y);

// Convert to CoreGraphics.CGPoint, Android.Graphics.Point, and Windows.Foundation.Point
var platform = system.ToPlatformPoint();

// Back to System.Drawing.Point
var system2 = platform.ToSystemPoint();

```

Size

```
var system = new System.Drawing.Size(width, height);

// Convert to CoreGraphics.CGSize, Android.Util.Size, and Windows.Foundation.Size
var platform = system.ToPlatformSize();

// Back to System.Drawing.Size
var system2 = platform.ToSystemSize();
```

Rectangle

```
var system = new System.Drawing.Rectangle(x, y, width, height);

// Convert to CoreGraphics.CGRect, Android.Graphics.Rect, and Windows.Foundation.Rect
var platform = system.ToPlatformRectangle();

// Back to System.Drawing.Rectangle
var system2 = platform.ToSystemRectangle();
```

API

- [Converters source code](#)
- [Point Converters API documentation](#)
- [Rectangle Converters API documentation](#)
- [Size Converters API documentation](#)

Xamarin.Essentials: Preferences

8/4/2022 • 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 check if a given *key* exists in preferences:

```
bool hasKey = Preferences.ContainsKey("my_key");
```

To remove the *key* from preferences:

```
Preferences.Remove("my_key");
```

To remove all preferences:

```
Preferences.Clear();
```

TIP

The above methods take in an optional `string` parameter called `sharedName`. This parameter is used to create additional containers for preferences which are helpful in some use cases. One use case is when your application needs to share preferences across extensions or to a watch application. Please read the platform implementation specifics below.

Supported Data Types

The following data types are supported in `Preferences`:

- `bool`
- `double`
- `int`
- `float`
- `long`
- `string`
- `DateTime`

Integrate with System Settings

Preferences are stored natively, which allows you to integrate your settings into the native system settings. Follow the platform documentation and samples to integrate with the platform:

- Apple: [Implementing an iOS Settings Bundle](#)
- [iOS Application Preferences Sample](#)
- [watchOS Settings](#)
- Android: [Getting Started with Settings Screens](#)

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, otherwise 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, with the exception being apps that target and run on Android 6.0 (API level 23) or later that use [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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Screenshot

8/4/2022 • 2 minutes to read • [Edit Online](#)

The `Screenshot` class lets you take a capture of the current displayed screen of the app.

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 Screenshot

Add a reference to `Xamarin.Essentials` in your class:

```
using Xamarin.Essentials;
```

Then call `CaptureAsync` to take a screenshot of the current screen of the running application. This will return back a `ScreenshotResult` that can be used to get the `Width`, `Height`, and a `Stream` of the screenshot taken.

```
async Task CaptureScreenshot()
{
    var screenshot = await Screenshot.CaptureAsync();
    var stream = await screenshot.OpenReadAsync();

    Image = ImageSource.FromStream(() => stream);
}
```

Limitations

Not all views support being captured at a screen level such as an OpenGL view.

API

- [Screenshot source code](#)
- [Screenshot API documentation](#)

Xamarin.Essentials: Secure Storage

8/4/2022 • 4 minutes to read • [Edit Online](#)

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 with the build action of `AndroidResource`. 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();
```

TIP

It is possible that an exception is thrown when calling `GetAsync` or `SetAsync`. This can be caused by a device not supporting secure storage, encryption keys changing, or corruption of data. It is best to handle this by removing and adding the setting back if possible.

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 (not a cryptographic key, the *key* to the *value*) 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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Share

8/4/2022 • 3 minutes to read • [Edit Online](#)

The **Share** 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.

- [Android](#)
- [iOS](#)
- [UWP](#)

No additional setup required.

Using Share

Add a reference to Xamarin.Essentials in your class:

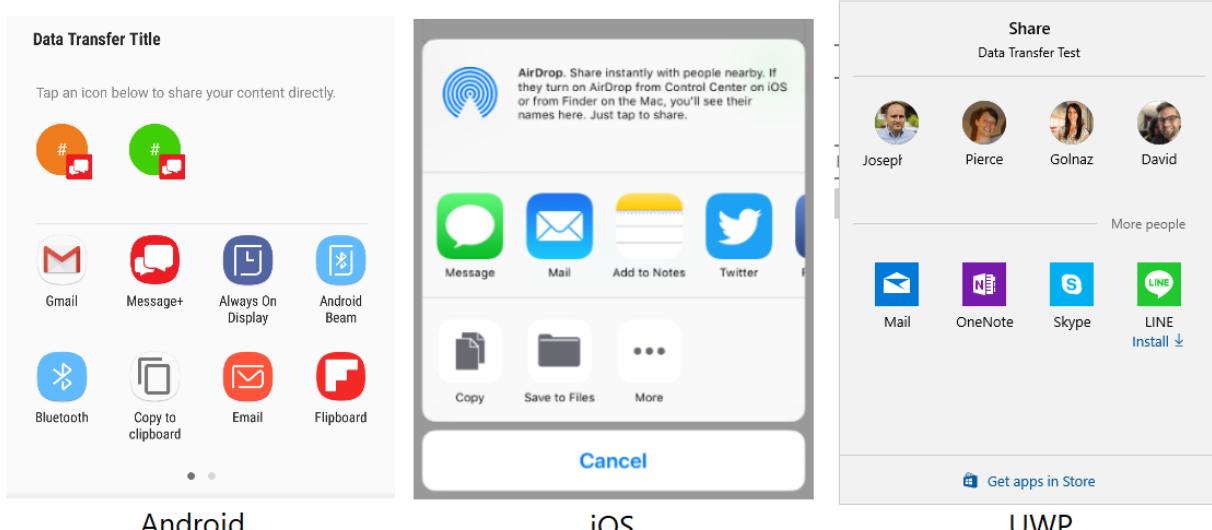
```
using Xamarin.Essentials;
```

The Share 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 ShareTest
{
    public async Task ShareText(string text)
    {
        await Share.RequestAsync(new ShareTextRequest
        {
            Text = text,
            Title = "Share Text"
        });
    }

    public async Task ShareUri(string uri)
    {
        await Share.RequestAsync(new ShareTextRequest
        {
            Uri = uri,
            Title = "Share Web Link"
        });
    }
}
```

User interface to share to external application that appears when request is made:



File

This feature enables an app to share files to other applications on the device. Xamarin.Essentials will automatically detect the file type (MIME) and request a share. Each platform may only support specific file extensions.

Here is a sample of writing text to disk and sharing it to other apps:

```
var fn = "Attachment.txt";
var file = Path.Combine(FileSystem.CacheDirectory, fn);
File.WriteAllText(file, "Hello World");

await Share.RequestAsync(new ShareFileRequest
{
    Title = Title,
    File = new ShareFile(file)
});
```

Multiple Files

The usage of share multiple files differs from the single file only in the ability of sending several files at once:

```
var file1 = Path.Combine(FileSystem.CacheDirectory, "Attachment1.txt");
File.WriteAllText(file, "Content 1");
var file2 = Path.Combine(FileSystem.CacheDirectory, "Attachment2.txt");
File.WriteAllText(file, "Content 2");

await Share.RequestAsync(new ShareMultipleFilesRequest
{
    Title = ShareFilesTitle,
    Files = new List<ShareFile> { new ShareFile(file1), new ShareFile(file2) }
});
```

Presentation Location

When requesting a share or opening launcher on iPadOS you have the ability to present in a pop over control. This specifies where the pop over will appear and point an arrow directly to. This location is often the control that launched the action. You can specify the location using the `PresentationSourceBounds` property:

```

await Share.RequestAsync(new ShareFileRequest
{
    Title = Title,
    File = new ShareFile(file),
    PresentationSourceBounds = DeviceInfo.Platform == DevicePlatform.iOS && DeviceInfo.Idiom == DeviceIdiom.Tablet
        ? new System.Drawing.Rectangle(0, 20, 0, 0)
        : System.Drawing.Rectangle.Empty
});

```

```

await Launcher.OpenAsync(new OpenFileRequest
{
    File = new ReadOnlyFile(file),
    PresentationSourceBounds = DeviceInfo.Platform == DevicePlatform.iOS && DeviceInfo.Idiom == DeviceIdiom.Tablet
        ? new System.Drawing.Rectangle(0, 20, 0, 0)
        : System.Drawing.Rectangle.Empty
});

```

Everything described here works equally for `Share` and `Launcher`.

If you are using Xamarin.Forms you are able to pass in a `View` and calculate the bounds:

```

public static class ViewHelpers
{
    public static Rectangle GetAbsoluteBounds(this Xamarin.Forms.View element)
    {
        Element looper = element;

        var absoluteX = element.X + element.Margin.Top;
        var absoluteY = element.Y + element.Margin.Left;

        // Add logic to handle titles, headers, or other non-view bars

        while (looper.Parent != null)
        {
            looper = looper.Parent;
            if (looper is Xamarin.Forms.View v)
            {
                absoluteX += v.X + v.Margin.Top;
                absoluteY += v.Y + v.Margin.Left;
            }
        }

        return new Rectangle(absoluteX, absoluteY, element.Width, element.Height);
    }

    public static System.Drawing.Rectangle ToSystemRectangle(this Rectangle rect) =>
        new System.Drawing.Rectangle((int)rect.X, (int)rect.Y, (int)rect.Width, (int)rect.Height);
}

```

This can then be used when calling `RequestAsync`:

```
public Command<Xamarin.Forms.View> ShareCommand { get; } = new Command<Xamarin.Forms.View>(Share);
async void Share(Xamarin.Forms.View element)
{
    try
    {
        Analytics.TrackEvent("ShareWithFriends");
        var bounds = element.GetAbsoluteBounds();

        await Share.RequestAsync(new ShareTextRequest
        {
            PresentationSourceBounds = bounds.ToSystemRectangle(),
            Title = "Title",
            Text = "Text"
        });
    }
    catch (Exception)
    {
        // Handle exception that share failed
    }
}
```

You can pass in the calling element when the `Command` is triggered:

```
<Button Text="Share"
       Command="{Binding ShareWithFriendsCommand}"
       CommandParameter="{Binding Source={RelativeSource Self}}"/>
```

Platform Differences

- [Android](#)
- [iOS](#)
- [UWP](#)
- `Subject` property is used for desired subject of a message.

API

- [Share source code](#)
- [Share API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: SMS

8/4/2022 • 2 minutes to read • [Edit Online](#)

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.

To access the **Sms** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new [package visibility requirements](#).

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="smsto"/>
  </intent>
</queries>
```

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, new []{ 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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Text-to-Speech

8/4/2022 • 2 minutes to read • [Edit Online](#)

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.

To access the **TextToSpeech** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new [package visibility requirements](#).

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.TTS_SERVICE" />
  </intent>
</queries>
```

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.

```

CancellationSource cts;
public async Task SpeakNowDefaultSettings()
{
    cts = new CancellationTokenSource();
    await TextToSpeech.SpeakAsync("Hello World", cancelToken: cts.Token);

    // This method will block until utterance finishes.
}

// Cancel speech if a cancellation token exists & hasn't been already requested.
public void CancelSpeech()
{
    if (cts?.IsCancellationRequested ?? true)
        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 `SpeechOptions` that allows setting the volume, pitch, and locale.

```

public async Task SpeakNow()
{
    var settings = new SpeechOptions()
    {
        Volume = .75f,
        Pitch = 1.0f
    };

    await TextToSpeech.SpeakAsync("Hello World", settings);
}

```

The following are supported values for these parameters:

PARAMETER	MINIMUM	MAXIMUM
Pitch	0	2.0
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 SpeechOptions()
    {
        Volume = .75f,
        Pitch = 1.0f,
        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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Unit Converters

8/4/2022 • 2 minutes to read • [Edit Online](#)

The `UnitConverters` class provides several unit converters to help developers when using Xamarin.Essentials.

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 Unit Converters

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

All unit converters are available by using the static `UnitConverters` class in Xamarin.Essentials. For instance you can easily convert Fahrenheit to Celsius.

```
var celsius = UnitConverters.FahrenheitToCelsius(32.0);
```

Here is a list of available conversions:

- `FahrenheitToCelsius`
- `CelsiusToFahrenheit`
- `CelsiusToKelvin`
- `KelvinToCelsius`
- `MilesToMeters`
- `MilesToKilometers`
- `KilometersToMiles`
- `MetersToInternationalFeet`
- `InternationalFeetToMeters`
- `DegreesToRadians`
- `RadiansToDegrees`
- `DegreesPerSecondToRadiansPerSecond`
- `RadiansPerSecondToDegreesPerSecond`
- `DegreesPerSecondToHertz`
- `RadiansPerSecondToHertz`
- `HertzToDegreesPerSecond`
- `HertzToRadiansPerSecond`
- `KilopascalsToHectopascals`
- `HectopascalsToKilopascals`
- `KilopascalsToPascals`
- `HectopascalsToPascals`
- `AtmospheresToPascals`
- `PascalsToAtmospheres`

- CoordinatesToMiles
- CoordinatesToKilometers
- KilogramsToPounds
- PoundsToKilograms
- StonesToPounds
- PoundsToStones

API

- [Unit Converters source code](#)
- [Unit Converters API documentation](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Version Tracking

8/4/2022 • 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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Vibration

8/4/2022 • 2 minutes to read • [Edit Online](#)

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](#)

Related Video

Find more Xamarin videos on [Channel 9](#) and [YouTube](#).

Xamarin.Essentials: Web Authenticator

8/4/2022 • 7 minutes to read • [Edit Online](#)

The **WebAuthenticator** class lets you initiate browser based flows which listen for a callback to a specific URL registered to the app.

Overview

Many apps require adding user authentication, and this often means enabling your users to sign in their existing Microsoft, Facebook, Google, and now Apple Sign In accounts.

[Microsoft Authentication Library \(MSAL\)](#) provides an excellent turn-key solution to adding authentication to your app. There's even support for Xamarin apps in their client NuGet package.

If you're interested in using your own web service for authentication, it's possible to use **WebAuthenticator** to implement the client side functionality.

Why use a server back end?

Many authentication providers have moved to only offering explicit or two-legged authentication flows to ensure better security. This means you'll need a '*client secret*' from the provider to complete the authentication flow. Unfortunately, mobile apps are not a great place to store secrets and anything stored in a mobile app's code, binaries, or otherwise is generally considered to be insecure.

The best practice here is to use a web backend as a middle layer between your mobile app and the authentication provider.

IMPORTANT

We strongly recommend against using older mobile-only authentication libraries and patterns which do not leverage a web backend in the authentication flow due to their inherent lack of security for storing client secrets.

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 **WebAuthenticator** functionality the following platform specific setup is required.

- [Android](#)
- [iOS](#)
- [UWP](#)

Android requires an Intent Filter setup to handle your callback URI. This is easily accomplished by subclassing the `WebAuthenticatorCallbackActivity` class:

```
const string CALLBACK_SCHEME = "myapp";

[Activity(NoHistory = true, LaunchMode = LaunchMode.SingleTop, Exported = true)]
[IntentFilter(new[] { Android.Content.Intent.ActionView },
    Categories = new[] { Android.Content.Intent.CategoryDefault, Android.Content.Intent.CategoryBrowsable },
    DataScheme = CALLBACK_SCHEME)]
public class WebAuthenticationCallbackActivity : Xamarin.Essentials.WebAuthenticatorCallbackActivity
{
}
```

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new [package visibility requirements](#).

Open the **AndroidManifest.xml** file under the Properties folder and add the following inside of the manifest node:

```
<queries>
    <intent>
        <action android:name="android.support.customtabs.action.CustomTabsService" />
    </intent>
</queries>
```

Using WebAuthenticator

Add a reference to `Xamarin.Essentials` in your class:

```
using Xamarin.Essentials;
```

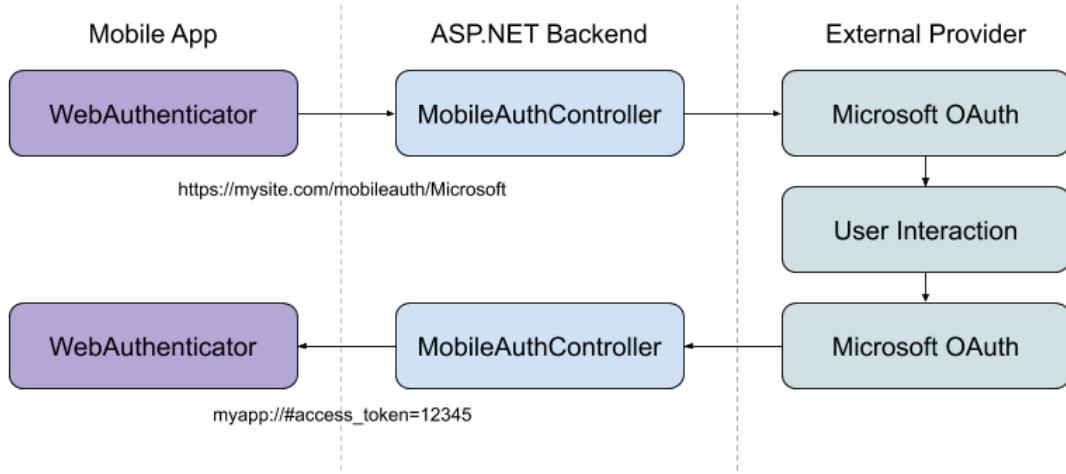
The API consists mainly of a single method `AuthenticateAsync` which takes two parameters: The url which should be used to start the web browser flow, and the Uri which you expect the flow to ultimately call back to and which your app is registered to be able to handle.

The result is a `WebAuthenticatorResult` which includes any query parameters parsed from the callback URI:

```
var authResult = await WebAuthenticator.AuthenticateAsync(
    new Uri("https://mysite.com/mobileauth/Microsoft"),
    new Uri("myapp://"));

var accessToken = authResult?.AccessToken;
```

The `WebAuthenticator` API takes care of launching the url in the browser and waiting until the callback is received:



If the user cancels the flow at any point, a `TaskCanceledException` is thrown.

Private authentication session

iOS 13 introduced an ephemeral web browser API for developers to launch the authentication session as private. This enables developers to request that no shared cookies or browsing data is available between authentication sessions and will be a fresh login session each time. This is available through the new `WebAuthenticatorOptions` that was introduced in Xamarin.Essentials 1.7 for iOS.

```

var url = new Uri("https://mysite.com/mobileauth/Microsoft");
var callbackUrl = new Uri("myapp:///");
var authResult = await WebAuthenticator.AuthenticateAsync(new WebAuthenticatorOptions
{
    Url = url,
    CallbackUrl = callbackUrl,
    PrefersEphemeralWebBrowserSession = true
});

```

Platform differences

- [Android](#)
- [iOS](#)
- [UWP](#)

Custom Tabs are used whenever available, otherwise an Intent is started for the URL.

Apple Sign In

According to [Apple's review guidelines](#), if your app uses any social login service to authenticate, it must also offer Apple Sign In as an option.

To add Apple Sign In to your apps, first you'll need to [configure your app to use Apple Sign In](#).

For iOS 13 and higher you'll want to call the `AppleSignInAuthenticator.AuthenticateAsync()` method. This will use the native Apple Sign in API's under the hood so your users get the best experience possible on these devices. You can write your shared code to use the right API at runtime like this:

```

var scheme = "..."; // Apple, Microsoft, Google, Facebook, etc.
WebAuthenticatorResult r = null;

if (scheme.Equals("Apple")
    && DeviceInfo.Platform == DevicePlatform.iOS
    && DeviceInfo.Version.Major >= 13)
{
    // Use Native Apple Sign In API's
    r = await AppleSignInAuthenticator.AuthenticateAsync();
}
else
{
    // Web Authentication flow
    var authUrl = new Uri(authenticationUrl + scheme);
    var callbackUrl = new Uri("xamarinessentials://");

    r = await WebAuthenticator.AuthenticateAsync(authUrl, callbackUrl);
}

var authToken = string.Empty;
if (r.Properties.TryGetValue("name", out var name) && !string.IsNullOrEmpty(name))
    authToken += $"Name: {name}{Environment.NewLine}";
if (r.Properties.TryGetValue("email", out var email) && !string.IsNullOrEmpty(email))
    authToken += $"Email: {email}{Environment.NewLine}";

// Note that Apple Sign In has an IdToken and not an AccessToken
authToken += r?.AccessToken ?? r?.IdToken;

```

TIP

For non-iOS 13 devices this will start the web authentication flow, which can also be used to enable Apple Sign In on your Android and UWP devices. You can sign into your iCloud account on your iOS simulator to test Apple Sign In.

ASP.NET core server back end

It's possible to use the `WebAuthenticator` API with any web back end service. To use it with an ASP.NET core app, first you need to configure the web app with the following steps:

1. Setup your desired [external social authentication providers](#) in an ASP.NET Core web app.
2. Set the Default Authentication Scheme to `CookieAuthenticationDefaults.AuthenticationScheme` in your `.AddAuthentication()` call.
3. Use `.AddCookie()` in your `Startup.cs` `.AddAuthentication()` call.
4. All providers must be configured with `.SaveTokens = true;`.

```

services.AddAuthentication(o =>
{
    o.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
})
.AddCookie()
.AddFacebook(fb =>
{
    fb.AppId = Configuration["FacebookAppId"];
    fb.AppSecret = Configuration["FacebookAppSecret"];
    fb.SaveTokens = true;
});

```

TIP

If you'd like to include Apple Sign In, you can use the `AspNet.Security.OAuth.Apple` NuGet package. You can view the full [Startup.cs sample](#) in the Essentials GitHub repository.

Add a custom mobile auth controller

With a mobile authentication flow it is usually desirable to initiate the flow directly to a provider that the user has chosen (e.g. by clicking a "Microsoft" button on the sign in screen of the app). It is also important to be able to return relevant information to your app at a specific callback URI to end the authentication flow.

To achieve this, use a custom API Controller:

```
[Route("mobileauth")]
[ApiController]
public class AuthController : ControllerBase
{
    const string callbackScheme = "myapp";

    [HttpGet("{scheme}")]
    public async Task Get([FromRoute]string scheme)
    {
        // 1. Initiate authentication flow with the scheme (provider)
        // 2. When the provider calls back to this URL
        //     a. Parse out the result
        //     b. Build the app callback URL
        //     c. Redirect back to the app
    }
}
```

The purpose of this controller is to infer the scheme (provider) that the app is requesting, and initiate the authentication flow with the social provider. When the provider calls back to the web backend, the controller parses out the result and redirects to the app's callback URI with parameters.

Sometimes you may want to return data such as the provider's `access_token` back to the app which you can do via the callback URI's query parameters. Or, you may want to instead create your own identity on your server and pass back your own token to the app. What and how you do this part is up to you!

Check out the [full controller sample](#) in the Essentials repository.

NOTE

The above sample demonstrates how to return the Access Token from the 3rd party authentication (ie: OAuth) provider. To obtain a token you can use to authorize web requests to the web backend itself, you should create your own token in your web app, and return that instead. The [Overview of ASP.NET Core authentication](#) has more information about advanced authentication scenarios in ASP.NET Core.

API

- [WebAuthenticator source code](#)
- [WebAuthenticator API documentation](#)
- [ASP.NET Core Server Sample](#)

Xamarin.Essentials: Troubleshooting

8/4/2022 • 2 minutes to read • [Edit Online](#)

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 1.3.1 -> Xamarin.Android.Support.CustomTabs 28.0.0.3 ->
Xamarin.Android.Support.Compat (= 28.0.0.3)
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 28.0.0.3 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](#).

Xamarin.Forms local data storage

8/4/2022 • 2 minutes to read • [Edit Online](#)

Files

File handling with Xamarin.Forms can be achieved using code in a .NET Standard library, or by using embedded resources. This article explains how to perform file handling from shared code in a Xamarin.Forms application.

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. This article describes how Xamarin.Forms applications can read and write data to a local SQLite database using SQLite.Net.

File Handling in Xamarin.Forms

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

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);
```

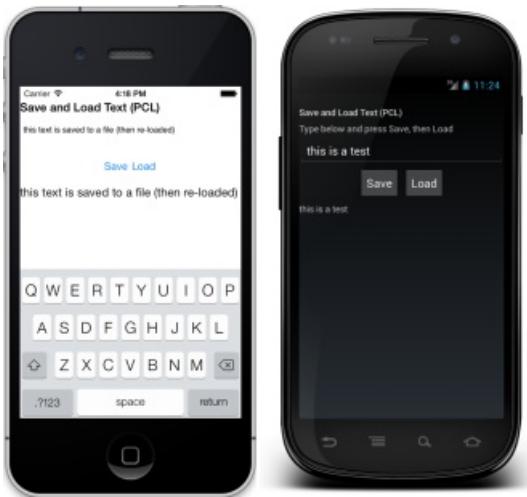
In addition, the `File.Exists` method determines whether the specified file exists:

```
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");
```

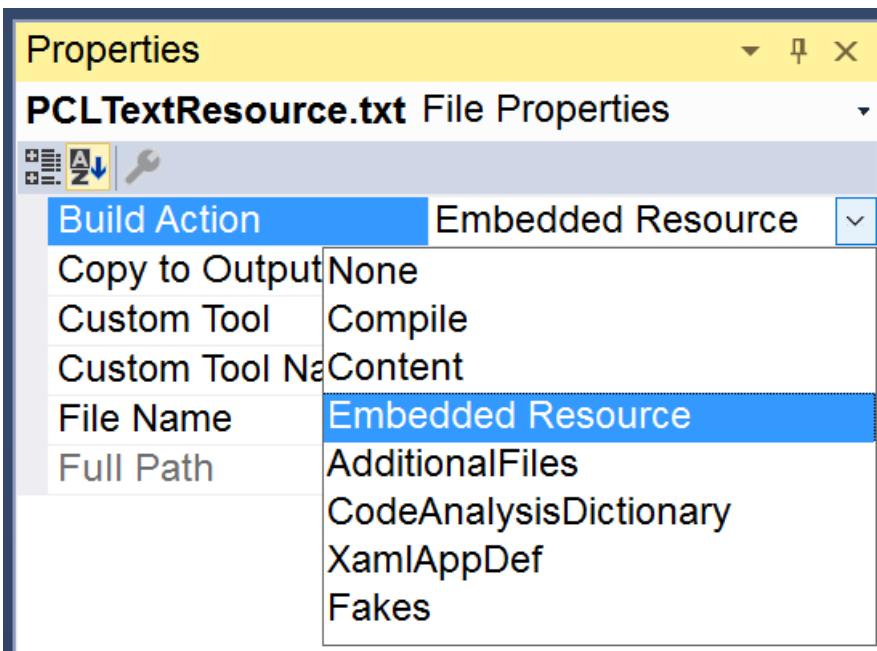
These operations are demonstrated in the sample app, which includes a page that saves and loads text:



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**.

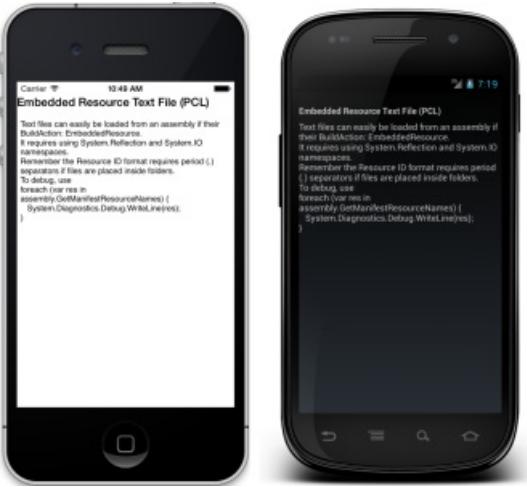
- [Visual Studio](#)
- [Visual Studio for Mac](#)



`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 **LibTextResource.txt**, so the resource ID is `WorkingWithFiles.LibTextResource.txt`.

```
var assembly = IntrospectionExtensions.GetTypeInfo(typeof(LoadResourceText)).Assembly;
Stream stream = assembly.GetManifestResourceStream("WorkingWithFiles.LibTextResource.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.LibXmlResource.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.LibTextResource.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.LibTextResource.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

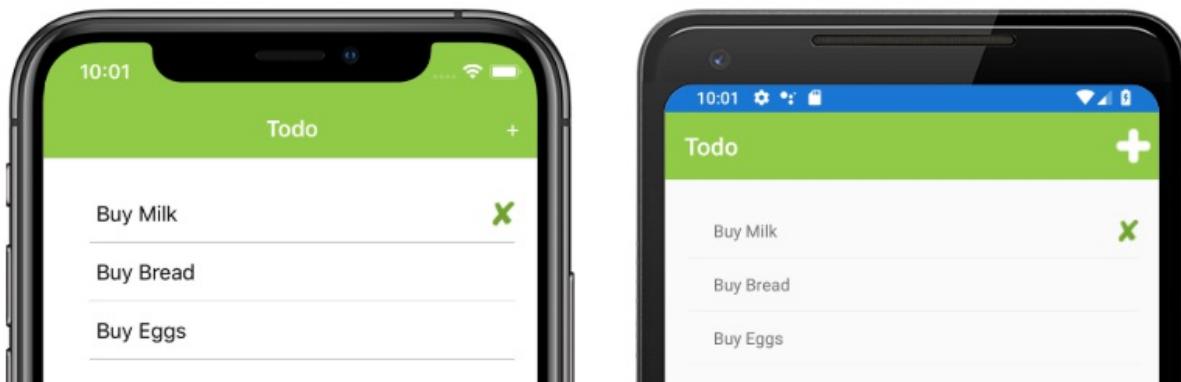
- [FileSample](#)
- [Xamarin.Forms Samples](#)
- [Working with the File System in Xamarin.iOS](#)

Xamarin.Forms Local Databases

8/4/2022 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

The SQLite database engine allows Xamarin.Forms applications to load and save data objects in shared code. The sample application uses a SQLite database table to store todo items. This article describes how to use SQLite.NET in shared code to store and retrieve information in a local database.



Integrate SQLite.NET into mobile apps by following these steps:

1. [Install the NuGet package](#).
2. [Configure constants](#).
3. [Create a database access class](#).
4. [Access data in Xamarin.Forms](#).
5. [Advanced configuration](#).

Install the SQLite NuGet package

Use the NuGet package manager to search for `sqlite-net-pcl` and add the latest version to the shared code project.

There are a number of NuGet packages with similar names. The correct package has these attributes:

- **ID:** sqlite-net-pcl
- **Authors:** SQLite-net
- **Owners:** praeclarum
- **NuGet link:** [sqlite-net-pcl](#)

NOTE

Despite the package name, use the `sqlite-net-pcl` NuGet package even in .NET Standard projects.

Configure app constants

The sample project includes a `Constants.cs` file that provides common configuration data:

```

public static class Constants
{
    public const string DatabaseFilename = "TodoSQLite.db3";

    public const SQLite.SQLiteOpenFlags Flags =
        // open the database in read/write mode
        SQLite.SQLiteOpenFlags.ReadWrite |
        // create the database if it doesn't exist
        SQLite.SQLiteOpenFlags.Create |
        // enable multi-threaded database access
        SQLite.SQLiteOpenFlags.SharedCache;

    public static string DatabasePath
    {
        get
        {
            var basePath = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
            return Path.Combine(basePath, DatabaseFilename);
        }
    }
}

```

The constants file specifies default `SQLiteOpenFlag` enum values that are used to initialize the database connection. The `SQLiteOpenFlag` enum supports these values:

- `Create` : The connection will automatically create the database file if it doesn't exist.
- `FullMutex` : The connection is opened in serialized threading mode.
- `NoMutex` : The connection is opened in multi-threading mode.
- `PrivateCache` : The connection will not participate in the shared cache, even if it's enabled.
- `ReadWrite` : The connection can read and write data.
- `SharedCache` : The connection will participate in the shared cache, if it's enabled.
- `ProtectionComplete` : The file is encrypted and inaccessible while the device is locked.
- `ProtectionCompleteUnlessOpen` : The file is encrypted until it's opened but is then accessible even if the user locks the device.
- `ProtectionCompleteUntilFirstUserAuthentication` : The file is encrypted until after the user has booted and unlocked the device.
- `ProtectionNone` : The database file isn't encrypted.

You may need to specify different flags depending on how your database will be used. For more information about `SQLiteOpenFlags`, see [Opening A New Database Connection](#) on sqlite.org.

Create a database access class

A database wrapper class abstracts the data access layer from the rest of the app. This class centralizes query logic and simplifies the management of database initialization, making it easier to refactor or expand data operations as the app grows. The Todo app defines a `TodoItemDatabase` class for this purpose.

Lazy initialization

The `TodoItemDatabase` uses asynchronous lazy initialization, represented by the custom `AsyncLazy<T>` class, to delay initialization of the database until it's first accessed:

```

public class TodoItemDatabase
{
    static SQLiteAsyncConnection Database;

    public static readonly AsyncLazy<TodoItemDatabase> Instance = new AsyncLazy<TodoItemDatabase>(async () =>
    {
        var instance = new TodoItemDatabase();
        CreateTableResult result = await Database.CreateTableAsync<TodoItem>();
        return instance;
    });

    public TodoItemDatabase()
    {
        Database = new SQLiteAsyncConnection(Constants.DatabasePath, Constants.Flags);
    }

    //...
}

```

The `Instance` field is used to create the database table for the `TodoItem` object, if it doesn't already exist, and returns a `TodoItemDatabase` as a singleton. The `Instance` field, of type `AsyncLazy<TodoItemDatabase>` is constructed the first time it's awaited. If multiple threads attempt to access the field simultaneously, they will all use the single construction. Then, when the construction completes, all `await` operations complete. In addition, any `await` operations after the construction is complete continue immediately since the value is available.

NOTE

The database connection is a static field which ensures that a single database connection is used for the life of the app. Using a persistent, static connection offers better performance than opening and closing connections multiple times during a single app session.

Asynchronous lazy initialization

In order to start the database initialization, avoid blocking execution, and have the opportunity to catch exceptions, the sample application uses asynchronous lazy initialization, represented by the `AsyncLazy<T>` class:

```

public class AsyncLazy<T>
{
    readonly Lazy<Task<T>> instance;

    public AsyncLazy(Func<T> factory)
    {
        instance = new Lazy<Task<T>>(() => Task.Run(factory));
    }

    public AsyncLazy(Func<Task<T>> factory)
    {
        instance = new Lazy<Task<T>>(() => Task.Run(factory));
    }

    public TaskAwaiter<T> GetAwaiter()
    {
        return instance.Value.GetAwaiter();
    }
}

```

The `AsyncLazy` class combines the `Lazy<T>` and `Task<T>` types to create a lazy-initialized task that represents the initialization of a resource. The factory delegate that's passed to the constructor can either be synchronous or asynchronous. Factory delegates will run on a thread pool thread, and will not be executed more than once.

(even when multiple threads attempt to start them simultaneously). When a factory delegate completes, the lazy-initialized value is available, and any methods awaiting the `AsyncLazy<T>` instance receive the value. For more information, see [AsyncLazy](#).

Data manipulation methods

The `TodoItemDatabase` class includes methods for the four types of data manipulation: create, read, edit, and delete. The SQLite.NET library provides a simple Object Relational Map (ORM) that allows you to store and retrieve objects without writing SQL statements.

```
public class TodoItemDatabase
{
    // ...
    public Task<List<TodoItem>> GetItemsAsync()
    {
        return Database.Table<TodoItem>().ToListAsync();
    }

    public Task<List<TodoItem>> GetItemsNotDoneAsync()
    {
        // SQL queries are also possible
        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);
    }
}
```

Access data in Xamarin.Forms

The `TodoItemDatabase` class exposes the `Instance` field, through which the data access operations in the `TodoItemDatabase` class can be invoked:

```
async void OnSaveClicked(object sender, EventArgs e)
{
    var todoItem = (TodoItem)BindingContext;
    TodoItemDatabase database = await TodoItemDatabase.Instance;
    await database.SaveItemAsync(todoItem);

    // Navigate backwards
    await Navigation.PopAsync();
}
```

Advanced configuration

SQLite provides a robust API with more features than are covered in this article and the sample app. The following sections cover features that are important for scalability.

For more information, see [SQLite Documentation](#) on sqlite.org.

Write-ahead logging

By default, SQLite uses a traditional rollback journal. A copy of the unchanged database content is written into a separate rollback file, then the changes are written directly to the database file. The COMMIT occurs when the rollback journal is deleted.

Write-Ahead Logging (WAL) writes changes into a separate WAL file first. In WAL mode, a COMMIT is a special record, appended to the WAL file, which allows multiple transactions to occur in a single WAL file. A WAL file is merged back into the database file in a special operation called a *checkpoint*.

WAL can be faster for local databases because readers and writers do not block each other, allowing read and write operations to be concurrent. However, WAL mode doesn't allow changes to the *page size*, adds additional file associations to the database, and adds the extra *checkpointing* operation.

To enable WAL in SQLite.NET, call the `EnableWriteAheadLoggingAsync` method on the `SQLiteAsyncConnection` instance:

```
await Database.EnableWriteAheadLoggingAsync();
```

For more information, see [SQLite Write-Ahead Logging](#) on sqlite.org.

Copy a database

There are several cases where it may be necessary to copy a SQLite database:

- A database has shipped with your application but must be copied or moved to writeable storage on the mobile device.
- You need to make a backup or copy of the database.
- You need to version, move, or rename the database file.

In general, moving, renaming, or copying a database file is the same process as any other file type with a few additional considerations:

- All database connections should be closed before attempting to move the database file.
- If you use [Write-Ahead Logging](#), SQLite will create a Shared Memory Access (.shm) file and a (Write Ahead Log) (.wal) file. Ensure that you apply any changes to these files as well.

For more information, see [File Handling in Xamarin.Forms](#).

Related links

- [Todo sample application](#)
- [SQLite.NET NuGet package](#)
- [SQLite documentation](#)
- [Using SQLite with Android](#)
- [Using SQLite with iOS](#)
- [AsyncLazy](#)

Xamarin.Forms and Azure Services

8/4/2022 • 2 minutes to read • [Edit Online](#)

Azure Mobile Apps

Azure Mobile Apps provide authentication, data querying, and offline data synchronization functionality to your mobile app.

Consume an Azure Cosmos DB Document Database in Xamarin.Forms

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.

Send and receive Push Notifications with Azure Notification Hubs and Xamarin.Forms

Azure Notification Hubs enable you to centralize notifications across platforms so your backend application can communicate with a single hub. Azure Notification Hubs take care of distributing push notifications to multiple platform providers. This article explains how to integrate Azure Notification Hubs into a Xamarin.Forms application.

Store and Access Data in Azure Storage from Xamarin.Forms

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.

Search Data with Azure Search and Xamarin.Forms

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.

Azure Functions with Xamarin.Forms

This article demonstrates how to build your first Azure Function that interacts with Xamarin.Forms.

Consume an Azure Cosmos DB Document Database in Xamarin.Forms

8/4/2022 • 7 minutes to read • [Edit Online](#)

 [Download the sample](#)

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 video

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](#).

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

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](#)

Store and Access Data in Azure Storage from Xamarin.Forms

8/4/2022 • 9 minutes to read • [Edit Online](#)

 [Download the sample](#)

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.

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 account 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.

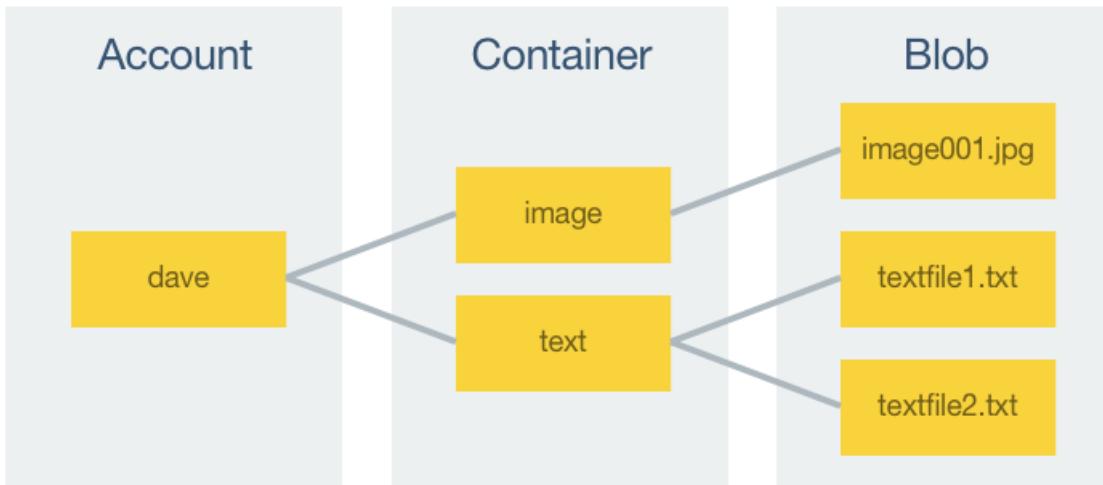
For more information about Azure Storage, see [Introduction to Storage](#).

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

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.

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.

Related Links

- [Azure Storage \(sample\)](#)
- [Introduction to Storage](#).
- [How to use Blob Storage from Xamarin](#)
- [Using Shared Access Signatures \(SAS\)](#)

- Windows Azure Storage (NuGet)

Search Data with Azure Search and Xamarin.Forms

8/4/2022 • 10 minutes to read • [Edit Online](#)



[Download the sample](#)

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](#).

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

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`.
- `IsSortable` – 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:

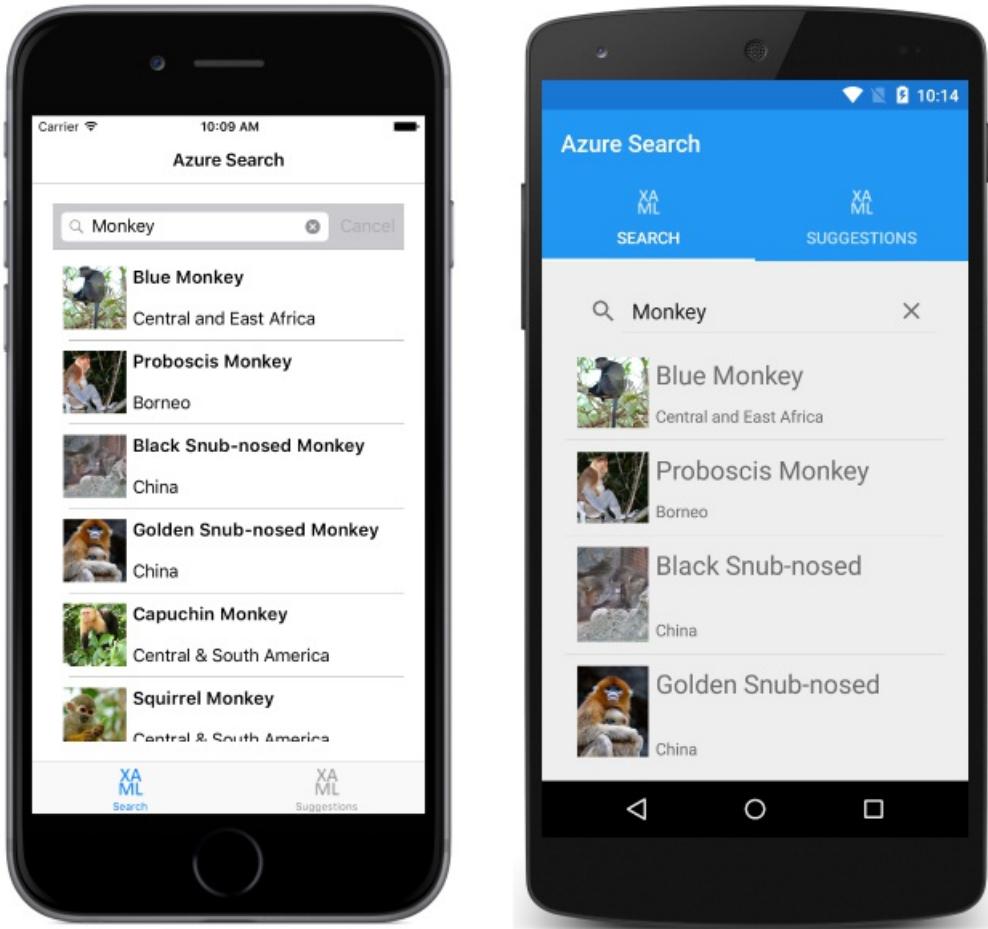
```
var parameters = new SearchParameters
{
    Filter = "location ne 'China' and location ne 'Vietnam'"
};
var searchResults = await indexClient.Documents.SearchAsync<Monkey>(text, parameters);
```

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
        });
    }
}

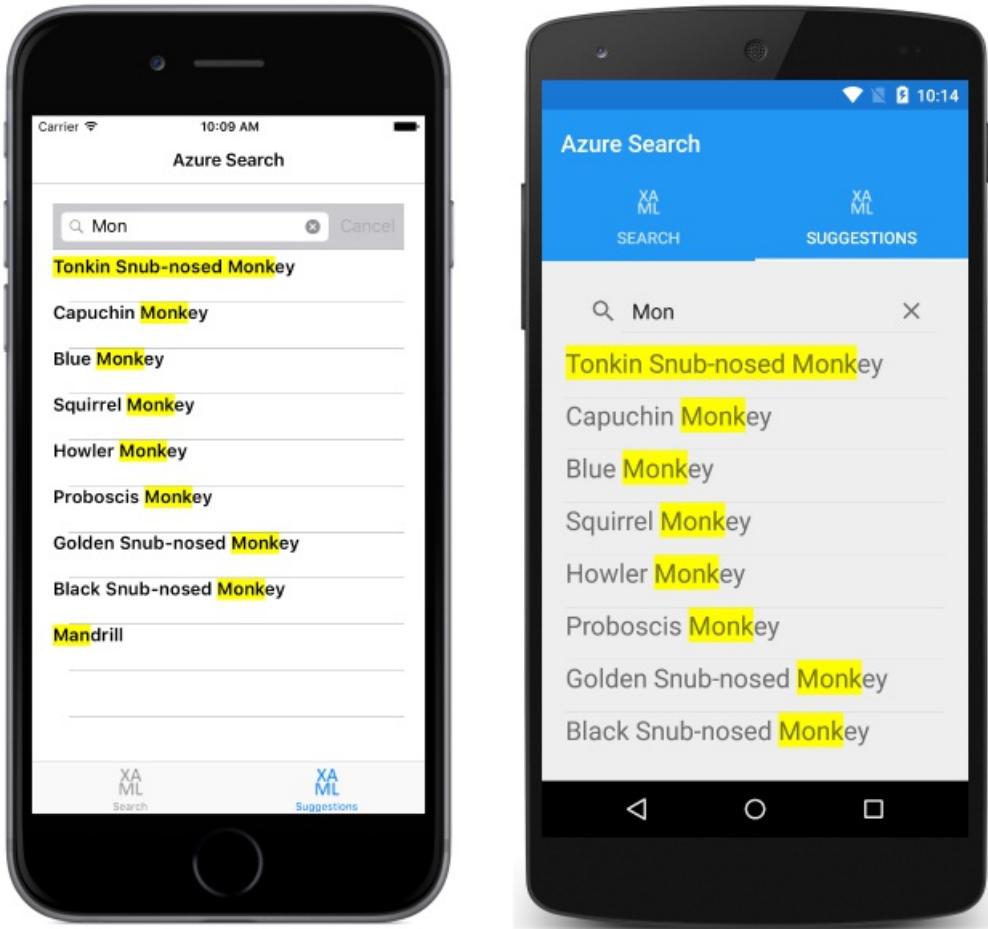
```

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](#)

Get started with Azure Functions

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

Get started building your first Azure Function that interacts with Xamarin.Forms.

- [Visual Studio 2019](#)
- [Visual Studio 2017](#)
- [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](#).

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

Related Links

- [Azure Functions docs](#)
- [Implementing a simple Azure Function with a Xamarin.Forms client \(sample\)](#)

Xamarin.Forms and Azure Cognitive Services

8/4/2022 • 2 minutes to read • [Edit Online](#)

Introduction

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 from Xamarin.Forms applications.

Speech Recognition

Azure Speech Service is a cloud-based API that provides algorithms to process spoken language. This article explains how to use the Azure Speech Service to transcribe speech to text in a Xamarin.Forms application.

Spell Check

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.

Text Translation

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.

Perceived Emotion Recognition

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.

Xamarin.Forms and Azure Cognitive Services Introduction

8/4/2022 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

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:

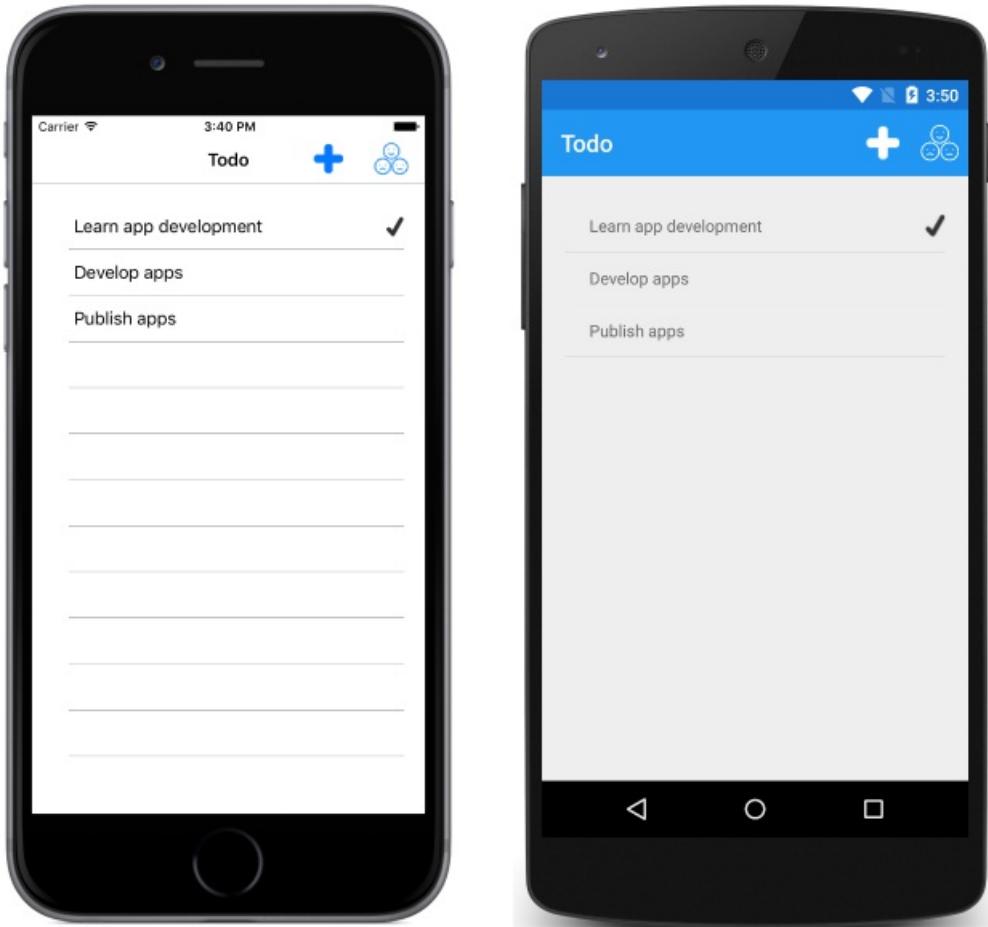
- View a list of tasks.
- Add and edit tasks through the soft keyboard, or by performing speech recognition with the Microsoft Speech API.
- Spell check tasks using the Bing Spell Check API. For more information, see [Spell Checking using the Bing Spell Check API](#).
- Translate tasks from English to 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](#).

WARNING

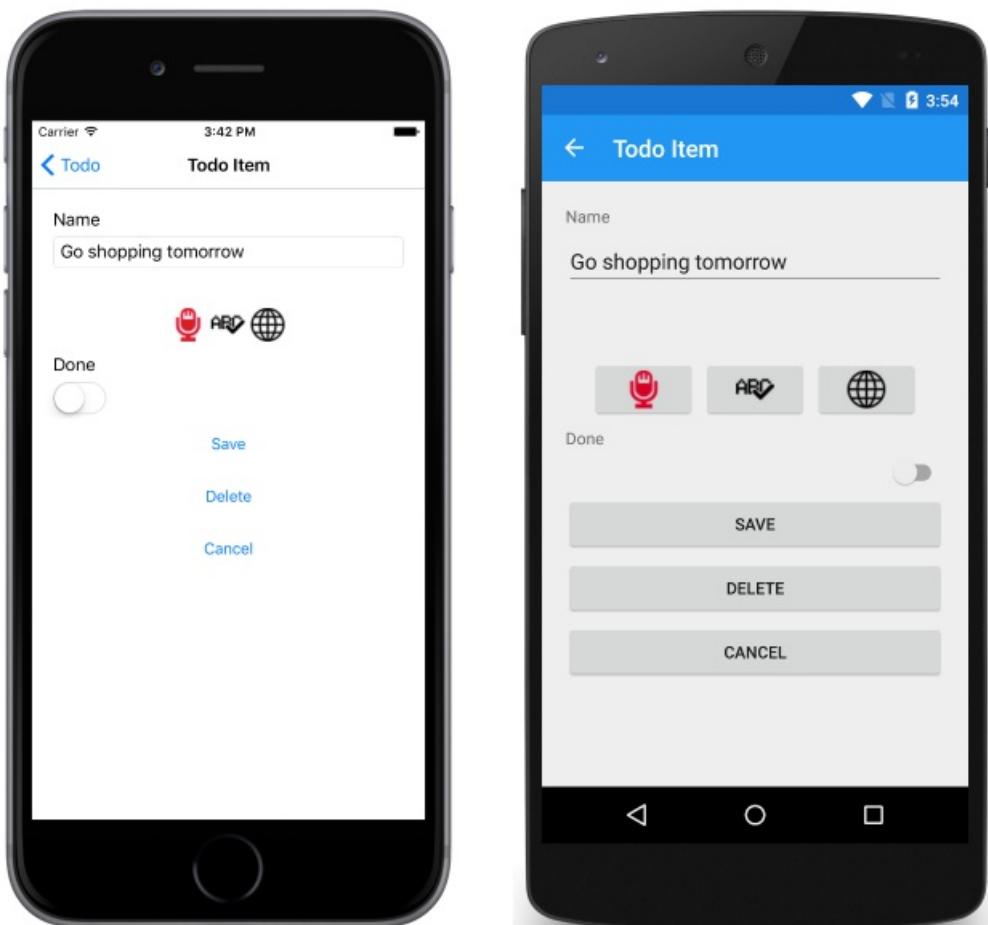
The Bing Speech API has been deprecated in favor of the Azure Speech Service. For a sample dedicated to Azure Speech Service, see [Speech recognition with the Speech Service API](#).

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:

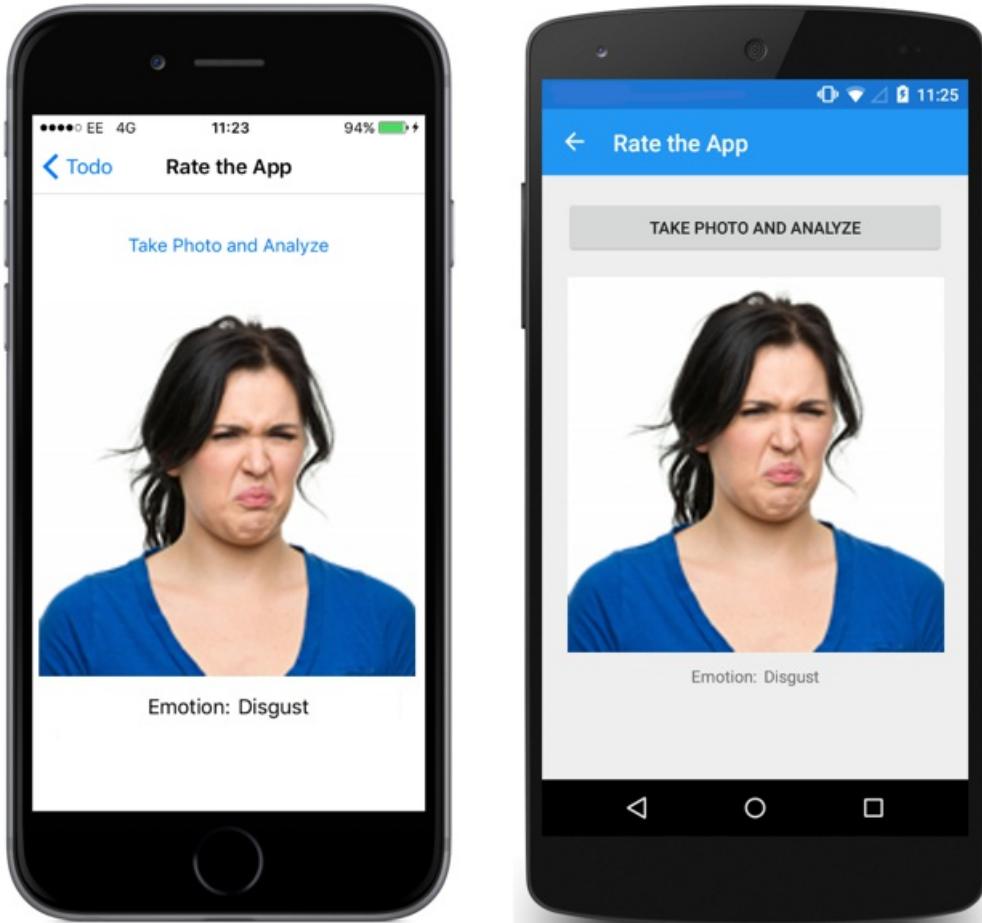


New items can be created by clicking on the + button, which navigates to the `TodoItemPage`. This page can also be navigated to by selecting a task:



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.

Understand the application anatomy

The shared code 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 shared code 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.

Model 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.

Invoke 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 shared code 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.

Invoke 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

- [Speech recognition with the Speech Service API](#)
- [Microsoft Cognitive Services Documentation](#)
- [Todo Cognitive Services \(sample\)](#)

Speech recognition using Azure Speech Service

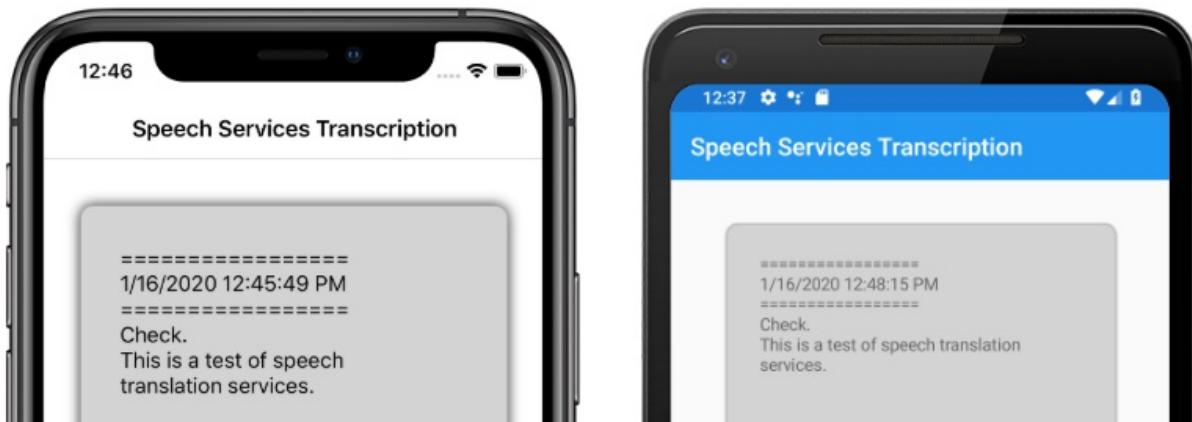
8/4/2022 • 8 minutes to read • [Edit Online](#)

 [Download the sample](#)

Azure Speech Service is a cloud-based API that offers the following functionality:

- **Speech-to-text** transcribes audio files or streams to text.
- **Text-to-speech** converts input text into human-like synthesized speech.
- **Speech translation** enables real-time, multi-language translation for both speech-to-text and speech-to-speech.
- **Voice assistants** can create human-like conversation interfaces for applications.

This article explains how speech-to-text is implemented in the sample Xamarin.Forms application using the Azure Speech Service. The following screenshots show the sample application on iOS and Android:



Create an Azure Speech Service resource

Azure Speech Service is part of Azure Cognitive Services, which provides cloud-based APIs for tasks such as image recognition, speech recognition and translation, and Bing search. For more information, see [What are Azure Cognitive Services?](#).

The sample project requires an Azure Cognitive Services resource to be created in your Azure portal. A Cognitive Services resource can be created for a single service, such as Speech Service, or as a multi-service resource. The steps to create a Speech Service resource are as follows:

1. Log into your [Azure portal](#).
2. Create a multi-service or single-service resource.
3. Obtain the API key and region information for your resource.
4. Update the sample `Constants.cs` file.

For a step-by-step guide to creating a resource, see [Create a Cognitive Services resource](#).

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin. Once you have an account, a single-service resource can be created at the free tier to try out the service.

Configure your app with the Speech Service

After creating a Cognitive Services resource, the `Constants.cs` file can be updated with the region and API key from your Azure resource:

```
public static class Constants
{
    public static string CognitiveServicesApiKey = "YOUR_KEY_Goes_Here";
    public static string CognitiveServicesRegion = "westus";
}
```

Install NuGet Speech Service package

The sample application uses the `Microsoft.CognitiveServices.Speech` NuGet package to connect to the Azure Speech Service. Install this NuGet package in the shared project and each platform project.

Create an IMicrophoneService interface

Each platform requires permission to access to the microphone. The sample project provides an `IMicrophoneService` interface in the shared project, and uses the `Xamarin.Forms DependencyService` to obtain platform implementations of the interface.

```
public interface IMicrophoneService
{
    Task<bool> GetPermissionAsync();
    void OnRequestPermissionsResult(bool isGranted);
}
```

Create the page layout

The sample project defines a basic page layout in the `MainPage.xaml` file. The key layout elements are a `Button` that starts the transcription process, a `Label` to contain the transcribed text, and an `ActivityIndicator` to show when transcription is in progress:

```
<ContentPage ...>
    <StackLayout>
        <Frame ...>
            <ScrollView x:Name="scroll"
                ...
                <Label x:Name="transcribedText"
                    ...
                    </Label>
                </ScrollView>
            </Frame>

            <ActivityIndicator x:Name="transcribingIndicator"
                IsRunning="False" />
            <Button x:Name="transcribeButton"
                ...
                Clicked="TranscribeClicked"/>
        </StackLayout>
    </ContentPage>
```

Implement the Speech Service

The `MainPage.xaml.cs` code-behind file contains all of the logic to send audio and receive transcribed text from the Azure Speech Service.

The `MainPage` constructor gets an instance of the `IMicrophoneService` interface from the `DependencyService`:

```
public partial class MainPage : ContentPage
{
    SpeechRecognizer recognizer;
    IMicrophoneService micService;
    bool isTranscribing = false;

    public MainPage()
    {
        InitializeComponent();

        micService = DependencyService.Resolve<IMicrophoneService>();
    }

    // ...
}
```

The `TranscribeClicked` method is called when the `transcribeButton` instance is tapped:

```

async void TranscribeClicked(object sender, EventArgs e)
{
    bool isMicEnabled = await micService.GetPermissionAsync();

    // EARLY OUT: make sure mic is accessible
    if (!isMicEnabled)
    {
        UpdateTranscription("Please grant access to the microphone!");
        return;
    }

    // initialize speech recognizer
    if (recognizer == null)
    {
        var config = SpeechConfig.FromSubscription(Constants.CognitiveServicesApiKey,
        Constants.CognitiveServicesRegion);
        recognizer = new SpeechRecognizer(config);
        recognizer.Recognized += (obj, args) =>
        {
            UpdateTranscription(args.Result.Text);
        };
    }

    // if already transcribing, stop speech recognizer
    if (isTranscribing)
    {
        try
        {
            await recognizer.StopContinuousRecognitionAsync();
        }
        catch(Exception ex)
        {
            UpdateTranscription(ex.Message);
        }
        isTranscribing = false;
    }

    // if not transcribing, start speech recognizer
    else
    {
        Device.BeginInvokeOnMainThread(() =>
        {
            InsertDateTimeRecord();
        });
        try
        {
            await recognizer.StartContinuousRecognitionAsync();
        }
        catch(Exception ex)
        {
            UpdateTranscription(ex.Message);
        }
        isTranscribing = true;
    }
    UpdateDisplayState();
}

```

The `TranscribeClicked` method does the following:

1. Checks if the application has access to the microphone and exits early if it does not.
2. Creates an instance of `SpeechRecognizer` class if it doesn't already exist.
3. Stops continuous transcription if it is in progress.
4. Inserts a timestamp and starts continuous transcription if it is not in progress.
5. Notifies the application to update its appearance based on the new application state.

The remainder of the `MainPage` class methods are helpers for displaying the application state:

```
void UpdateTranscription(string newText)
{
    Device.BeginInvokeOnMainThread(() =>
    {
        if (!string.IsNullOrWhiteSpace(newText))
        {
            transcribedText.Text += $"{newText}\n";
        }
    });
}

void InsertDateTimeRecord()
{
    var msg = $"=====\\n{DateTime.Now.ToString()}\\n=====";
    UpdateTranscription(msg);
}

void UpdateDisplayState()
{
    Device.BeginInvokeOnMainThread(() =>
    {
        if (isTranscribing)
        {
            transcribeButton.Text = "Stop";
            transcribeButton.BackgroundColor = Color.Red;
            transcribingIndicator.IsRunning = true;
        }
        else
        {
            transcribeButton.Text = "Transcribe";
            transcribeButton.BackgroundColor = Color.Green;
            transcribingIndicator.IsRunning = false;
        }
    });
}
```

The `UpdateTranscription` method writes the provided `newText` `string` to the `Label` element named `transcribedText`. It forces this update to happen on the UI thread so it can be called from any context without causing exceptions. The `InsertDateTimeRecord` writes the current date and time to the `transcribedText` instance to mark the start of a new transcription. Finally, the `UpdateDisplayState` method updates the `Button` and `ActivityIndicator` elements to reflect whether or not transcription is in progress.

Create platform microphone services

The application must have microphone access to collect speech data. The `IMicrophoneService` interface must be implemented and registered with the `DependencyService` on each platform for the application to function.

Android

The sample project defines an `IMicrophoneService` implementation for Android called `AndroidMicrophoneService`:

```

[assembly: Dependency(typeof(AndroidMicrophoneService))]
namespace CognitiveSpeechService.Droid.Services
{
    public class AndroidMicrophoneService : IMicrophoneService
    {
        public const int RecordAudioPermissionCode = 1;
        private TaskCompletionSource<bool> tcsPermissions;
        string[] permissions = new string[] { Manifest.Permission.RecordAudio };

        public Task<bool> GetPermissionAsync()
        {
            tcsPermissions = new TaskCompletionSource<bool>();

            if ((int)Build.VERSION.SdkInt < 23)
            {
                tcsPermissions.TrySetResult(true);
            }
            else
            {
                var currentActivity = MainActivity.Instance;
                if (ActivityCompat.CheckSelfPermission(currentActivity, Manifest.Permission.RecordAudio) != (int)Permission.Granted)
                {
                    RequestMicPermissions();
                }
                else
                {
                    tcsPermissions.TrySetResult(true);
                }
            }

            return tcsPermissions.Task;
        }

        public void OnRequestPermissionsResult(bool isGranted)
        {
            tcsPermissions.TrySetResult(isGranted);
        }

        void RequestMicPermissions()
        {
            if (ActivityCompat.ShouldShowRequestPermissionRationale(MainActivity.Instance, Manifest.Permission.RecordAudio))
            {
                Snackbar.Make(MainActivity.Instance.FindViewById(Android.Resource.Id.Content),
                    "Microphone permissions are required for speech transcription!",
                    Snackbar.LengthIndefinite)
                    .SetAction("Ok", v =>
                {
                    ((Activity)MainActivity.Instance).RequestPermissions(permissions,
RecordAudioPermissionCode);
                })
                    .Show();
            }
            else
            {
                ActivityCompat.RequestPermissions((Activity)MainActivity.Instance, permissions,
RecordAudioPermissionCode);
            }
        }
    }
}

```

The `AndroidMicrophoneService` has the following features:

1. The `Dependency` attribute registers the class with the `DependencyService`.
2. The `GetPermissionAsync` method checks if permissions are required based on the Android SDK version, and calls `RequestMicPermissions` if permission has not already been granted.
3. The `RequestMicPermissions` method uses the `Snackbar` class to request permissions from the user if a rationale is required, otherwise it directly requests audio recording permissions.
4. The `OnRequestPermissionsResult` method is called with a `bool` result once the user has responded to the permissions request.

The `MainActivity` class is customized to update the `AndroidMicrophoneService` instance when permissions requests are complete:

```
public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
{
    IMicrophoneService micService;
    internal static MainActivity Instance { get; private set; }

    protected override void OnCreate(Bundle savedInstanceState)
    {
        Instance = this;
        // ...
        micService = DependencyService.Resolve<IMicrophoneService>();
    }
    public override void OnRequestPermissionsResult(int requestCode, string[] permissions, [GeneratedEnum] Android.Content.PM.Permission[] grantResults)
    {
        // ...
        switch(requestCode)
        {
            case AndroidMicrophoneService.RecordAudioPermissionCode:
                if (grantResults[0] == Permission.Granted)
                {
                    micService.OnRequestPermissionsResult(true);
                }
                else
                {
                    micService.OnRequestPermissionsResult(false);
                }
                break;
        }
    }
}
```

The `MainActivity` class defines a static reference called `Instance`, which is required by the `AndroidMicrophoneService` object when requesting permissions. It overrides the `OnRequestPermissionsResult` method to update the `AndroidMicrophoneService` object when the permissions request is approved or denied by the user.

Finally, the Android application must include the permission to record audio in the `AndroidManifest.xml` file:

```
<manifest ...>
    ...
    <uses-permission android:name="android.permission.RECORD_AUDIO" />
</manifest>
```

iOS

The sample project defines an `IMicrophoneService` implementation for iOS called `iOSMicrophoneService`:

```
[assembly: Dependency(typeof(iOSMicrophoneService))]
namespace CognitiveSpeechService.iOS.Services
{
    public class iOSMicrophoneService : IMicrophoneService
    {
        TaskCompletionSource<bool> tcsPermissions;

        public Task<bool> GetPermissionAsync()
        {
            tcsPermissions = new TaskCompletionSource<bool>();
            RequestMicPermission();
            return tcsPermissions.Task;
        }

        public void OnRequestPermissionsResult(bool isGranted)
        {
            tcsPermissions.TrySetResult(isGranted);
        }

        void RequestMicPermission()
        {
            var session = AVAudioSession.SharedInstance();
            session.RequestRecordPermission((granted) =>
            {
                tcsPermissions.TrySetResult(granted);
            });
        }
    }
}
```

The `iOSMicrophoneService` has the following features:

1. The `Dependency` attribute registers the class with the `DependencyService`.
2. The `GetPermissionAsync` method calls `RequestMicPermissions` to request permissions from the device user.
3. The `RequestMicPermissions` method uses the shared `AVAudioSession` instance to request recording permissions.
4. The `OnRequestPermissionsResult` method updates the `TaskCompletionSource` instance with the provided `bool` value.

Finally, the iOS app `Info.plist` must include a message that tells the user why the app is requesting access to the microphone. Edit the `Info.plist` file to include the following tags within the `<dict>` element:

```
<plist>
<dict>
    ...
    <key>NSMicrophoneUsageDescription</key>
    <string>Voice transcription requires microphone access</string>
</dict>
</plist>
```

UWP

The sample project defines an `IMicrophoneService` implementation for UWP called `UWPMicrophoneService`:

```

[assembly: Dependency(typeof(UWPMicrophoneService))]
namespace CognitiveSpeechService.UWP.Services
{
    public class UWPMicrophoneService : IMicrophoneService
    {
        public async Task<bool> GetPermissionAsync()
        {
            bool isMicAvailable = true;
            try
            {
                var mediaCapture = new MediaCapture();
                var settings = new MediaCaptureInitializationSettings();
                settings.StreamingCaptureMode = StreamingCaptureMode.Audio;
                await mediaCapture.InitializeAsync(settings);
            }
            catch(Exception ex)
            {
                isMicAvailable = false;
            }

            if(!isMicAvailable)
            {
                await Windows.System.Launcher.LaunchUriAsync(new Uri("ms-settings:privacy-microphone"));
            }

            return isMicAvailable;
        }

        public void OnRequestPermissionsResult(bool isGranted)
        {
            // intentionally does nothing
        }
    }
}

```

The `UWPMicrophoneService` has the following features:

1. The `Dependency` attribute registers the class with the `DependencyService`.
2. The `GetPermissionAsync` method attempts to initialize a `MediaCapture` instance. If that fails, it launches a user request to enable the microphone.
3. The `OnRequestPermissionsResult` method exists to satisfy the interface but is not required for the UWP implementation.

Finally, the **UWP Package.appxmanifest** must specify that the application uses the microphone. Double-click the **Package.appxmanifest** file and select the **Microphone** option on the **Capabilities** tab in Visual Studio 2019:



Test the application

Run the app and click the **Transcribe** button. The app should request microphone access and begin the transcription process. The `ActivityIndicator` will animate, showing that transcription is active. As you speak, the app will stream audio data to the Azure Speech Services resource, which will respond with transcribed text. The transcribed text will appear in the `Label` element as it is received.

NOTE

Android emulators fail to load and initialize the Speech Service libraries. Testing on a physical device is recommended for the Android platform.

Related links

- [Azure Speech Service sample](#)
- [Azure Speech Service overview](#)
- [Create a Cognitive Services resource](#)
- [Quickstart: Recognize speech from a microphone](#)

Spell Checking Using the Bing Spell Check API

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

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

NOTE

The example above uses `Replace` for simplicity, but across a large amount of text it could replace the wrong token. The API provides the `offset` value which should be used in production apps to identify the correct location in the source text to perform an update.

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](#)
- [Consume a RESTful Web Service](#)
- [Todo Cognitive Services \(sample\)](#)
- [Bing Spell Check API v7 reference](#)

Text Translation Using the Translator API

8/4/2022 • 4 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

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](#).

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 [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.

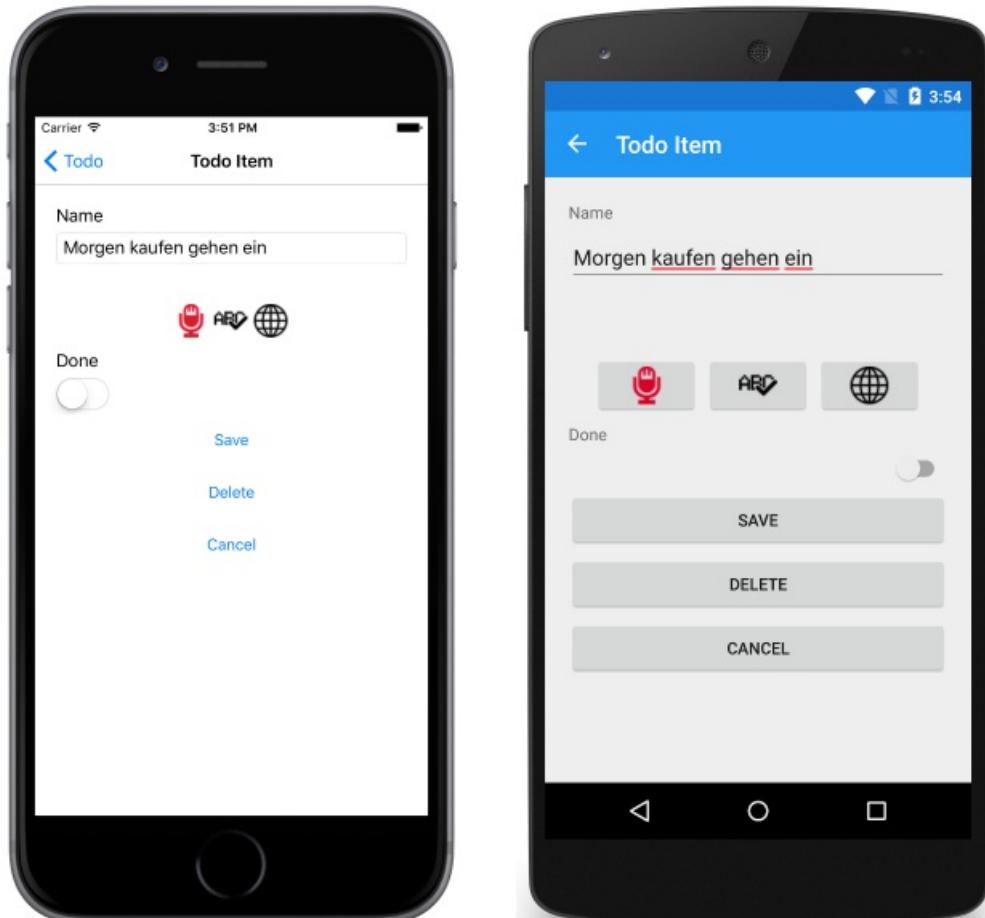
The `translate` API will send HTTP status code 200 (OK) in the response, provided that the request is valid, which indicates that the request succeeded and that the requested information is in the response. For a list of possible error responses, see Response Messages at [GET Translate](#).

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](#)
- [Consume a RESTful Web Service](#)
- [Todo Cognitive Services \(sample\)](#)
- [Translator Text API](#)

Perceived Emotion Recognition Using the Face API

8/4/2022 • 5 minutes to read • [Edit Online](#)

 [Download the sample](#)

The Face API can perform emotion detection to detect anger, contempt, disgust, fear, happiness, neutral, sadness, and surprise, in a facial expression based on perceived annotations by human coders. It is important to note, however, that facial expressions alone may not necessarily represent the internal states of people.

In addition to returning an emotion result for a facial expression, the Face API can also returns a bounding box for detected faces.

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](#).

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

An API key must be obtained to use the Face API. This can be obtained at [Try Cognitive Services](#).

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.

Perform 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.
```

Send 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](#).

Process 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:

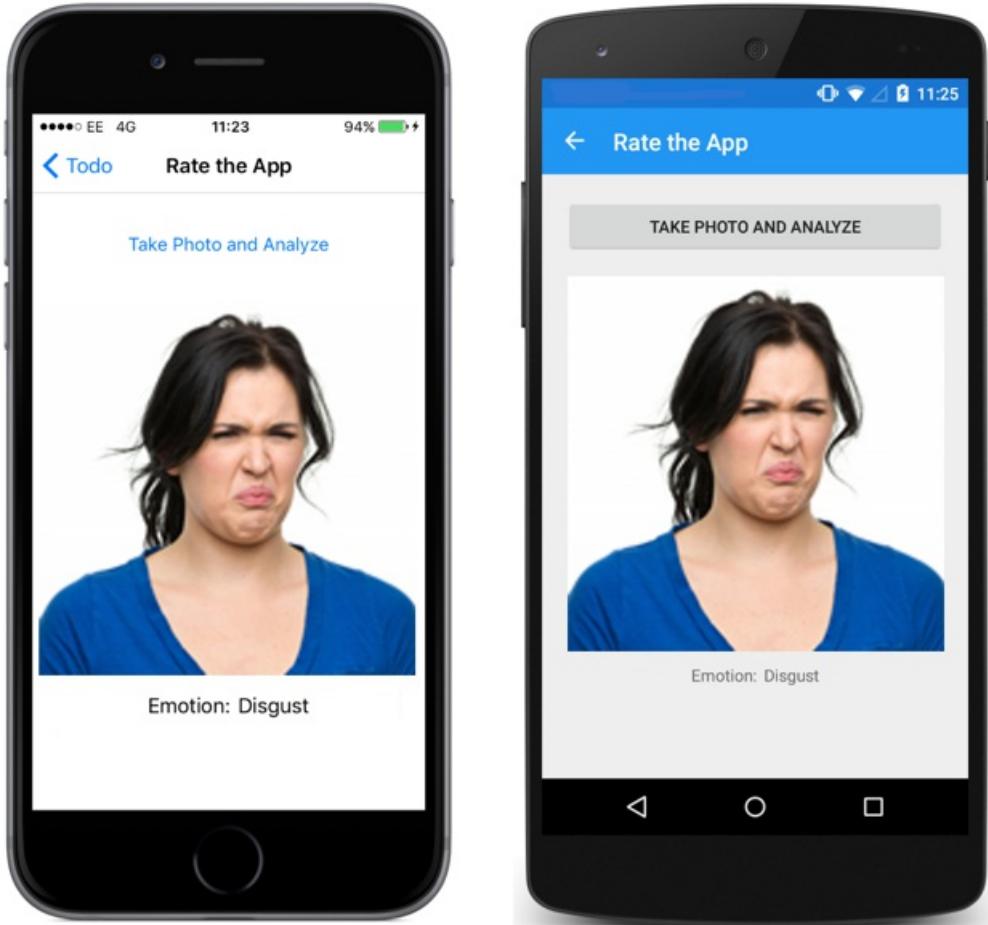
```
[
  {
    "faceId": "8a1a80fe-1027-48cf-a7f0-e61c0f005051",
    "faceRectangle": {
      "top": 192,
      "left": 164,
      "width": 339,
      "height": 339
    },
    "faceAttributes": {
      "emotion": {
        "anger": 0.0,
        "contempt": 0.0,
        "disgust": 0.0,
        "fear": 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:



Related links

- [Face API](#).
- [Todo Cognitive Services \(sample\)](#)
- [Face REST API](#)

Xamarin.Forms and Web Services

8/4/2022 • 2 minutes to read • [Edit Online](#)

Introduction

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.

Consume 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.

Consume 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.

Consume 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.

Xamarin.Forms Web Services Introduction

8/4/2022 • 3 minutes to read • [Edit Online](#)

 [Download the sample](#)

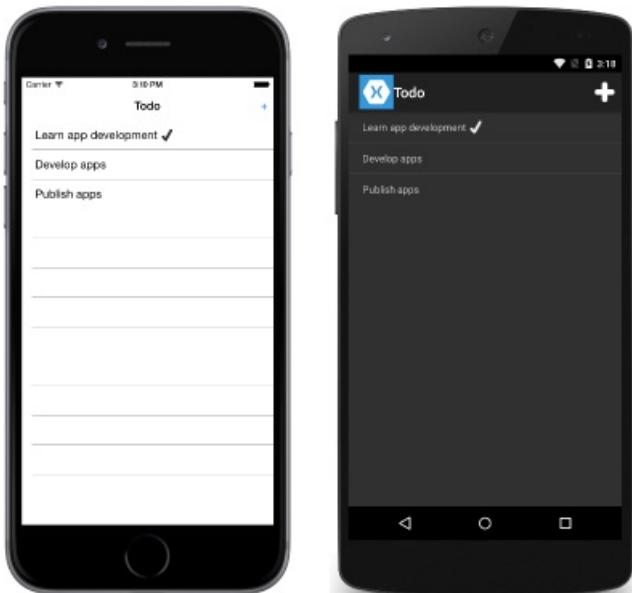
This topic provides a walkthrough of the `Xamarin.Forms` sample application that demonstrates how to communicate with different web services. While each web service uses a separate sample application, they are functionally similar and share common classes.

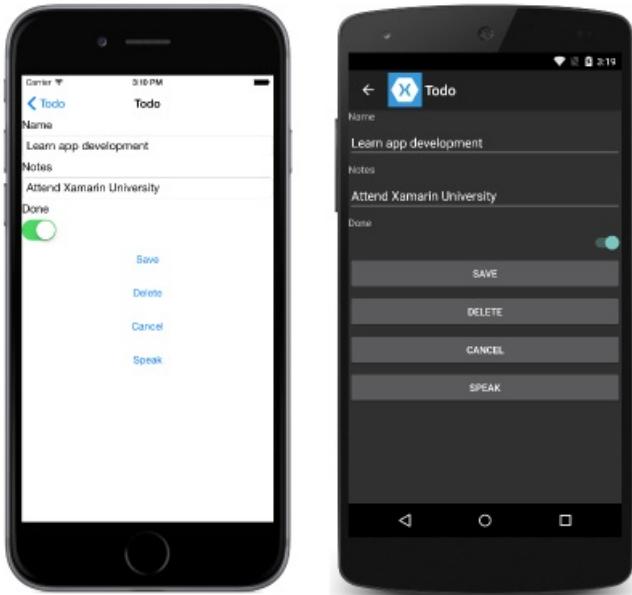
The sample to-do list application described below is used to demonstrate how to access different types of web service backends with `Xamarin.Forms`. It provides functionality to:

- 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.

Understand the application anatomy

The shared code 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 shared code 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.

View 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.

Model 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.

Invoke 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.

Related links

- [ASMX \(sample\)](#)

- [WCF \(sample\)](#)
- [REST \(sample\)](#)

Consume an ASP.NET Web Service (ASMX)

8/4/2022 • 6 minutes to read • [Edit Online](#)

 [Download the sample](#)

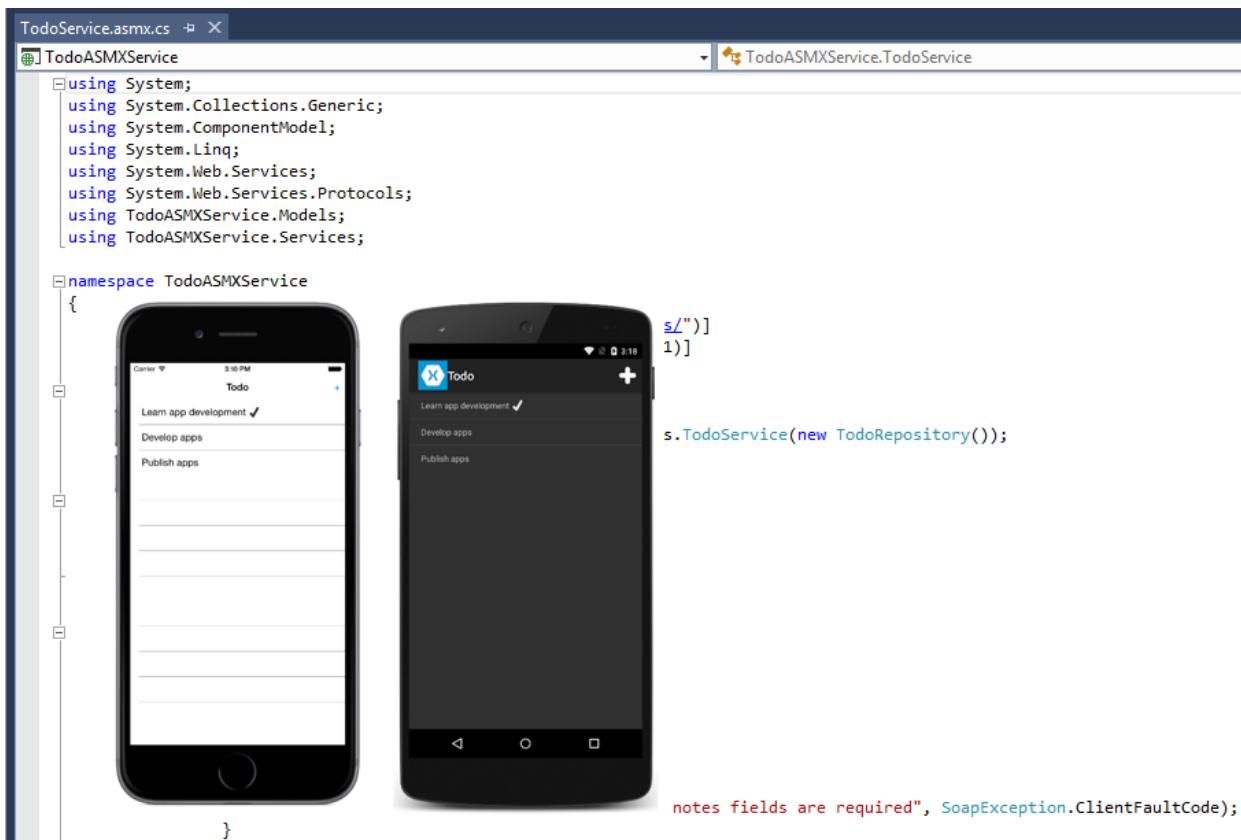
ASMX provides the ability to build web services that send messages using the Simple Object Access Protocol (SOAP). SOAP is a platform-independent and language-independent protocol for building and accessing web services. Consumers of an ASMX service do not need to know anything about the platform, object 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.

This sample includes the mobile applications that run on physical or emulated devices, and an ASMX service that provides methods to get, add, edit, and delete data. When the mobile applications are run, they connect to the locally-hosted ASMX service as shown in the following screenshot:



```
TodoService.asmx.cs X TodoASMXService
TodoASMXService.cs TodoASMXService.TodoService

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Web.Services;
using System.Web.Services.Protocols;
using TodoASMXService.Models;
using TodoASMXService.Services;

namespace TodoASMXService
{
    public class TodoService : SoapService
    {
        public TodoService()
        {
            s = new TodoService(new TodoRepository());
        }

        [SoapMethod("GetTodos")]
        public List<Todo> GetTodos()
        {
            return s.GetTodos();
        }

        [SoapMethod("AddTodo")]
        public void AddTodo(Todo todo)
        {
            s.AddTodo(todo);
        }

        [SoapMethod("UpdateTodo")]
        public void UpdateTodo(Todo todo)
        {
            s.UpdateTodo(todo);
        }

        [SoapMethod("DeleteTodo")]
        public void DeleteTodo(int id)
        {
            s.DeleteTodo(id);
        }
    }
}
```

The screenshot shows the Xamarin IDE with the code editor open to `TodoService.asmx.cs`. The code defines a `TodoASMXService` with a `TodoService` method that initializes a `TodoRepository`. Below the code editor are two mobile device emulators. The left emulator shows a list of todos with items like "Learn app development" and "Develop apps". The right emulator shows a blank screen with a plus sign button. At the bottom of the code editor, there is a note in red: "notes fields are required", `SoapException.ClientFaultCode`;

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](#).

Consume 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](#).

Create the TodoService proxy

A proxy class, called `TodoService`, extends `SoapHttpClientProtocol` and provides methods for communicating with the ASMX service over HTTP. The proxy is generated by adding a web reference to each platform-specific project in Visual Studio 2019 or Visual Studio 2017. The web reference generates methods and events for each action defined in the service's Web Services Description Language (WSDL) document.

For example, the `GetTodoItems` service action results in a `GetTodoItemsAsync` method and a `GetTodoItemsCompleted` event in the proxy. The generated method has a void return type and invokes the `GetTodoItems` action on the parent `SoapHttpClientProtocol` class. When the invoked method receives a response from the service, it fires the `GetTodoItemsCompleted` event and provides the response data within the event's `Result` property.

Create the ISoapService implementation

To enable the shared, cross-platform project to work with the service, the sample defines the `ISoapService` interface, which follows the [Task asynchronous programming model in C#](#). Each platform implements the `ISoapService` to expose the platform-specific proxy. The sample uses `TaskCompletionSource` objects to expose the proxy as a task asynchronous interface. Details on using `TaskCompletionSource` are found in the implementations of each action type in the sections below.

The sample `SoapService`:

1. Instantiates the `TodoService` as a class-level instance
2. Creates a collection called `Items` to store `TodoItem` objects
3. Specifies a custom endpoint for the optional `Url` property on the `TodoService`

```

public class SoapService : ISoapService
{
    ASMXService.TodoService todoService;
    public List<TodoItem> Items { get; private set; } = new List<TodoItem>();

    public SoapService ()
    {
        todoService = new ASMXService.TodoService ();
        todoService.Url = Constants.SapUrl;
        ...
    }
}

```

Create 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 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 retrieves the data from the proxy generated `TodoItem` type and sets it in the newly created `TodoItem` instance.

Retrieve data

The `ISoapService` interface expects the `RefreshDataAsync` method to return a `Task` with the item collection. However, the `TodoService.GetTodoItemsAsync` method returns void. To satisfy the interface pattern, you must call `GetTodoItemsAsync`, wait for the `GetTodoItemsCompleted` event to fire, and populate the collection. This allows you to return a valid collection to the UI.

The example below creates a new `TaskCompletionSource`, begins the async call in the `RefreshDataAsync` method, and awaits the `Task` provided by the `TaskCompletionSource`. When the `TodoService_GetTodoItemsCompleted` event handler is invoked it populates the `Items` collection and updates the `TaskCompletionSource`:

```

public class SoapService : ISoapService
{
    TaskCompletionSource<bool> getRequestComplete = null;
    ...

    public SoapService()
    {
        ...
        todoService.GetTodoItemsCompleted += TodoService_GetTodoItemsCompleted;
    }

    public async Task<List<TodoItem>> RefreshDataAsync()
    {
        getRequestComplete = new TaskCompletionSource<bool>();
        todoService.GetTodoItemsAsync();
        await getRequestComplete.Task;
        return Items;
    }

    private void TodoService_GetTodoItemsCompleted(object sender, ASMXService.GetTodoItemsCompletedEventArgs e)
    {
        try
        {
            getRequestComplete = getRequestComplete ?? new TaskCompletionSource<bool>();

            Items = new List<TodoItem>();
            foreach (var item in e.Result)
            {
                Items.Add(FromASMXServiceTodoItem(item));
            }
            getRequestComplete?.TrySetResult(true);
        }
        catch (Exception ex)
        {
            Debug.WriteLine(@"\t\tERROR {0}", ex.Message);
        }
    }
}

...
}

```

For more information, see [Asynchronous Programming Model](#) and [TPL and Traditional .NET Framework Asynchronous Programming](#).

Create or edit data

When you create or edit data, you must implement the `ISoapService.SaveTodoItemAsync` method. This method detects whether the `TodoItem` is a new or updated item and calls the appropriate method on the `todoService` object. The `CreateTodoItemCompleted` and `EditTodoItemCompleted` event handlers should also be implemented so you know when the `todoService` has received a response from the ASMX service (these can be combined into a single handler because they perform the same operation). The following example demonstrates the interface and event handler implementations, as well as the `TaskCompletionSource` object used to operate asynchronously:

```

public class SoapService : ISoapService
{
    TaskCompletionSource<bool> saveRequestComplete = null;
    ...

    public SoapService()
    {
        ...
        todoService.CreateTodoItemCompleted += TodoService_SaveTodoItemCompleted;
        todoService.EditTodoItemCompleted += TodoService_SaveTodoItemCompleted;
    }

    public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
    {
        try
        {
            var todoItem = ToASMXServiceTodoItem(item);
            saveRequestComplete = new TaskCompletionSource<bool>();
            if (isNewItem)
            {
                todoService.CreateTodoItemAsync(todoItem);
            }
            else
            {
                todoService.EditTodoItemAsync(todoItem);
            }
            await saveRequestComplete.Task;
        }
        catch (SoapException se)
        {
            Debug.WriteLine("\t\t{0}", se.Message);
        }
        catch (Exception ex)
        {
            Debug.WriteLine("\t\tTERROR {0}", ex.Message);
        }
    }

    private void TodoService_SaveTodoItemCompleted(object sender,
System.ComponentModel.AsyncCompletedEventArgs e)
    {
        saveRequestComplete?.TrySetResult(true);
    }

    ...
}

```

Delete data

Deleting data requires a similar implementation. Define a `TaskCompletionSource`, implement an event handler, and the `ISoapService.DeleteTodoItemAsync` method:

```

public class SoapService : ISoapService
{
    TaskCompletionSource<bool> deleteRequestComplete = null;
    ...

    public SoapService()
    {
        ...
        todoService.DeleteTodoItemCompleted += TodoService_DeleteTodoItemCompleted;
    }

    public async Task DeleteTodoItemAsync (string id)
    {
        try
        {
            deleteRequestComplete = new TaskCompletionSource<bool>();
            todoService.DeleteTodoItemAsync(id);
            await deleteRequestComplete.Task;
        }
        catch (SoapException se)
        {
            Debug.WriteLine("\t\t{0}", se.Message);
        }
        catch (Exception ex)
        {
            Debug.WriteLine("\t\tERROR {0}", ex.Message);
        }
    }

    private void TodoService_DeleteTodoItemCompleted(object sender,
System.ComponentModel.AsyncCompletedEventArgs e)
    {
        deleteRequestComplete?.TrySetResult(true);
    }

    ...
}

```

Test the web service

Testing physical or emulated devices with a locally-hosted service requires custom IIS Configuration, endpoint addresses, and firewall rules to be in place. For more detail on how to set up your environment for testing, see the [Configure remote access to IIS Express](#). The only difference between testing WCF and ASMX is the port number of the TodoService.

Related links

- [TodoASMX \(sample\)](#)
- [IAsyncResult](#)

Consume a Windows Communication Foundation (WCF) Web Service

8/4/2022 • 12 minutes to read • [Edit Online](#)

 [Download the sample](#)

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 including:

- **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 WCF supports the same capabilities that ASMX provides – SOAP messages over HTTP. For more information about consuming an ASMX service, see [Consume ASP.NET Web Services \(ASMX\)](#).

IMPORTANT

The Xamarin platform support for WCF is limited to text-encoded SOAP messages over HTTP/HTTPS using the `BasicHttpBinding` class.

WCF support requires the use of tools only available in a Windows environment to generate the proxy and host the `TodoWCFService`. Building and testing the iOS app will require deploying the `TodoWCFService` on a Windows computer, or as an Azure web service.

Xamarin Forms native apps typically share code with a .NET Standard Class Library. However, .NET Core does not currently support WCF so the shared project must be a legacy Portable Class Library. For information about WCF support in .NET Core, see [Choosing between .NET Core and .NET Framework for server apps](#).

The sample application solution includes a WCF service which can be run locally, and is shown in the following screenshot:



The screenshot shows a Windows Phone displaying a mobile application named "Todo". The app's interface includes a header with the title "Todo" and a subtitle "Learn app development ✓". Below the header, there are three items in a list: "Develop apps", "Publish apps", and a large, empty text input field for entering new todo items.

```

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))
                throw new FaultException("Text is required");

            item.ID = Guid.NewGuid();
            repository.Add(item);
        }

        public void UpdateTodoItem(TodoItem item)
        {
            if (string.IsNullOrEmpty(item.Text))
                throw new FaultException("Text is required");

```

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](#).

Consume 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](#).

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.

Create 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](#).

Create 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.

Retrieve 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.

Create 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.

Update 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.

Delete 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.

Configure remote access to IIS Express

In Visual Studio 2017 or Visual Studio 2019, you should be able to test the UWP application on a PC with no additional configuration. Testing Android and iOS clients may require the additional steps in this section. See [Connect to Local Web Services from iOS Simulators and Android Emulators](#) for more information.

By default, IIS Express will only respond to requests to `localhost`. Remote devices (such as an Android device, an iPhone or even a simulator) will not have access to your local WCF service. You will need to know your Windows 10 workstation IP address on the local network. For the purpose of this example, assume that your workstation has the IP address `192.168.1.143`. The following steps explain how to configure Windows 10 and

IIS Express to accept remote connections and connect to the service from a physical or virtual device:

1. **Add an exception to Windows Firewall.** You must open a port through Windows Firewall that applications on your subnet can use to communicate with the WCF service. Create an inbound rule opening port 49393 in the firewall. From an administrative command prompt, run this command:

```
netsh advfirewall firewall add rule name="TodoWCFService" dir=in protocol=tcp localport=49393 profile=private remoteip=localsubnet action=allow
```

2. **Configure IIS Express to Accept Remote connections.** You can configure IIS Express by editing the configuration file for IIS Express at [solution directory].vs\config\applicationhost.config. Find the `site` element with the name `TodoWCFService`. It should look similar to the following XML:

```
<site name="TodoWCFService" id="2">
    <application path="/" applicationPool="Clr4IntegratedAppPool">
        <virtualDirectory path="/" physicalPath="C:\Users\tom\TodoWCF\TodoWCFService\TodoWCFService" />
    </application>
    <bindings>
        <binding protocol="http" bindingInformation="*:49393:localhost" />
    </bindings>
</site>
```

You will need to add two `binding` elements to open up port 49393 to outside traffic and the Android emulator. The binding uses a `[IP address]:[port]:[hostname]` format that specifies how IIS Express will respond to requests. External requests will have hostnames that must be specified as a `binding`. Add the following XML to the `bindings` element, replacing the IP address with your own IP address:

```
<binding protocol="http" bindingInformation="*:49393:192.168.1.143" />
<binding protocol="http" bindingInformation="*:49393:127.0.0.1" />
```

After your changes the `bindings` element should look like the following:

```
<site name="TodoWCFService" id="2">
    <application path="/" applicationPool="Clr4IntegratedAppPool">
        <virtualDirectory path="/" physicalPath="C:\Users\tom\TodoWCF\TodoWCFService\TodoWCFService" />
    </application>
    <bindings>
        <binding protocol="http" bindingInformation="*:49393:localhost" />
        <binding protocol="http" bindingInformation="*:49393:192.168.1.143" />
        <binding protocol="http" bindingInformation="*:49393:127.0.0.1" />
    </bindings>
</site>
```

IMPORTANT

By default, IIS Express will not accept connections from external sources for security reasons. To enable connections from remote devices you must run IIS Express with Administrative permissions. The easiest way to do this is to run Visual Studio 2017 with Administrative permissions. This will launch IIS Express with Administrative permissions when running the TodoWCFService.

With these steps complete, you should be able to run the TodoWCFService and connect from other devices on your subnet. You can test this by running your application and visiting

<http://localhost:49393/TodoService.svc>. If you get a **Bad Request** error when visiting that URL, your

`bindings` may be incorrect in the IIS Express configuration (the request is reaching IIS Express but is being rejected). If you get a different error it may be that your application is not running or your firewall is incorrectly configured.

To allow IIS Express to keep running and serving the service, turn off the **Edit and Continue** option in **Project Properties > Web > Debuggers**.

3. **Customize the endpoint devices use to access the service.** This step involves configuring the client application, running on a physical or emulated device, to access the WCF service.

The Android emulator utilizes an internal proxy that prevents the emulator from directly accessing the host machine's `localhost` address. Instead, the address `10.0.2.2` on the emulator is routed to `localhost` on the host machine through an internal proxy. These proxied requests will have `127.0.0.1` as the hostname in the request header, which is why you created the IIS Express binding for this hostname in the steps above.

The iOS Simulator runs on a Mac build host, even if you are using the [Remoted iOS Simulator for Windows](#). Network requests from the simulator will have your workstation IP on the local network as the hostname (in this example it's `192.168.1.143`, but your actual IP address will likely be different). This is why you created the IIS Express binding for this hostname in the steps above.

Ensure the `SoapUrl` property in the `Constants.cs` file in the TodoWCF (Portable) project have values that are correct for your network:

```
public static string SoapUrl
{
    get
    {
        var defaultUrl = "http://localhost:49393/TodoService.svc";

        if (Device.RuntimePlatform == Device.Android)
        {
            defaultUrl = "http://10.0.2.2:49393/TodoService.svc";
        }
        else if (Device.RuntimePlatform == Device.iOS)
        {
            defaultUrl = "http://192.168.1.143:49393/TodoService.svc";
        }

        return defaultUrl;
    }
}
```

Once you have configured the `Constants.cs` with the appropriate endpoints, you should be able to connect to the TodoWCFService running on your Windows 10 workstation from physical or virtual devices.

Related links

- [TodoWCF \(sample\)](#)
- [How to: Create a Windows Communication Foundation Client](#)
- [ServiceModel Metadata Utility Tool \(svcutil.exe\)](#)

Consume a RESTful web service

8/4/2022 • 7 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

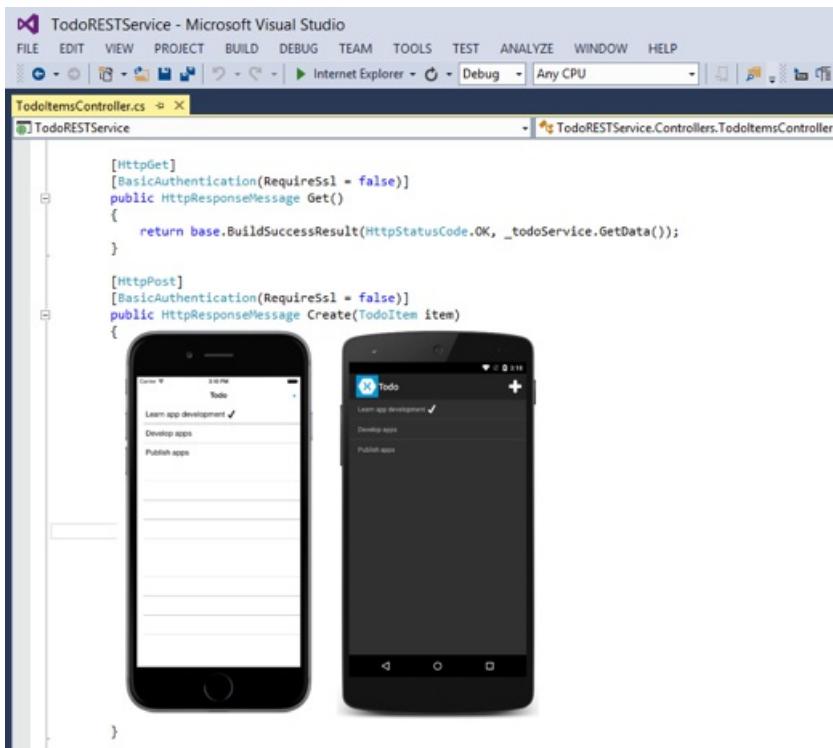
Web service APIs that adhere to REST are called RESTful APIs, and are defined using:

- 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.

When the sample application is run, it will connect to a locally hosted REST service, 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](#).

Consume 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 URLs 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 routing middleware to match the URLs of incoming requests and map them to actions. REST APIs should use attribute routing to map the app's functionality as a set of resources whose operations are represented by HTTP verbs. Attribute routing uses a set

of attributes to map actions directly to route templates. For more information about attribute routing, see [Attribute routing for REST APIs](#). For more information about building the REST service using ASP.NET Core, see [Creating Backend Services for Native Mobile Applications](#).

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.

Create 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 ();
        ...
    }
    ...
}
```

Retrieve 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 ()
{
    ...
    Uri uri = new Uri (string.Format (Constants.TodoItemsUrl, string.Empty));
    ...
    HttpResponseMessage response = await client.GetAsync (uri);
    if (response.IsSuccessStatusCode)
    {
        string content = await response.Content.ReadAsStringAsync ();
        Items = JsonSerializer.Deserialize<List<TodoItem>>(content, serializerOptions);
    }
    ...
}
```

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

deserialized from JSON to a `List` of `TodoItem` instances.

WARNING

Using the `ReadAsStringAsync` method to retrieve a large response can have a negative performance impact. In such circumstances the response should be directly deserialized to avoid having to fully buffer it.

Create 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)
{
    Uri uri = new Uri (string.Format (Constants.TodoItemsUrl, string.Empty));

    ...
    string json = JsonSerializer.Serialize<TodoItem>(item, serializerOptions);
    StringContent content = new StringContent (json, Encoding.UTF8, "application/json");

    HttpResponseMessage response = null;
    if (isNewItem)
    {
        response = await client.PostAsync (uri, content);
    }
    ...

    if (response.IsSuccessStatusCode)
    {
        Debug.WriteLine (@"\tTodoItem successfully saved.");
    }
    ...
}
```

The `TodoItem` instance is serialized 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.

Update 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.

Delete 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)
{
    Uri uri = new Uri (string.Format (Constants.TodoItemsUrl, id));
    ...
    HttpResponseMessage response = await client.DeleteAsync (uri);
    if (response.IsSuccessStatusCode)
    {
        Debug.WriteLine (@"\tTodoItem 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.

Local development

If you are developing your REST web service locally with a framework such as ASP.NET Core Web API, you can debug your web service and mobile app at the same time. In this scenario you must enable clear-text HTTP traffic for the iOS simulator and Android emulator. For information about configuration your project to allow communication, see [Connect to local web services](#).

Related links

- [Microsoft Learn: Create a web API with ASP.NET Core](#)
- [Creating Backend Services for Native Mobile Applications](#)
- [Attribute routing for REST APIs](#)
- [TodoREST \(sample\)](#)
- [HttpClient API](#)
- [Android Network Security Configuration](#)
- [iOS App Transport Security](#)
- [Connect to local web services](#)

Xamarin.Forms Web Service Authentication

8/4/2022 • 2 minutes to read • [Edit Online](#)

Authenticate 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 explains how to use basic authentication to protect access to RESTful web service resources.

Authenticate 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 explains how to use Microsoft Authentication Library (MSAL) and Azure Active Directory B2C to integrate consumer identity management into a Xamarin.Forms application.

Authenticate Users with an Azure Cosmos DB Document Database and Xamarin.Forms

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.

Authenticate a RESTful Web Service

8/4/2022 • 3 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.

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: Basic` 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 WGFTYXJpb1VzZXI6WGFTYXJpb1Bhc3N3b3Jk
```

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.

IMPORTANT

While this code stores credentials as constants, they should not be stored in an insecure format in a published application.

Processing the Authorization Header Server Side

The REST service should decorate each action with the `[BasicAuthentication]` attribute. This attribute 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 a 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.

Related Links

- [Consume a RESTful web service](#)
- [HttpClient](#)

Authenticate Users with Azure Active Directory B2C

8/4/2022 • 9 minutes to read • [Edit Online](#)



[Download the sample](#)

Azure Active Directory B2C provides cloud identity management for consumer-facing web and mobile applications. This article shows how to use Azure Active Directory B2C to integrate identity management into a mobile application with the Microsoft Authentication Library.

Overview

Azure Active Directory B2C (ADB2C) is an identity management service for consumer-facing applications. It allows users to sign in to your application using their existing social accounts or custom credentials such as email or username, and password. Custom credential accounts 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.
2. Register your mobile application with the Azure Active Directory B2C tenant.
3. Create policies for sign-up and sign-in, and forgot password user flows.
4. Use the Microsoft Authentication Library (MSAL) to start an authentication workflow with your Azure Active Directory B2C tenant.

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

Azure Active Directory B2C supports multiple identity providers including Microsoft, GitHub, Facebook, Twitter and more. For more information on Azure Active Directory B2C capabilities, see [Azure Active Directory B2C Documentation](#).

Microsoft Authentication Library supports multiple application architectures and platforms. For information about MSAL capabilities, see [Microsoft Authentication Library](#) on GitHub.

Configure an Azure Active Directory B2C tenant

To run the sample project, you must create an Azure Active Directory B2C tenant. For more information, see [Create an Azure Active Directory B2C tenant in the Azure portal](#).

Once you create a tenant, you will need the **tenant name** and **tenant ID** to configure the mobile application. The tenant ID and name are defined by the domain generated when you created your tenant URL. If your generated tenant URL is `https://contoso20190410tenant.onmicrosoft.com/` the **tenant ID** is

`contoso20190410tenant.onmicrosoft.com` and the **tenant name** is `contoso20190410tenant`. Find the tenant domain in the Azure portal by clicking the **directory and subscription filter** in the top menu. The following screenshot shows the Azure directory and subscription filter button and the tenant domain:

The screenshot shows the Azure portal's directory list. At the top, there are tabs for 'Favorites' and 'All Directories'. Below is a search bar. The main area lists tenants, with 'contoso20190410tenant.onmicrosoft.com' highlighted by a red box.

In the sample project, edit the `Constants.cs` file to set the `tenantName` and `tenantId` fields. The following code shows how these values should be set if your tenant domain is <https://contoso20190410tenant.onmicrosoft.com/>, replace these values with values from your portal:

```
public static class Constants
{
    static readonly string tenantName = "contoso20190410tenant";
    static readonly string tenantId = "contoso20190410tenant.onmicrosoft.com";
    ...
}
```

Register your mobile application with Azure Active Directory B2C

A mobile application must be registered with the tenant before it can connect and authenticate users. The registration process assigns a unique **Application ID** to the application, and a **Redirect URL** that directs responses back to the application after authentication. For more information, see [Azure Active Directory B2C: Register your application](#). You will need to know the **Application ID** assigned to your application, which is listed after the application name in the properties view. The following screenshot shows where to find the Application ID:

The screenshot shows the properties view of a mobile application named 'adb2cAuthApp'. It includes fields for 'Name' (adb2cAuthApp) and 'Application ID' (which is blurred). The 'Application ID' field is highlighted by a red box. Below the application details, it says 'Web App / Web API'.

Microsoft Authentication Library expects the **Redirect URL** for your application to be your **Application ID** prefixed with the text "msal", and followed by an endpoint called "auth". If your Application ID is "1234abcd", the full URL should be `msal1234abcd://auth`. Make sure that your application has enabled the **Native client** setting and create a **Custom Redirect URI** using your Application ID as shown in the following screenshot:

The screenshot shows the 'Properties' page for an application named 'adb2cAuthApp' in the Azure AD B2C - Applications section. The 'Native client' section is highlighted with a red box. It contains a toggle switch labeled 'Include native client' with options 'Yes' and 'No' (the latter is selected). Below this, there are two redirect URI fields: 'urn:ietf:wg:oauth:2.0:oob' and 'https://login.microsoftonline.com/tfp/oauth2/nativeclient'. A note below the first field states 'Redirect URLs must not be http or https'. At the bottom of the Native client section, another red box highlights the 'Custom Redirect URI' field, which contains 'msal [REDACTED]://auth'.

The URL will be used later in both the `Android ApplicationManifest.xml` and the `iOS Info.plist`.

In the sample project, edit the `Constants.cs` file to set the `clientId` field to your **Application ID**. The following code shows how this value should be set if your Application ID is `1234abcd`:

```
public static class Constants
{
    static readonly string tenantName = "contoso20190410tenant";
    static readonly string tenantId = "contoso20190410tenant.onmicrosoft.com";
    static readonly string clientId = "1234abcd";
    ...
}
```

Create sign-up and sign-in policies, and forgot password policies

A policy is an experience users go through to complete a task such as creating an account or resetting a password. A policy also specifies the contents of tokens the application receives when the user returns from the experience. You must set up policies for both account sign-up and sign-in, and reset password. Azure has built-in policies that simplify creation of common policies. For more information, see [Azure Active Directory B2C: Built-in policies](#).

When you've completed policy setup, you should have two policies in the **User flows (policies)** view in the Azure portal. The following screenshot demonstrates two configured policies in the Azure portal:

In the sample project, edit the `Constants.cs` file to set the `policySignin` and `policyPassword` fields to reflect the names you chose during policy setup:

```
public static class Constants
{
    static readonly string tenantName = "contoso20190410tenant";
    static readonly string tenantId = "contoso20190410tenant.onmicrosoft.com";
    static readonly string clientId = "1234abcd";
    static readonly string policySignin = "B2C_1_signupsignin1";
    static readonly string policyPassword = "B2C_1_passwordreset";
    ...
}
```

Use the Microsoft Authentication Library (MSAL) for authentication

The Microsoft Authentication Library (MSAL) NuGet package must be added to the shared, .NET Standard project, and the platform projects in a Xamarin.Forms solution. MSAL includes a `PublicClientApplicationBuilder` class that constructs an object adhering to the `IPublicClientApplication` interface. MSAL utilizes `With` clauses to supply additional parameters to the constructor and authentication methods.

In the sample project, the code behind for `App.xaml` defines static properties named `AuthenticationClient` and `UIParent`, and instantiates the `AuthenticationClient` object in the constructor. The `WithIosKeychainSecurityGroup` clause provides a security group name for iOS applications. The `WithB2CAuthority` clause provides the default `Authority`, or policy, that will be used to authenticate users. The `WithRedirectUri` clause tells the Azure Notification Hubs instance which Redirect URI to use if multiple URIs are specified. The following example demonstrates how to instantiate the `PublicClientApplication`:

```

public partial class App : Application
{
    public static IPublicClientApplication AuthenticationClient { get; private set; }

    public static object UIParent { get; set; } = null;

    public App()
    {
        InitializeComponent();

        AuthenticationClient = PublicClientApplicationBuilder.Create(Constants.ClientId)
            .WithIosKeychainSecurityGroup(Constants.IosKeychainSecurityGroups)
            .WithB2CAuthority(Constants.AuthoritySignin)
            .WithRedirectUri($"msal{Constants.ClientId}://auth")
            .Build();

        MainPage = new NavigationPage(new LoginPage());
    }

    ...

```

NOTE

If your Azure Notification Hubs instance only has one Redirect URI defined, the `AuthenticationClient` instance may work without specifying the Redirect URI with the `WithRedirectUri` clause. However, you should always specify this value in case your Azure configuration expands to support other clients or authentication methods.

The `OnAppearing` event handler in the `LoginPage.xaml.cs` code behind calls `AcquireTokenSilentAsync` to refresh the authentication token for users that have logged in before. The authentication process redirects to the `LogoutPage` if successful and does nothing on failure. The following example shows the silent reauthentication process in `OnAppearing`:

```

public partial class LoginPage : ContentPage
{
    ...

    protected override async void OnAppearing()
    {
        try
        {
            // Look for existing account
            IEnumerable<IAccount> accounts = await App.AuthenticationClient.GetAccountsAsync();

            AuthenticationResult result = await App.AuthenticationClient
                .AcquireTokenSilent(Constants.Scopes, accounts.FirstOrDefault())
                .ExecuteAsync();

            await Navigation.PushAsync(new LogoutPage(result));
        }
        catch
        {
            // Do nothing - the user isn't logged in
        }
        base.OnAppearing();
    }

    ...
}

```

The `OnLoginButtonClicked` event handler (fired when the Login button is clicked) calls `AcquireTokenAsync`. The MSAL library automatically opens the mobile device browser and navigates to the login page. The sign-in URL,

called an **Authority**, is a combination of the tenant name and policies defined in the `Constants.cs` file. If the user chooses the forgot password option, they are returned to the app with an exception, which launches the forgot password experience. The following example shows the authentication process:

```
public partial class LoginPage : ContentPage
{
    ...

    async void OnLoginButtonClicked(object sender, EventArgs e)
    {
        AuthenticationResult result;
        try
        {
            result = await App.AuthenticationClient
                .AcquireTokenInteractive(Constants.Scopes)
                .WithPrompt(Prompt.SelectAccount)
                .WithParentActivityOrWindow(App.UIParent)
                .ExecuteAsync();

            await Navigation.PushAsync(new LogoutPage(result));
        }
        catch (MsalException ex)
        {
            if (ex.Message != null && ex.Message.Contains("AADB2C90118"))
            {
                result = await OnForgotPassword();
                await Navigation.PushAsync(new LogoutPage(result));
            }
            else if (ex.ErrorCode != "authentication_canceled")
            {
                await DisplayAlert("An error has occurred", "Exception message: " + ex.Message, "Dismiss");
            }
        }
    }

    ...
}
```

The `OnForgotPassword` method is similar to the sign-in process but implements a custom policy.

`OnForgotPassword` uses a different overload of `AcquireTokenAsync`, which allows you to provide a specific **Authority**. The following example shows how to supply a custom **Authority** when acquiring a token:

```
public partial class LoginPage : ContentPage
{
    ...

    async Task<AuthenticationResult> OnForgotPassword()
    {
        try
        {
            return await App.AuthenticationClient
                .AcquireTokenInteractive(Constants.Scopes)
                .WithPrompt(Prompt.SelectAccount)
                .WithParentActivityOrWindow(App.UIParent)
                .WithB2CAuthority(Constants.AuthorityPasswordReset)
                .ExecuteAsync();
        }
        catch (MsalException)
        {
            // Do nothing - ErrorCode will be displayed in OnLoginButtonClicked
            return null;
        }
    }
}
```

The final piece of authentication is the sign out process. The `OnLogoutButtonClicked` method is called when the user presses the sign out button. It loops through all accounts and ensures their tokens have been invalidated. The sample below demonstrates the sign out implementation:

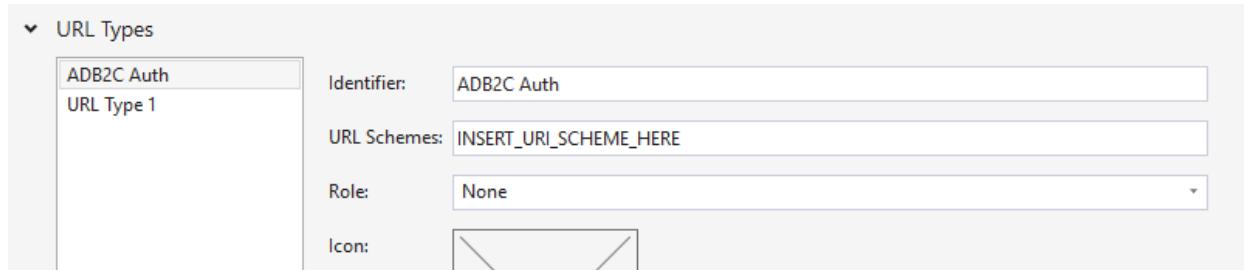
```
public partial class LogoutPage : ContentPage
{
    ...
    async void OnLogoutButtonClicked(object sender, EventArgs e)
    {
        IEnumerable<IAccount> accounts = await App.AuthenticationClient.GetAccountsAsync();

        while (accounts.Any())
        {
            await App.AuthenticationClient.RemoveAsync(accounts.First());
            accounts = await App.AuthenticationClient.GetAccountsAsync();
        }

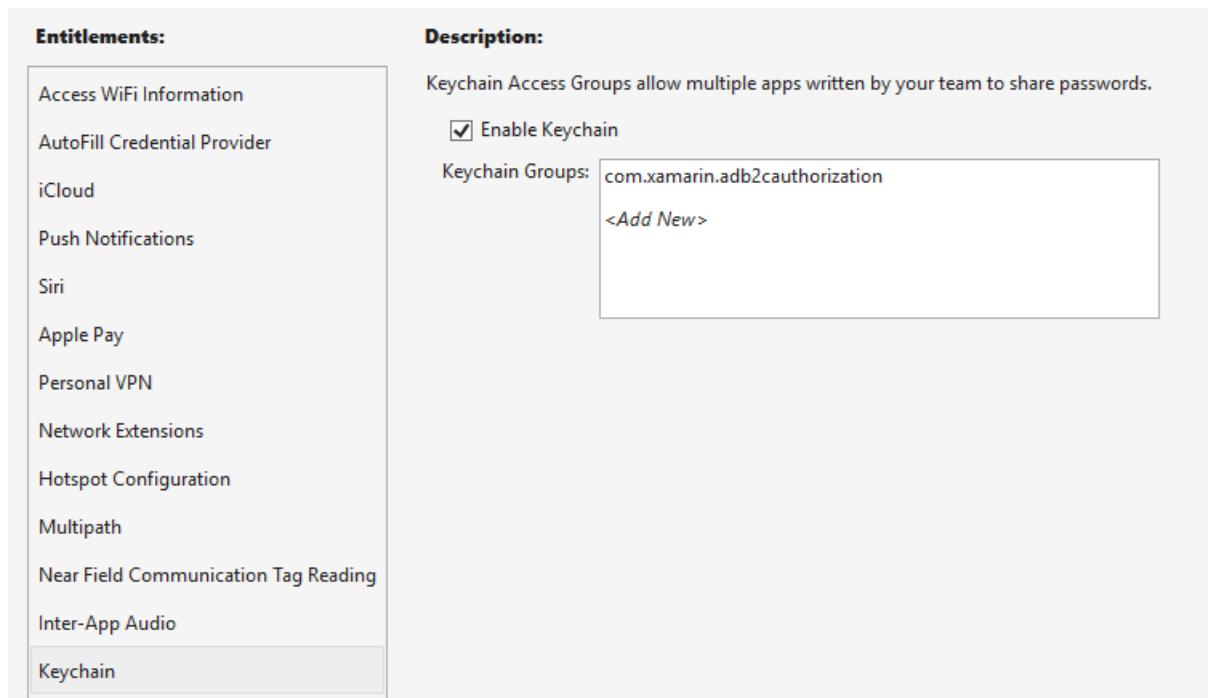
        await Navigation.PopAsync();
    }
}
```

iOS

On iOS, the custom URL scheme that was registered with Azure Active Directory B2C must be registered in `Info.plist`. MSAL expects the URL scheme to adhere to a specific pattern, described previously in [Register your mobile application with Azure Active Directory B2C](#). The following screenshot shows the custom URL scheme in `Info.plist`.



MSAL also requires Keychain Entitlements on iOS, registered in the `Entitlements.plist`, as shown in the following screenshot:



When Azure Active Directory B2C completes the authorization request, it redirects to the registered redirect URL. The custom URL scheme results in iOS launching the mobile application and passing in the URL as a launch parameter, where it's processed by the `OpenUrl` override of the application's `AppDelegate` class, and returns control of the experience to MSAL. The `OpenUrl` implementation 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 base.OpenUrl(app, url, options);
        }
    }
}
```

Android

On Android, the custom URL scheme that was registered with Azure Active Directory B2C must be registered in the `AndroidManifest.xml`. MSAL expects the URL scheme to adhere to a specific pattern, described previously in [Register your mobile application with Azure Active Directory B2C](#). The following example shows the custom URL scheme in the `AndroidManifest.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1"
    android:versionName="1.0" package="com.xamarin.adb2cauthorization">
    <uses-sdk android:minSdkVersion="15" />
    <application android:label="ADB2CAuthorization">
        <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" />
                <!-- example -->
                <!-- <data android:scheme="msalaaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeee" android:host="auth" /> -->
                <data android:scheme="INSERT_URI_SCHEME_HERE" android:host="auth" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The `MainActivity` class must be modified to provide the `UIParent` object to the application during the `OnCreate` call. When Azure Active Directory B2C completes the authorization request, it redirects to the registered URL scheme from the `AndroidManifest.xml`. The registered URI scheme results in Android calling the `OnActivityResult` method with the URL as a launch parameter, where it's processed by the `SetAuthenticationContinuationEventArgs` method.

```
public class MainActivity : FormsAppCompatActivity
{
    protected override void OnCreate(Bundle bundle)
    {
        TabLayoutResource = Resource.Layout.Tabbar;
        ToolbarResource = Resource.Layout.Toolbar;

        base.OnCreate(bundle);

        Forms.Init(this, bundle);
        LoadApplication(new App());
        App.UIParent = this;
    }

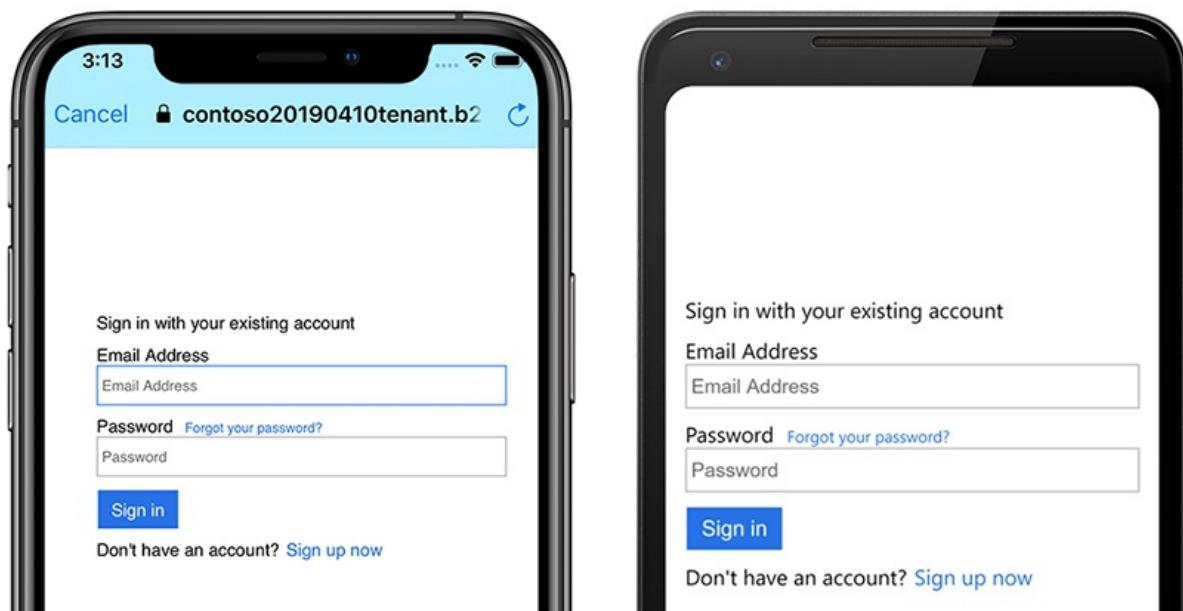
    protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
    {
        base.OnActivityResult(requestCode, resultCode, data);
        AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(requestCode, resultCode,
data);
    }
}
```

Universal Windows Platform

No additional setup is required to use MSAL on the Universal Windows Platform

Run the project

Run the application on a virtual or physical device. Tapping the **Login** button should open the browser and navigate to a page where you can sign in or create an account. After completing the sign in process, you should be returned to the application's logout page. The following screenshot shows the user sign in screen running on Android and iOS:



Related Links

- [AzureADB2CAuth \(sample\)](#)
- [Azure Active Directory B2C](#)
- [Microsoft Authentication Library](#)
- [Microsoft Authentication Library Documentation](#)

Authenticate Users with an Azure Cosmos DB Document Database and Xamarin.Forms

8/4/2022 • 10 minutes to read • [Edit Online](#)

 [Download the sample](#)

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

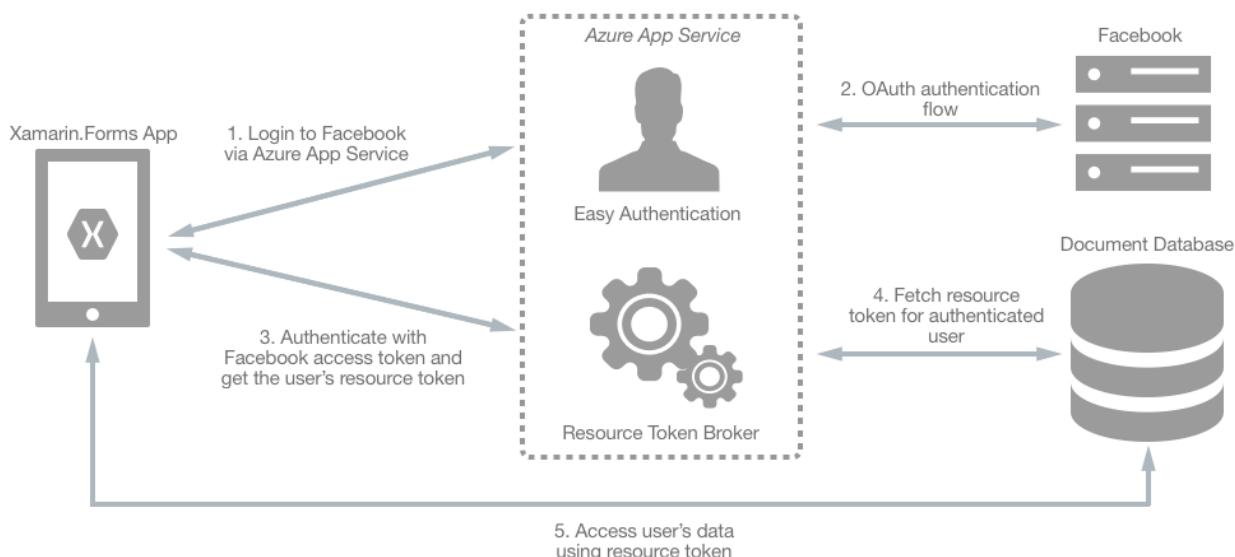
A partition key must be specified when creating a partitioned collection, and documents with the same partition key will be stored in the same partition. Therefore, specifying the user's identity as a partition key will result in a partitioned collection that will only store documents for that user. 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 [Azure 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](#).

NOTE

If you don't have an [Azure subscription](#), create a [free account](#) before you begin.

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	https://cosmosdb-test.documents.azure.com/
accountKey	[REDACTED]
databaseId	TodoList
collectionId	UserItems
hostURL	https://cosmosdb-test.azurewebsites.net/

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

<input checked="" type="checkbox"/> Yes	<input type="checkbox"/>	Client OAuth Login 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. [?]
<input checked="" type="checkbox"/> Yes	<input type="checkbox"/>	Web OAuth Login Enables web based OAuth client login for building custom login flows. [?]
<input type="checkbox"/>	<input checked="" type="checkbox"/> No	Force Web OAuth Reauthentication When on, prompts people to enter their Facebook password in order to log in on the web. [?]
<input type="checkbox"/>	<input checked="" type="checkbox"/> No	Embedded Browser OAuth Login Enables browser control redirect uri for OAuth client login. [?]

Valid OAuth redirect URIs

http://www.contoso.com/.auth/login/facebook/callback ×

<input type="checkbox"/>	<input checked="" type="checkbox"/> No	Login from Devices 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:

The screenshot shows the 'Authentication / Authorization' blade in the Azure portal. It includes sections for 'App Service Authentication' (set to 'On'), 'Action to take when request is not authenticated' (set to 'Log in with Facebook'), 'Authentication Providers' (listing Facebook, Google, Twitter, and Microsoft Account, all configured), and 'Advanced Settings' (Token Store set to '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:

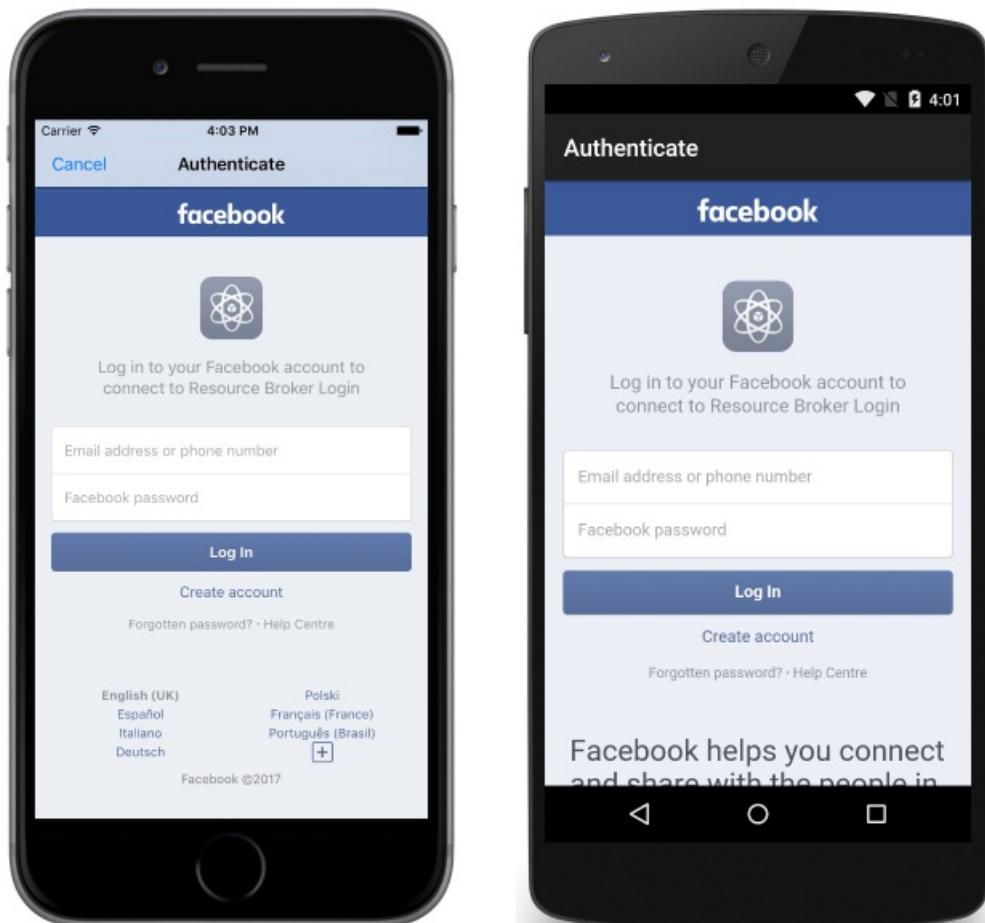
1. Open the Xamarin.Forms solution.
2. Open `Constants.cs` and update the values of the following constants:
 - `EndpointUri` – the value should be the Cosmos DB account URL from the Keys blade of the Cosmos DB account.
 - `DatabaseName` – the value should be the name of the document database.
 - `CollectionName` – 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 redirecting 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.

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==;a74aukk99qtwk8v5rxfrfz7ay7zzqfkbfkremrwtapavvw2mrvia4umbi/7iikrqq+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](#)

Improve Xamarin.Forms App Performance

8/4/2022 • 15 minutes to read • [Edit Online](#)

Evolve 2016: Optimizing App Performance with Xamarin.Forms

Poor application performance presents itself in many ways. It can make an application seem unresponsive, can cause slow scrolling, and can reduce device 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 many techniques for increasing the performance, and perceived 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.

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 enabled by default in new Xamarin.Forms solutions. However, it may need to be enabled in older solutions. For more information, see [Compiling XAML](#).

Use compiled bindings

Compiled bindings improve data binding performance in Xamarin.Forms applications by resolving binding expressions at compile time, rather than at runtime with reflection. Compiling a binding expression generates compiled code that typically resolves a binding 8-20 times quicker than using a classic binding. For more information, see [Compiled Bindings](#).

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".

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.

From Xamarin.Forms 4.0 onwards, all applications targeting `FormsAppCompatActivity` use fast renderers by default. For more information, see [Fast Renderers](#).

Enable startup tracing on Android

Ahead of Time (AOT) compilation on Android minimizes Just in Time (JIT) application startup overhead and memory usage, at the cost of creating a much larger APK. An alternative is to use startup tracing, which provides a trade-off between Android APK size and startup time, when compared to conventional AOT compilation.

Instead of compiling as much of the application as possible to unmanaged code, startup tracing compiles only the set of managed methods that represent the most expensive parts of application startup in a blank Xamarin.Forms application. This approach results in a reduced APK size, when compared to conventional AOT compilation, while still providing similar startup improvements.

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](#).

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">
    <StackLayout>
        <Image Source="waterfront.jpg" />
    </StackLayout>
</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">
    <Image Source="waterfront.jpg" />
</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">
    <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>

```

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">
    <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>

```

Optimize layout performance

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.
- Avoid calling 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.

Use asynchronous programming

The overall responsiveness of your application can be enhanced, and performance bottlenecks often avoided, by using asynchronous programming. In .NET, the [Task-based Asynchronous Pattern \(TAP\)](#) is the recommended design pattern for asynchronous operations. However, incorrect use of the TAP can result in unperformant applications. Therefore, the following guidelines should be followed when using the TAP.

Fundamentals

- Understand the task lifecycle, which is represented by the `TaskStatus` enumeration. For more information, see [The meaning of TaskStatus](#) and [Task status](#).
- Use the `Task.WhenAll` method to asynchronously wait for multiple asynchronous operations to finish, rather than individually `await` a series of asynchronous operations. For more information, see [Task.WhenAll](#).
- Use the `Task.WhenAny` method to asynchronously wait for one of multiple asynchronous operations to finish. For more information, see [Task.WhenAny](#).
- Use the `Task.Delay` method to produce a `Task` object that finishes after the specified time. This is useful for scenarios such as polling for data, and delaying handling user input for a predetermined time. For more information, see [Task.Delay](#).
- Execute intensive synchronous CPU operations on the thread pool with the `Task.Run` method. This method is a shortcut for the `TaskFactory.StartNew` method, with the most optimal arguments set. For more information, see [Task.Run](#).
- Avoid trying to create asynchronous constructors. Instead, use lifecycle events or separate initialization logic to correctly `await` any initialization. For more information, see [Async Constructors](#) on blog.stephencleary.com.
- Use the lazy task pattern to avoid waiting for asynchronous operations to complete during application startup. For more information, see [AsyncLazy](#).
- Create a task wrapper for existing asynchronous operations, that don't use the TAP, by creating

`TaskCompletionSource<T>` objects. These objects gain the benefits of `Task` programmability, and enable you to control the lifetime and completion of the associated `Task`. For more information, see [The Nature of TaskCompletionSource](#).

- Return a `Task` object, instead of returning an awaited `Task` object, when there's no need to process the result of an asynchronous operation. This is more performant due to less context switching being performed.
- Use the Task Parallel Library (TPL) Dataflow library in scenarios such as processing data as it becomes available, or when you have multiple operations that must communicate with each other asynchronously. For more information, see [Dataflow \(Task Parallel Library\)](#).

UI

- Call an asynchronous version of an API, if it's available. This will keep the UI thread unblocked, which will help to improve the user's experience with the application.
- Update UI elements with data from asynchronous operations on the UI thread, to avoid exceptions being thrown. However, updates to the `ListView.ItemsSource` property will automatically be marshaled to the UI thread. For information about determining if code is running on the UI thread, see [Xamarin.Essentials: MainThread](#).

IMPORTANT

Any control properties that are updated via data binding will be automatically marshaled to the UI thread.

Error handling

- Learn about asynchronous exception handling. Unhandled exceptions that are thrown by code that's running asynchronously are propagated back to the calling thread, except in certain scenarios. For more information, see [Exception handling \(Task Parallel Library\)](#).
- Avoid creating `async void` methods, and instead create `async Task` methods. These enable easier error-handling, composability, and testability. The exception to this guideline is asynchronous event handlers, which must return `void`. For more information, see [Avoid Async Void](#).
- Don't mix blocking and asynchronous code by calling the `Task.Wait`, `Task.Result`, or `GetAwaiter().GetResult` methods, as they can result in deadlock occurring. However, if this guideline must be violated, the preferred approach is to call the `GetAwaiter().GetResult` method because it preserves the task exceptions. For more information, see [Async All the Way](#) and [Task Exception Handling in .NET 4.5](#).
- Use the `ConfigureAwait` method whenever possible, to create context-free code. Context-free code has better performance for mobile applications and is a useful technique for avoiding deadlock when working with a partially asynchronous codebase. For more information, see [Configure Context](#).
- Use *continuation tasks* for functionality such as handling exceptions thrown by the previous asynchronous operation, and canceling a continuation either before it starts or while it is running. For more information, see [Chaining Tasks by Using Continuous Tasks](#).
- Use an asynchronous `ICommand` implementation when asynchronous operations are invoked from the `ICommand`. This ensures that any exceptions in the asynchronous command logic can be handled. For more information, see [Async Programming: Patterns for Asynchronous MVVM Applications: Commands](#).

Choose a dependency injection container carefully

Dependency injection containers introduce additional performance constraints into mobile applications. Registering and resolving types with a 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 app. If there are many or deep dependencies, the cost of creation can increase significantly. In addition, type

registration, which usually occurs during application startup, can have a noticeable impact on startup time, dependent upon the container being used.

As an alternative, dependency injection can be made more performant by implementing it manually using factories.

Create Shell applications

Xamarin.Forms Shell applications provide an opinionated navigation experience based on flyouts and tabs. If your application user experience can be implemented with Shell, it is beneficial to do so. Shell applications help to avoid a poor startup experience, because pages are created on demand in response to navigation rather than at application startup, which occurs with applications that use a ['TabbedPage'](#). For more information, see [Xamarin.Forms Shell](#).

Use CollectionView instead of ListView

[CollectionView](#) is a view for presenting lists of data using different layout specifications. It provides a more flexible, and performant alternative to [ListView](#). For more information, see [Xamarin.Forms CollectionView](#).

Optimize ListView performance

When using [ListView](#), 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 an application'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 containing multiple [Label](#) objects:

```

<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>

```

The same page layout can be maintained with a reduced element count, as shown in the following code example:

```

<StackLayout Padding="20,35,20,20" Spacing="25">
    <Label Text="Hello" />
    <Label Text="Welcome to the App!" />
    <Label Text="Downloading Data..." />
</StackLayout>

```

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 application'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>

```

For more information about application resources, see [XAML Styles](#).

Use the custom renderer pattern

Most Xamarin.Forms 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 project, 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 (e.OldElement != null)
    {
        // Unsubscribe from event handlers and cleanup any resources
    }

    if (e.NewElement != null)
    {
        if (Control == null)
        {
            // Instantiate the native control with the SetNativeControl method
        }
        // Configure the control and subscribe to event handlers
    }
}

```

A new native control should only be instantiated once, when the `Control` property is `null`. In addition, the control should only be created, 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 an efficiently performing custom renderer that doesn't suffer from memory leaks.

IMPORTANT

The `SetNativeControl` method should only be invoked if the `e.NewElement` property is not `null`, and the `Control` property is `null`.

For more information about custom renderers, see [Customizing Controls on Each Platform](#).

Related links

- [Cross-Platform Performance](#)
- [Compiling XAML](#)
- [Compiled Bindings](#)
- [Fast Renderers](#)
- [Layout Compression](#)
- [Xamarin.Forms Shell](#)
- [Xamarin.Forms CollectionView](#)
- [ListView Performance](#)
- [Optimize Image Resources](#)
- [XAML Styles](#)
- [Customizing Controls on Each Platform](#)

Xamarin Hot Restart

8/4/2022 • 3 minutes to read • [Edit Online](#)

Xamarin Hot Restart enables you to quickly test changes to your app during development, including multi-file code edits, resources, and references. It pushes the new changes to the existing app bundle on the debug target which results in a much faster build and deploy cycle.

IMPORTANT

Xamarin Hot Restart is currently available in Visual Studio 2019 version 16.5 stable and supports iOS apps using Xamarin.Forms. Support for Visual Studio for Mac and non-Xamarin.Forms apps is on the roadmap.

Requirements

- Visual Studio 2019 version 16.5 or higher
- iTunes (Microsoft Store or 64-bit versions)
- Apple Developer account and paid [Apple Developer Program](#) enrollment

Initial setup

NOTE

Xamarin Hot Restart is disabled by default on Visual Studio 16.8 and previous versions. You can enable it under **Tools > Options > Environment > Preview Features > Enable Xamarin Hot Restart**. Starting in Visual Studio 16.9, Xamarin Hot Restart is on by default and can be turned off from **Tools > Options > Xamarin > iOS Settings > Enable Hot Restart**.

1. Ensure the iOS project is set as the startup project and the build configuration is set to **Debug|iPhone**.
 - a. If this is an existing project, go to **Build > Configuration Manager...** and ensure **Deploy** is enabled for the iOS project.
2. Select and click **Local Device** in the toolbar to launch the setup wizard:



3. If iTunes is not installed, click **Download iTunes** to download the installer. Click **Next** when the iTunes installation is complete.
4. Connect an iOS device to your machine. If a device was already plugged in, unplug then reconnect it. The device name will appear in the wizard once it is detected. Click **Next**.
5. Enter your Apple Developer account credentials and click **Next**.
6. Select a development team using the dropdown menu in order to enable **automatic provisioning** in the project. Click **Finish**.

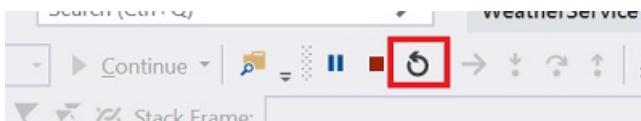
NOTE

Using automatic provisioning is recommended so additional iOS devices can be easily configured for deployment. However, you can disable it and continue using manual provisioning if the correct provisioning profiles are present.

Use Xamarin Hot Restart

After the initial setup, your connected device will appear in the debug target dropdown menu. To debug your app, select your device in the dropdown and click the **Run** button. You may see a message in Visual Studio asking you to manually launch the app on the device in order to start the debug session.

You can make edits to your code files while debugging, then press the **Restart** button in the debug toolbar or use **Ctrl+Shift+F5** to restart the debug session with your new changes applied:



You can also use the `HOTRESTART` preprocessor symbol to prevent certain code from executing when debugging with Xamarin Hot Restart.

Limitations

- Only iOS apps built with Xamarin.Forms and iOS devices are currently supported.
- Only 64-bit iOS devices are supported. As of iOS 11, Apple no longer allows running iOS apps on the 32-bit architecture (devices earlier than iPhone 5s).
- Storyboard and XIB files are not supported and the app may crash if it attempts to load these at runtime. Use the `HOTRESTART` preprocessor symbol to prevent this code from executing.
- Static iOS libraries and frameworks are not supported and you may see runtime errors or crashes if your app attempts to load these. Use the `HOTRESTART` preprocessor symbol to prevent this code from executing. Dynamic iOS libraries are supported.
- You cannot use Xamarin Hot Restart to create app bundles for publishing. You will still need a Mac machine to do a full compilation, signing, and deployment for your application to production.
- Asset Catalogs are currently not supported. When using Hot Restart, your app will show the default icon and launch screen for Xamarin apps. When paired to a Mac, or developing on a Mac, your Asset Catalogs will work.

Troubleshoot

- There is a known issue where having device-specific builds enabled prevents the app from entering debug mode. Workaround is to disable this under **Properties > iOS Build** and retry debugging. This will be fixed in a future release.
- If the app is already present on the device, trying to deploy with Hot Restart may fail with a `AMDeviceStartHouseArrestService` error. The workaround is to uninstall the app on the device then deploy again.
- Entering an Apple ID that is not part of the Apple Developer Program might result in the following error:
`Authentication Error. Xcode 7.3 or later is required to continue developing with your Apple ID.` You must have a valid Apple Developer account to use Xamarin Hot Restart on iOS devices.

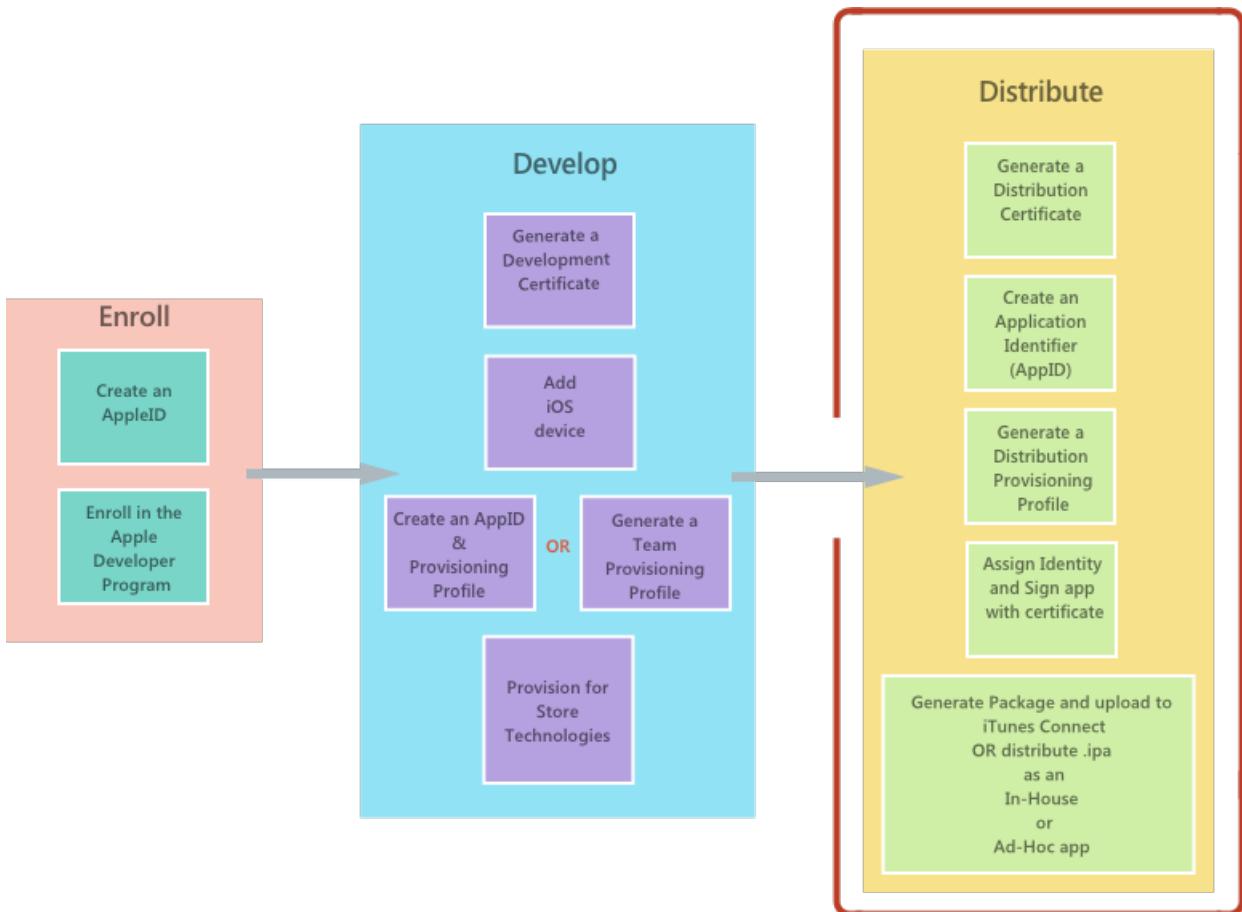
To report additional issues, please use the feedback tool at [Help > Send Feedback > Report a Problem](#).

Xamarin.iOS app distribution overview

8/4/2022 • 2 minutes to read • [Edit Online](#)

This document gives an overview of the distribution techniques that are available for Xamarin.iOS applications and serves as a pointer to more detailed documents on the topic.

Once an Xamarin.iOS app has been developed, the next step in the software development lifecycle is to distribute the app to users, as shown in the highlighted section of the diagram below:



Apple provides the following ways to distribute an iOS application:

- [App Store](#)
- [In-house \(enterprise\)](#)
- [Ad hoc](#)
- [Custom apps for business](#)

All these scenarios require that applications be provisioned using the appropriate *provisioning profile*. Provisioning profiles are files that contain code signing information, as well as the identity of the application and the intended distribution mechanism. For the non-App Store distribution they also contain information about what devices the app can be deployed to.

App Store distribution

IMPORTANT

Apple has indicated that starting in March 2019, all apps and updates submitted to the App Store must have been built with the iOS 12.1 SDK or later, included in Xcode 10.1 or later. Apps should also support the iPhone XS and 12.9" iPad Pro screen sizes.

This is the main way that iOS applications are distributed to consumers on iOS devices. All apps submitted to the App Store require approval by Apple.

Apps are submitted to the App Store through a portal called *iTunes Connect*. The [Configure your App in iTunes Connect](#) guide provides more information on how to set up and use this portal to prepare a Xamarin.iOS app for publishing in the App Store.

It is important to note that only developers who belong to the **Apple Developer Program** have access to iTunes Connect. Members of the **Apple Developer Enterprise Program** do not have access.

For more information, please visit the [App Store Distribution](#) guide.

In-house distribution

Sometimes called *Enterprise Distribution*, in-house distribution allows members of the **Apple Developer Enterprise Program** to distribute apps internally to other members of the same organization. In-house distribution has the advantages of not requiring an App Store review, and having no limit on the number of devices on which an application can be installed. However, it is important to note that **Apple Developer Enterprise Program** members do **not** have access to iTunes Connect, and therefore the licensee is responsible for distributing the app.

For more information on getting set-up and how to distribute an application In-House, please refer to the [In-House distribution guide](#).

Ad-hoc distribution

Xamarin.iOS applications can be user-tested via ad hoc distribution, which is available on both the **Apple Developer Program**, and the **Apple Developer Enterprise Program**, and allows up to 100 iOS devices to be tested. The best use case for ad hoc distribution is distribution within a company when iTunes Connect is not an option.

For more information on getting set-up and how to distribute an application In-House, please refer to the [Ad-hoc distribution guide](#).

Custom apps for business

Apple allows [custom distribution](#) of apps to businesses and education. Review the [Apple Business Manager User Guide](#) for information.

Related links

- [App Store distribution](#)
- [Configuring an app in iTunes Connect](#)
- [Publishing to the App Store](#)
- [In-house distribution](#)
- [Ad-hoc distribution](#)
- [The iTunesMetadata.plist File](#)
- [IPA support](#)

- [Troubleshooting](#)

Publishing an Application

8/4/2022 • 3 minutes to read • [Edit Online](#)

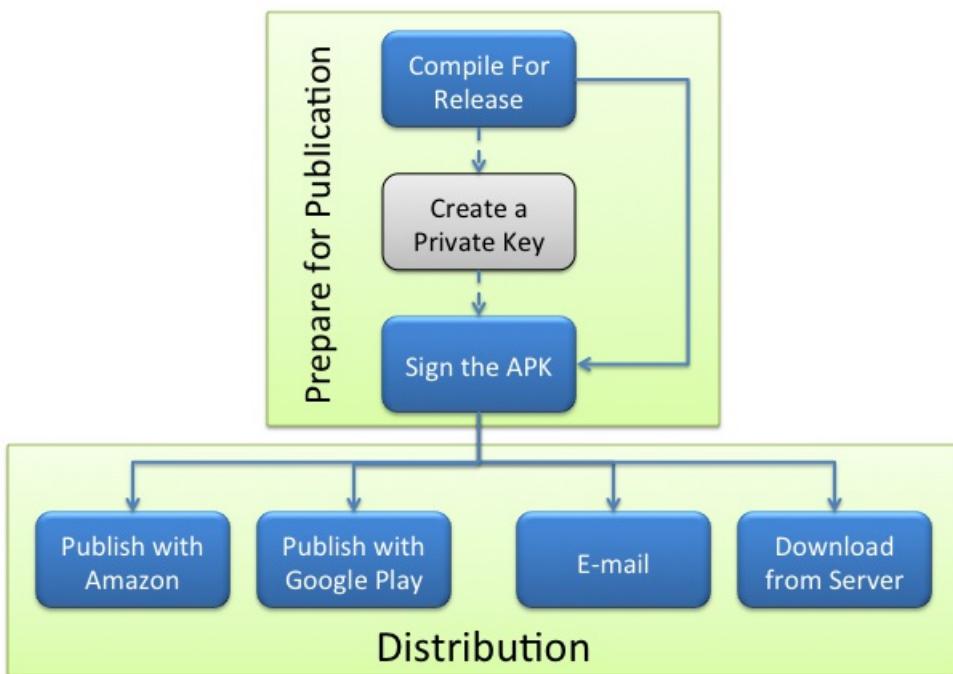
After a great application has been created, people will want to use it. This section covers the steps involved with the public distribution of an application created with Xamarin.Android via channels such as e-mail, a private web server, Google Play, or the Amazon App Store for Android.

Overview

The final step in the development of a Xamarin.Android application is to publish the application. Publishing is the process of compiling a Xamarin.Android application so that it is ready for users to install on their devices, and it involves two essential tasks:

- **Preparing for Publication** – A release version of the application is created that can be deployed to Android-powered devices (see [Preparing an Application for Release](#) for more information about release preparation).
- **Distribution** – The release version of an application is made available through one or more of the various distribution channels.

The following diagram illustrates the steps involved with publishing a Xamarin.Android application:



As can be seen by the diagram above, the preparation is the same regardless of the distribution method that is used. There are several ways that an Android application may be released to users:

- **Via a Website** – A Xamarin.Android application can be made available for download on a website, from which users may then install the application by clicking on a link.
- **By e-mail** – It is possible for users to install a Xamarin.Android application from their e-mail. The application will be installed when the attachment is opened with an Android-powered device.
- **Through a Market** – There are several application marketplaces that exist for distribution, such as [Google Play](#) or [Amazon App Store for Android](#).

Using an established marketplace is the most common way to publish an application as it provides the broadest

market reach and the greatest control over distribution. However, publishing an application through a marketplace requires additional effort.

Multiple channels can distribute a Xamarin.Android application simultaneously. For example, an application could be published on Google Play, the Amazon App Store for Android, and also be downloaded from a web server.

The other two methods of distribution (downloading or e-mail) are most useful for a controlled subset of users, such as an enterprise environment or an application that is only meant for a small or well-specified set of users. Server and e-mail distribution are also simpler publishing models, requiring less preparation to publish an application.

The Amazon Mobile App Distribution Program enables mobile app developers to distribute and sell their applications on Amazon. Users can discover and shop for apps on their Android-powered devices by using the Amazon App Store application. A screenshot of the Amazon App Store running on an Android device appears below:

Google Play is arguably the most comprehensive and popular marketplace for Android applications. Google Play allows users to discover, download, rate, and pay for applications by clicking a single icon either on their device or on their computer. Google Play also provides tools to assist in the analysis of sales and market trends and to control which devices and users may download an application. A screenshot of Google Play running on an Android device appears below:

23°



17:58



Apps



CATEGORIES

FEATURED

TOP PAID

 pocket

Staff Picks



Games



Editors' Choice


TED
Ideas worth spreading hipmunk

flights



This section shows how to upload the application to a store such as Google Play, along with the appropriate promotional materials. APK expansion files are explained, providing a conceptual overview of what they are and how they work. Google Licensing services are also described. Finally, alternate means of distribution are introduced, including the use of an HTTP web server, simple e-mail distribution, and the Amazon App Store for Android.

Related Links

- [HelloWorldPublishing \(sample\)](#)
- [Build Process](#)
- [Linking](#)
- [Obtaining A Google Maps API Key](#)
- [Deploy via Visual Studio App Center](#)
- [Application Signing](#)
- [Publishing on Google Play](#)
- [Google Application Licensing](#)
- [Android.Play.ExpansionLibrary](#)
- [Mobile App Distribution Portal](#)
- [Amazon Mobile App Distribution FAQ](#)

Publishing Xamarin.Mac Apps to the Mac App Store

8/4/2022 • 2 minutes to read • [Edit Online](#)

Overview

Xamarin.Mac apps can be distributed in two different ways:

- **Developer ID** – Applications signed with a Developer ID can be distributed outside of the App Store but are recognized by GateKeeper and allowed to install.
- **Mac App Store** – Apps must have an installer package, and both the app and the installer must be signed, for submission to the Mac App Store.

This document explains how to use Visual Studio for Mac and Xcode to setup a Apple Developer account and configure a Xamarin.Mac project for each deployment type.

Mac developer program

When you join the [Mac Developer Program](#) the developer will be offered a choice to join as an Individual or a Company, as shown in the screenshot below:

Enter Account Info Select Program Review & Submit Agree to License Purchase Program Activate Program

Are you enrolling as an individual or company?

Individual

Select this option if you are an individual or sole proprietor/single person company.



Individual Development Only

You are the only one allowed access to program resources.



App Store Distribution

Your name will appear as the "seller" for apps you distribute on the App Store.

[View example](#)



You will need:

- Credit card billing information.
- A valid credit card for purchase.
We may also require additional personal documentation to verify your identity.

Company

Select this option if you are a company, non-profit organization, joint venture, partnership, or government organization.



Development Team

You can add additional developers to your team who can access program resources. Companies who have hired a contractor to create apps for distribution on the App Store should enroll with their company name and add the contractors to their team.



App Store Distribution

Your legal entity name will appear as the "seller" for apps you distribute on the App Store.

[View example](#)



You will need:

- The legal authority to bind your company/organization to Apple Developer Program legal agreements.
- An address for the company's principal place of business or corporate headquarters.
- A D-U-N-S® Number assigned to a legal entity.
D-U-N-S Numbers, available from D&B for free in most jurisdictions, are unique nine-digit numbers widely used as standard business identifiers. To learn more, read our [FAQs](#). Before enrolling, check to see if D&B has assigned you a D-U-N-S Number. If not, please request one. [Check now ▶](#)
- Note: We do not accept DBAs, Fictitious Business, or Trade names at this time.
- A valid credit card for purchase.

[Individual](#)

[Company](#)

Choose the correct enrollment type for your situation.

NOTE

The choices made here will affect the way some screens appear when configuring a developer account. The descriptions and screenshots in this document are done from the perspective of an **Individual** developer account. In a **Company**, some options will only be available to **Team Admin** users.

Certificates and identifiers

This guide walks through creating the necessary Certificates and Identifiers that will be required to publish a Xamarin.Mac app.

Create provisioning profile

This guide walks through creating the necessary Provisioning Profiles that will be required to publish a Xamarin.Mac app.

Mac app configuration

This guide walks through configuring a Xamarin.Mac app for publication.

Sign with Developer ID

This guide walks through signing a Xamarin.Mac app with a Developer ID for publication.

Bundle for Mac App Store

This guide walks through bundling a Xamarin.Mac app for publication to the Mac App Store.

Upload to Mac App Store

This guide walks through uploading a Xamarin.Mac app for publication to the Mac App Store.

Related links

- [Installation](#)
- [Hello, Mac sample](#)
- [Developer ID and GateKeeper](#)

Xamarin.Forms advanced concepts & internals

8/4/2022 • 2 minutes to read • [Edit Online](#)

Learn about advanced concepts and the internals of Xamarin.Forms.

Controls class hierarchy

Learn about the hierarchy of types used to create the user interface of a Xamarin.Forms application.

Dependency resolution

Learn how to inject a dependency resolution method into Xamarin.Forms, so that an application has control over the creation and lifetime of custom renderers, effect, and `DependencyService` implementations.

Experimental flags

Xamarin.Forms experimental flags enable the engineering team to ship new features to users more quickly, while still being able to change feature APIs before they move to a stable release.

Fast renderers

Learn about fast renderers, which reduce the inflation and rendering costs of a Xamarin.Forms control on Android by flattening the resulting native control hierarchy.

Source Link

Learn how to debug your application into the Xamarin.Forms source code.

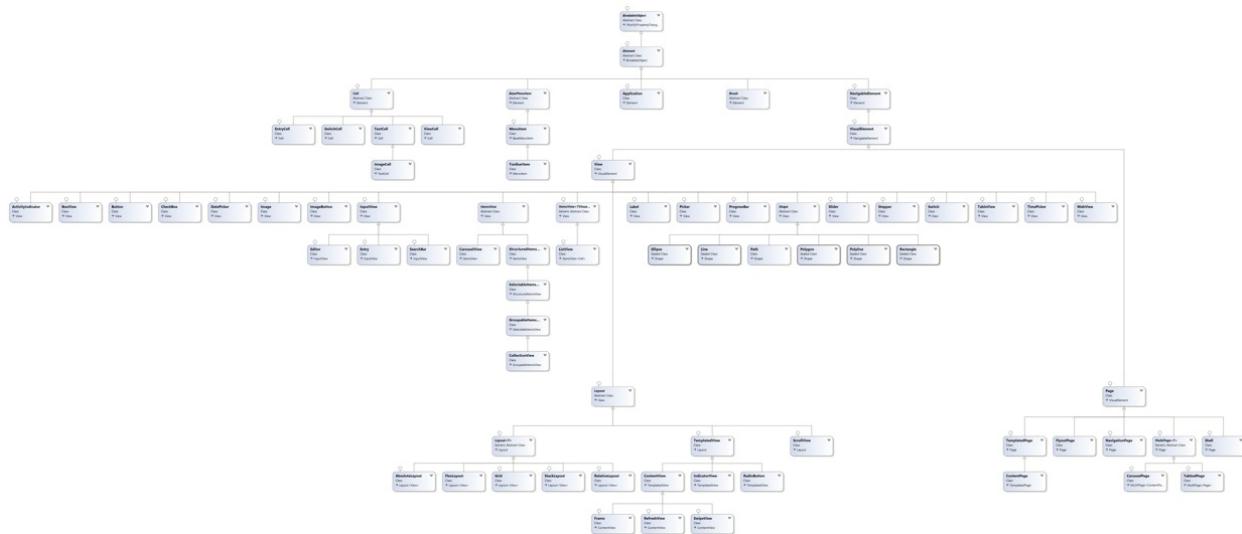
Xamarin.Forms Controls Class Hierarchy

8/4/2022 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms is made up of hundreds of types, over multiple namespaces. Developers should be most familiar with the hierarchy of types used to create the user interface of a Xamarin.Forms application, which reside in the `Xamarin.Forms` namespace.

These types can be divided into pages, layouts, views, and cells. A Xamarin.Forms page generally occupies the entire screen, and all the page types derive from the `Page` class. Pages usually contain a layout, and all the layout types derive from the `Layout` class. A layout usually contains views and possibly other layouts, and all the view types ultimately derive from the `View` class. Finally, cells are specialized controls that are used in display data in the `TableView` and `ListView` controls. Pages, layouts, views, and cells are all ultimately derived from the `Element` class.

The following class diagram shows the hierarchy of types that are typically used to build a user interface in Xamarin.Forms:



However, note that the diagram only shows a single Shell type.

NOTE

A high resolution version of the class diagram can be downloaded from [here](#).

Related links

- [Xamarin.Forms Controls Reference](#)

Dependency resolution in Xamarin.Forms

8/4/2022 • 9 minutes to read • [Edit Online](#)



[Download the sample](#)

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](#)

Xamarin.Forms experimental flags

8/4/2022 • 2 minutes to read • [Edit Online](#)

When a new Xamarin.Forms feature is implemented, it's sometimes put behind an experimental flag. This enables the engineering team to provide new features to you more quickly, while still being able to change feature APIs before they move to a stable release. The experimental flag is then removed once the feature moves to a stable release.

Xamarin.Forms includes the following experimental flags:

- `Shell_UWP_Experimental`

Using functionality that's behind an experimental flag requires you to enable the flag, or flags, in your application. There are two approaches for enabling experimental flags:

- Enable the experimental flag in your platform projects.
- Enable the experimental flag in your `App` class.

WARNING

Consuming functionality that's behind an experimental flag, without enabling the flag, will result in your application throwing an exception that indicates which flag must be enabled.

Enable flags in platform projects

The `xamarin.Forms.Forms.SetFlags` method can be used to enable an experimental flag in your platform projects:

```
Xamarin.Forms.Forms.SetFlags("Shell_UWP_Experimental");
```

The `SetFlags` method should be invoked in your `AppDelegate` class on iOS, in your `MainActivity` class on Android, and in your `App` class on UWP.

IMPORTANT

Enabling an experimental flag in your platform projects must occur before the `Forms.Init` method is invoked.

The `xamarin.Forms.Forms.SetFlags` method accepts a `string` array argument, which makes it possible to enable multiple experimental flags in a single method call:

```
Xamarin.Forms.Forms.SetFlags(new string[] { "Shell_UWP_Experimental", "AnotherFeature_Experimental" });
```

WARNING

Never call the `SetFlags` method more than once, as subsequent calls will overwrite the result of previous calls.

Enable flags in your App class

The `Device.SetFlags` method can be used to enable an experimental flag in the `App` class in your shared code

project:

```
Device.SetFlags(new string[]{ "Shell_UWP_Experimental" });
```

The `Device.SetFlags` method accepts an `IReadOnlyList<string>` argument, which makes it possible to enable multiple experimental flags in a single method call:

```
Device.SetFlags(new string[]{ "Shell_UWP_Experimental", "AnotherFeature_Experimental" });
```

WARNING

Never call the `SetFlags` method more than once, as subsequent calls will overwrite the result of previous calls.

Old experimental flags

The following table lists experimental flags for features that are now in general availability, and the Xamarin.Forms release in which the experimental flag was removed:

FLAG	XAMARIN.FORMS RELEASE
<code>AppTheme_Experimental</code>	4.8
<code>Brush_Experimental</code>	5.0
<code>CarouselView_Experimental</code>	5.0
<code>CollectionView_Experimental</code>	4.3
<code>DragAndDrop_Experimental</code>	5.0
<code>FastRenderers_Experimental</code>	4.0
<code>IndicatorView_Experimental</code>	4.7
<code>Markup_Experimental</code>	5.0 (moved to Xamarin Community Toolkit)
<code>MediaElement_Experimental</code>	5.0 (moved to Xamarin Community Toolkit)
<code>RadioButton_Experimental</code>	5.0
<code>Shapes_Experimental</code>	5.0
<code>Shell_Experimental</code>	4.0
<code>StateTriggers_Experimental</code>	4.7
<code>SwipeView_Experimental</code>	5.0
<code>Visual_Experimental</code>	3.6

Xamarin.Forms Fast Renderers

8/4/2022 • 2 minutes to read • [Edit Online](#)

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 on Android:

- `Button`
- `Frame`
- `Image`
- `Label`

Functionally, these fast renderers are no different to the legacy renderers. From Xamarin.Forms 4.0 onwards, all applications targeting `FormsAppCompatActivity` will use these fast renderers by default. Renderers for all new controls, including `ImageButton` and `CollectionView`, use the fast renderer approach.

Performance improvements when using fast renderers 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.

NOTE

Custom renderers can be created for fast renderers using the same approach as used for the legacy renderers. For more information, see [Custom Renderers](#).

Backwards compatibility

Fast renderers can be overridden with the following approaches:

1. Enabling the legacy renderers by adding the following line of code to your `MainActivity` class before calling `Forms.Init`:

```
Forms.SetFlags("UseLegacyRenderers");
```

2. Using custom renderers that target the legacy renderers. Any existing custom renderers will continue to function with the legacy renderers.
3. Specifying a different `View.Visual`, such as `Material`, that uses different renderers. For more

information about Material Visual, see [Xamarin.Forms Material Visual](#).

Related links

- [Custom Renderers](#)

Source Link with Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)

Xamarin.Forms NuGet packages include Source Link mappings. Source Link maps compiled libraries, contained in a NuGet package, to a source code repository. Visual Studio will download source code files during debugging and allow developers to step through code, enabling debugging of packages without building from source.

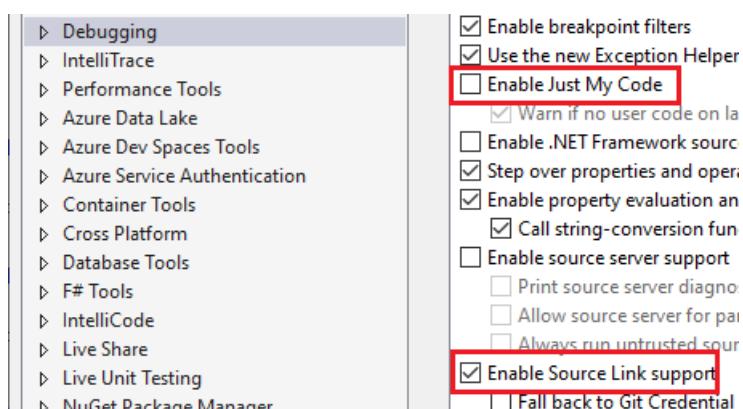
For more information about using Source Link, see [Source Link Documentation](#).

WARNING

Visual Studio 2019 supports Source Link for the **.NET debugger** but does not currently support Source Link for the **Mono debugger**. Therefore, you can use Source Link to debug UWP apps, but not Android or iOS app. When debugging UWP apps you must ensure that the PDB files for libraries you want to debug are copied to the **AppX** folder in the **bin** directory where your app is compiled.

Enable Source Link

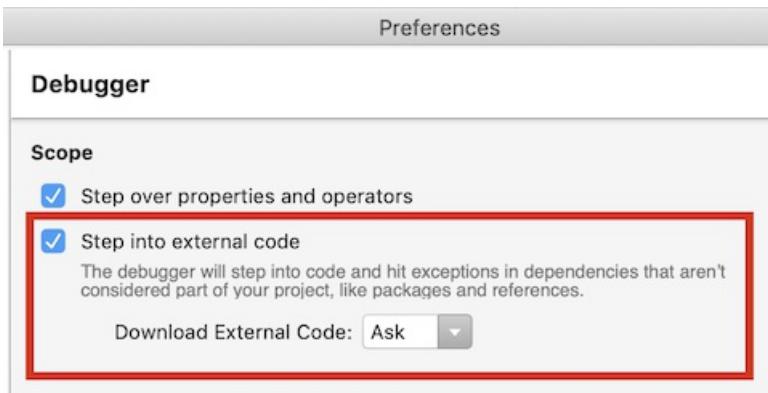
Using Source Link requires enabling debugging for external code, otherwise the debugger will step past calls to code not contained in the current solution. In Visual Studio 2019 this can be found under the **Options** menu in the **Debugging** section:



Ensure that **Enable just my code** is disabled and that **Enable Source Link support** is enabled.

Enable Source Link

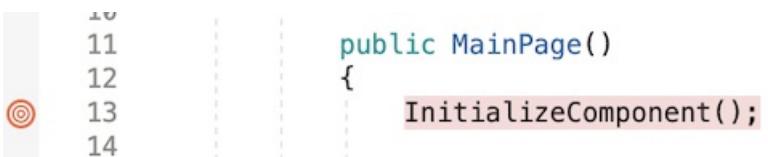
Using Source Link requires enabling debugging for external code, otherwise the debugger will step past calls to code not contained in the current solution. This option can be found in the **Preferences** window in the **Debugger** section:



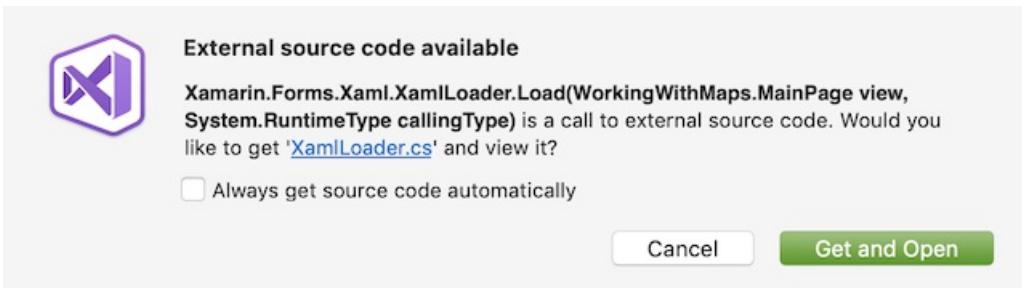
Ensure that **Step into external code** is enabled.

Debug Xamarin.Forms using Source Link

If debugging external packages is enabled, Visual Studio will use the Source Link mappings contained in the NuGet package to download and step through external source code. This can be tested by setting a breakpoint on a call to a method provided by Xamarin.Forms:



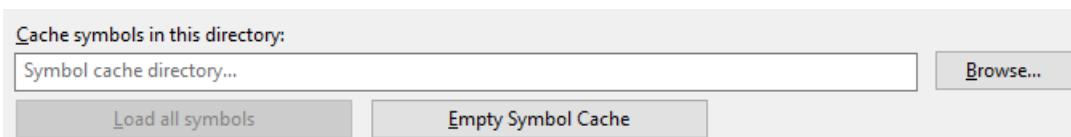
Depending on the settings you specified in the **Debugger** options, Visual Studio will warn you that it is downloading source files:



Once you allow Visual Studio to download the files, the debugger will step into the external code.

Source Link caching

Source Link uses caching for performance. The caching directory for Source link is defined in the **Options** menu under **Debugging** in the **Symbols** section:



This menu allows you to specify the caching directory for all debug symbols, as well as clear the cache if you encounter issues with cached symbols.

Source Link caching

Source Link uses caching for performance. The caching directory for Source Link on MacOS is `/Users/<username>/Library/Caches/VisualStudio/8.0/Symbols`. This folder contains subfolders that store the repository used to download source files. If the backing repository for a NuGet package has changed, you may need to manually delete these folders to refresh the cache.

Related links

- [Source Link Documentation](#)
- [Source Link on GitHub](#)

Troubleshooting Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)

Common error conditions and how to resolve them

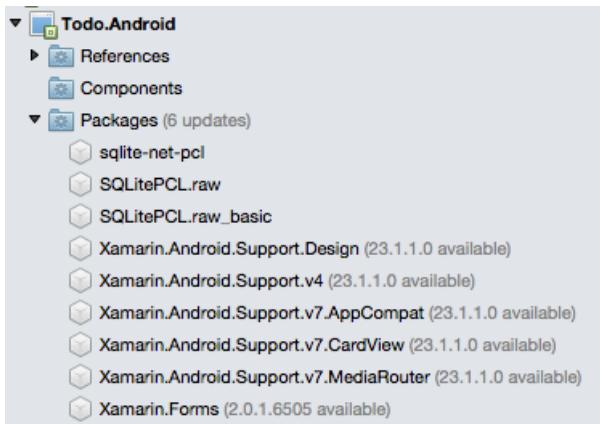
Error: "Unable to find a version of Xamarin.Forms compatible with..."

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.

How do I migrate my app to Xamarin.Forms 5.0?

8/4/2022 • 3 minutes to read • [Edit Online](#)

Xamarin.Forms 5.0 includes the following breaking changes:

- `Expander` has moved to the Xamarin Community Toolkit. For more information, see [Features moved from Xamarin.Forms](#).
- `MediaElement` has moved to the Xamarin Community Toolkit. For more information, see [Features moved from Xamarin.Forms](#).
- DataPages, and associated projects, have been removed from Xamarin.Forms.
- `MasterDetailPage` has been renamed to `FlyoutPage`. Similarly, the `MasterBehavior` enumeration has been renamed to `FlyoutLayoutBehavior`.
- References to `UIWebView` have been removed from Xamarin.Forms on iOS.
- Support for Visual Studio 2017 has been removed.
- `XFCorePostProcessor.Tasks` has been removed. This project injected IL to maintain Xamarin.Forms 2.5 compatibility.

In addition, Android and UWP projects built with Xamarin.Forms 5.0 will require updating.

IMPORTANT

When updating an application to Xamarin.Forms 5.0, ensure that you update each project that references the Xamarin.Forms NuGet package to an identical version.

Android

Android projects built with Xamarin.Forms 5.0 require that you've installed the AndroidX (Android 10.0) platform to your development environment. This can be accomplished with the Android SDK manager. For more information about AndroidX, see [AndroidX migration in Xamarin.Forms](#).

Android projects will then require several updates to build correctly.

Minimum TargetFrameworkVersion

Xamarin.Forms 5.0 requires a minimum target framework version of 10.0 (AndroidX) for Android projects. The target framework version can be set in Visual Studio, or in the Android .csproj file:

```
<TargetFrameworkVersion>v10.0</TargetFrameworkVersion>
```

A build error will be produced if this minimum requirement isn't met:

```
error XF005: The $(TargetFrameworkVersion) for MyProject.Android (v9.0) is less than the minimum required $(TargetFrameworkVersion) for Xamarin.Forms (10.0). You need to increase the $(TargetFrameworkVersion) for MyProject.Android.
```

Minimum TargetSdkVersion

AndroidX requires that your Android manifest sets the `targetSdkVersion` to 29+:

```
<uses-sdk android:minSdkVersion="21" android:targetSdkVersion="29" />
```

Failure to do this will cause the `targetSdkVersion` to be set to the `minSdkVersion`. In addition, in some circumstances a `Android.Views.InflateException` will be produced if the `targetSdkVersion` isn't correctly set:

```
Android.Views.InflateException has been thrown.
```

```
Binary XML file line #1 in com.companyname.myproject:layout/toolbar: Binary XML file line #1 in  
com.companyname.myproject:/layout/toolbar: Error inflating class android.support.v7.widget.Toolbar
```

NOTE

In Visual Studio 2019, the Android manifest will automatically be updated to specify a `targetSdkVersion` of API 29 when the target framework version is set to v10.0.

Automatic migration to AndroidX

If your Android project references any Android support libraries, either as direct dependencies or transitive dependencies, these support library dependencies and bindings are automatically swapped with AndroidX dependencies during the build process. For more information about automatic AndroidX migration, see [Automatic migration in Xamarin.Forms](#).

Manual migration to AndroidX

If your Android project doesn't have direct or transitive dependencies on Android support libraries, but still attempts to consume support library types through code, you will have to manually migrate your app to AndroidX. This can be accomplished by using AndroidX types, and by removing any AXML files that you haven't customized.

TIP

Manual migration to AndroidX will result in the fastest build process for your app.

Use AndroidX types

AndroidX replaces the Android support libraries, and so any references to Android support library types must be replaced with references to AndroidX types.

This can be accomplished by updating your `using` statements to use `AndroidX` namespaces, rather than `Android.Support` namespaces. The following table lists some of the common namespace changes when moving from the Android support libraries to AndroidX:

ANDROID SUPPORT LIBRARY NAMESPACE	ANDROIDX NAMESPACE
<code>Android.Support.V4.App</code>	<code>AndroidX.Core.App</code>
<code>Android.Support.V4.Content</code>	<code>AndroidX.Core.Content</code>
<code>Android.Support.V4.App</code>	<code>AndroidX.Fragment.App</code>
<code>Android.Support.V7.App</code>	<code>AndroidX.AppCompat.App</code>
<code>Android.Support.V7.Widget</code>	<code>AndroidX.AppCompat.Widget</code>

For a complete list of class mappings from support libraries to AndroidX, see [AndroidX class mappings](#) on github.com. For a complete list of assembly mappings from support libraries to AndroidX, see [AndroidX assemblies](#) on github.com

Remove AXML files

You should delete any AXML files from your Android project, provided that it doesn't use customized AXML files. After deletion, the following lines should be removed from your `MainActivity` class:

```
TabLayoutResource = Resource.Layout.Tabbar;  
ToolbarResource = Resource.Layout.Toolbar;
```

IMPORTANT

Android projects with customized AXML files should be updated so that these files use AndroidX types.

UWP

Xamarin.Forms 5.0 recommends a target platform version of $\geq 10.0.18362.0$ for UWP projects. The target platform version can be set in Visual Studio, or in the UWP .csproj file:

```
<TargetPlatformVersion Condition=" '$(TargetPlatformVersion)' == '' ">10.0.18362.0</TargetPlatformVersion>
```

A build warning will be produced if your UWP project uses a lower target platform version.

Related links

- [Features moved from Xamarin.Forms](#)
- [AndroidX migration in Xamarin.Forms](#)
- [AndroidX class mappings](#)
- [AndroidX assemblies](#)

Can I update the Xamarin.Forms default template to a newer NuGet package?

8/4/2022 • 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 Xamarin.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:

```
<Name>Blank App (Xamarin.Forms Portable) - 2.1.0.6529</Name>
```

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,
<https://nuget.org/api/v2/package/Xamarin.Forms/2.1.0.6529> (see also
<https://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?

8/4/2022 • 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 [Xamarin.Forms XAML Basics](#) guide:

"There is no visual designer for generating XAML in Xamarin.Forms applications, so all XAML must be hand-written."

However, the Xamarin.Forms XAML Editor can be displayed by selecting the **View > Other Windows > Xamarin.Forms Editor** menu option.

Android build error – The LinkAssemblies task failed unexpectedly

8/4/2022 • 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 supported platforms](#))

The resolution to this issue is to make sure you have the latest supported Android SDK versions, and set the **Target Framework** to the latest installed platform. It's also recommended that you set the **Target Android Version** to the latest installed platform, and the **minimum Android version** to API 19 or higher. This is considered the supported configuration.

Setting in Visual Studio for Mac

1. Right click on the Android project, and select **Options** in the menu.
2. In the **Project Options** dialog, go to **Build > General**.
3. Set the **Compile using Android version: (Target Framework)** to the latest installed platform.
4. In the **Project Options** dialog, go to **Build > Android Application**.
5. Set the **Minimum Android version** to API level 19 or higher, and the **Target Android version** to the latest installed platform you chose in (3).

Setting in Visual Studio

1. Right click on the Android project, and select **Properties** in the menu.
2. In the project properties, go to **Application**.
3. Set the **Compile using Android version: (Target Framework)** to the latest installed platform.
4. In the project properties, go to **Android Manifest**.
5. Set the **Minimum Android version** to API level 19 or higher, and the **Target Android version** to the latest installed platform you chose in (3).

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?

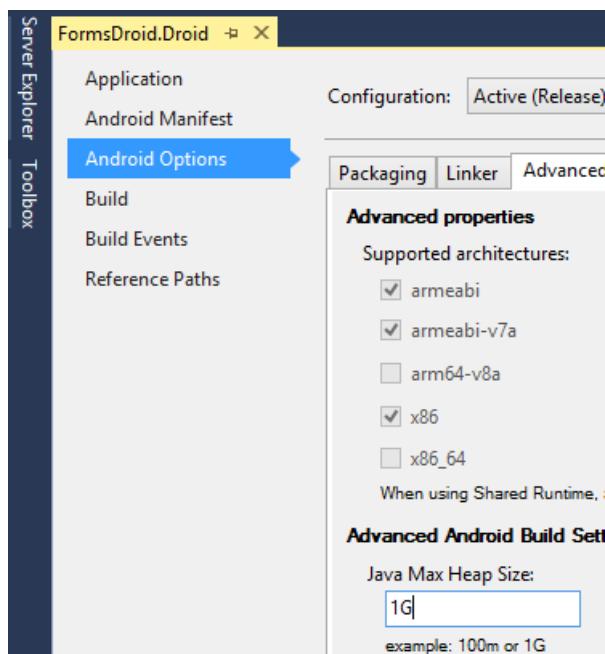
8/4/2022 • 2 minutes to read • [Edit Online](#)

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:

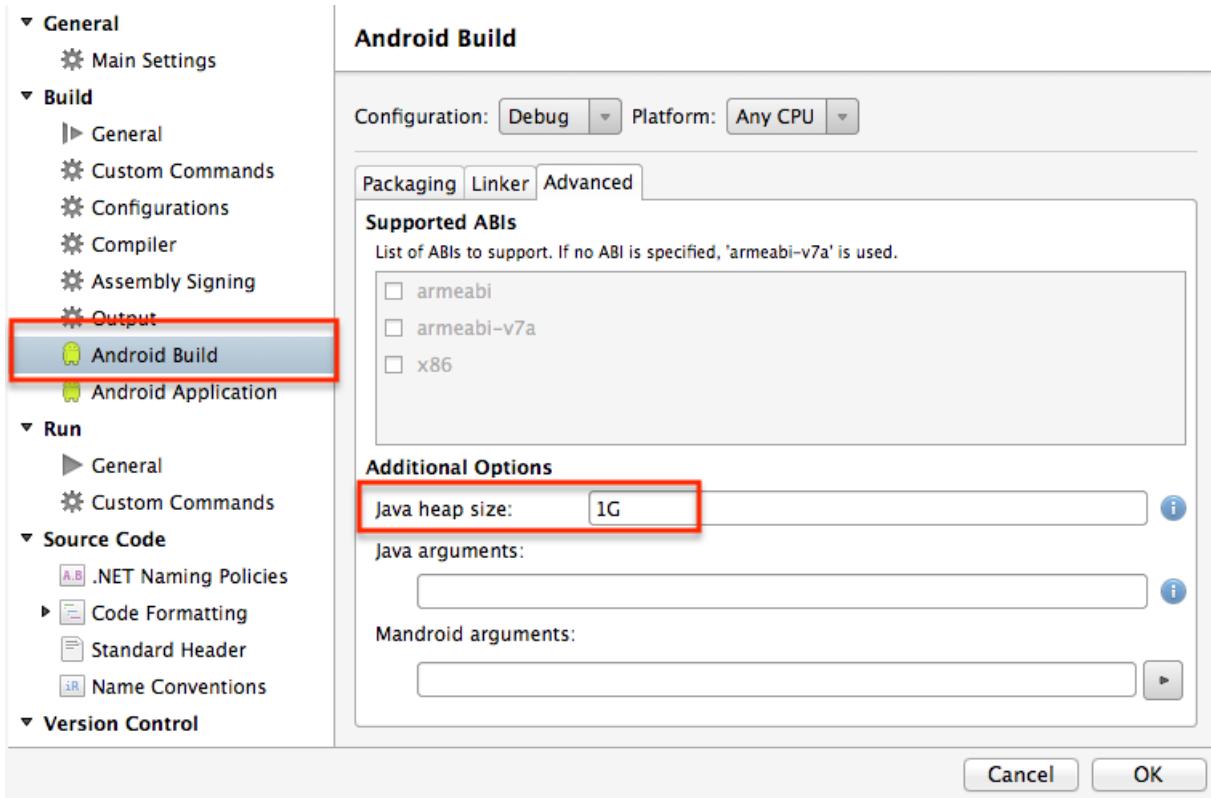
Visual Studio

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.



Xamarin.Forms Samples

8/4/2022 • 2 minutes to read • [Edit Online](#)

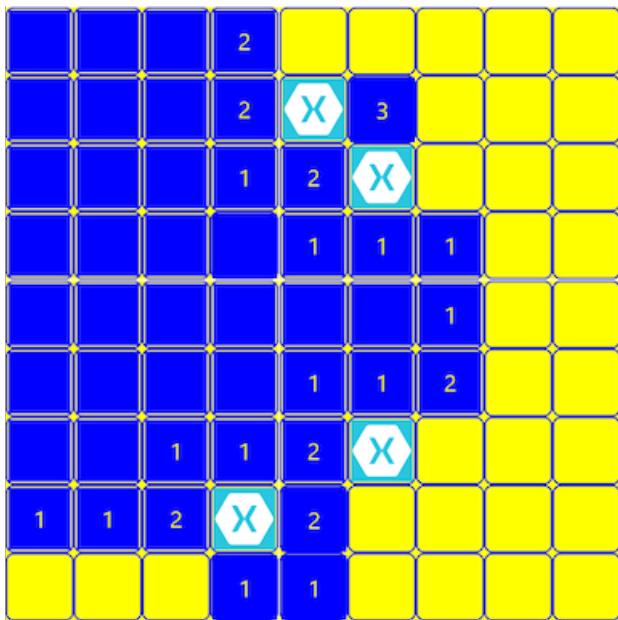
Xamarin.Forms sample apps and code demos to help you get started and understand concepts in Xamarin.Forms.

[All Xamarin.Forms samples](#)



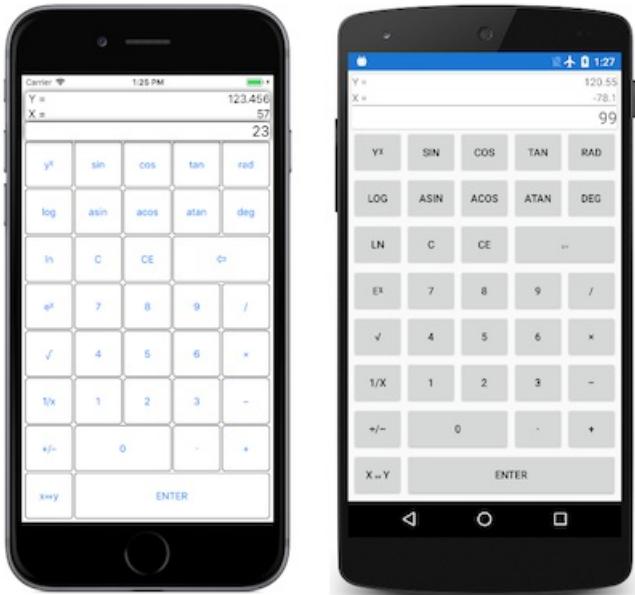
Todo

This sample demonstrates a Todo list application where the data is stored and accessed in a local SQLite database.



BugSweeper

This is a familiar game with a new twist. Ten bugs are hidden in a 9-by-9 grid of tiles. To win, you must find and flag all ten bugs.



RPN Calculator

An RPN (Reverse Polish Notation) calculator allows numbers and operations to be entered without parentheses or an equal key.



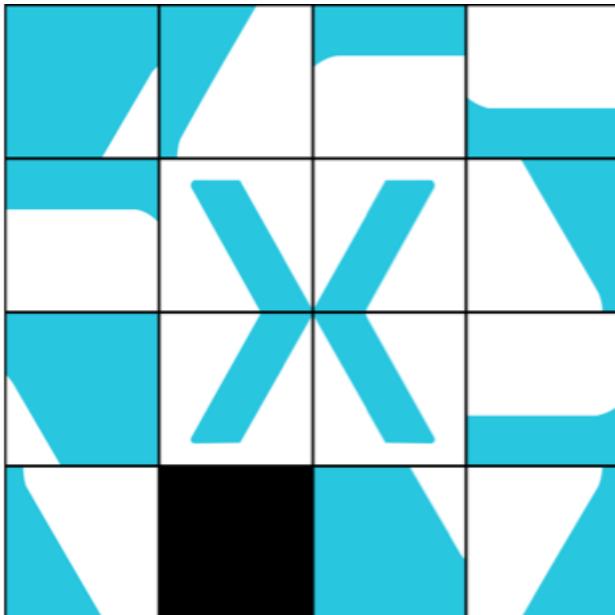
SpinPaint

The program simulates a revolving disk that you can paint on by touching and moving your finger. SpinPaint responds to touch by painting a line under your finger, but it also duplicates that line in three mirror images in the other three quadrants of the disk.



XAML Samples

XAML—the eXtensible Application Markup Language—allows developers to define user interfaces in Xamarin.Forms applications using markup rather than code.



Xuzzle

This game is a variation of the classic 14-15 puzzle that you can solve by sliding tiles into the correct order.

All samples

For the complete set of Xamarin.Forms sample apps and code demos, see [All Xamarin.Forms samples](#).

Creating Mobile Apps with Xamarin.Forms book

8/4/2022 • 5 minutes to read • [Edit Online](#)



[Download the sample](#)



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 chapter summaries 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 shown below. These summaries describe the contents of each chapter, and include several types of links:

- Link to the actual eBook (Free download)
- 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 entire eBook or view a chapter summary

CHAPTER	COMPLETE EBOOK	CHAPTER SUMMARY
Chapter 1. How Does Xamarin.Forms Fit In?	Download book PDF	Summary

CHAPTER	COMPLETE EBOOK	CHAPTER SUMMARY
Chapter 2. Anatomy of an App	Download book PDF	Summary
Chapter 3. Deeper into Text	Download book PDF	Summary
Chapter 4. Scrolling the Stack	Download book PDF	Summary
Chapter 5. Dealing with Sizes	Download book PDF	Summary
Chapter 6. Button Clicks	Download book PDF	Summary
Chapter 7. XAML vs. Code	Download book PDF	Summary
Chapter 8. Code and XAML in Harmony	Download book PDF	Summary
Chapter 9. Platform-Specific API Calls	Download book PDF	Summary
Chapter 10. XAML Markup Extensions	Download book PDF	Summary
Chapter 11. The Bindable Infrastructure	Download book PDF	Summary
Chapter 12. Styles	Download book PDF	Summary
Chapter 13. Bitmaps	Download book PDF	Summary
Chapter 14. Absolute Layout	Download book PDF	Summary
Chapter 15. The Interactive Interface	Download book PDF	Summary
Chapter 16. Data Binding	Download book PDF	Summary
Chapter 17. Mastering the Grid	Download book PDF	Summary
Chapter 18. MVVM	Download book PDF	Summary
Chapter 19. Collection Views	Download book PDF	Summary
Chapter 20. Async and File I/O	Download book PDF	Summary
Chapter 21. Transforms	Download book PDF	Summary
Chapter 22. Animation	Download book PDF	Summary
Chapter 23. Triggers and Behaviors	Download book PDF	Summary
Chapter 24. Page Navigation	Download book PDF	Summary
Chapter 25. Page Varieties	Download book PDF	Summary

CHAPTER	COMPLETE EBOOK	CHAPTER SUMMARY
Chapter 26. Custom Layouts	Download book PDF	Summary
Chapter 27. Custom renderers	Download book PDF	Summary
Chapter 28. Location and Maps	Download book PDF	Summary

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 main branch contains projects that have been upgraded to remove deprecated APIs and reflect enhanced APIs. In addition, the Android projects in the main 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

8/4/2022 • 4 minutes to read • [Edit Online](#)

Architectural guidance for developing adaptable, maintainable, and testable Xamarin.Forms enterprise applications



NOTE

This eBook was published in the spring of 2017, and has not been updated since then. There is much in the book that remains valuable, but some of the material is outdated.

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.

Community Site

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.

Related Links

- [Download eBook \(2Mb PDF\)](#)
- [eShopOnContainers \(GitHub\) \(sample\)](#)

SkiaSharp Graphics in Xamarin.Forms

8/4/2022 • 2 minutes to read • [Edit Online](#)



[Download the sample](#)

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.

This guide assumes that you are familiar with Xamarin.Forms programming.

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, edit its **Info.plist** file 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 \(sample\)](#)