

Contents

.NET MAUI

[What is .NET Multi-platform App UI?](#)

[Supported platforms](#)

[Get started](#)

[Build your first app](#)

[Learning resources](#)

[Migrate from Xamarin.Forms](#)

[Android](#)

[Emulator](#)

[How to enable hardware acceleration](#)

[Manage and create virtual devices](#)

[Edit virtual devices](#)

[Debug on a virtual device](#)

[Troubleshoot common problems](#)

[Devices](#)

[How to set up a physical device](#)

[iOS](#)

[Build an iOS app with .NET CLI](#)

[Pair to Mac](#)

[Simulators](#)

[Remote iOS simulator for Windows](#)

[macOS](#)

[Build a Mac Catalyst app with .NET CLI](#)

[Windows](#)

[How to set up Windows for debugging](#)

[Tutorials](#)

[Create a .NET MAUI app](#)

[XAML](#)

[Overview](#)

Fundamentals

[Get started](#)

[Essential syntax](#)

[Markup extensions](#)

[Data binding basics](#)

[Data binding and MVVM](#)

[Compilation](#)

[Field modifiers](#)

[Generics](#)

[Markup extensions](#)

[Consume markup extensions](#)

[Create markup extensions](#)

[Namespaces](#)

[Overview](#)

[Custom namespace schemas](#)

[Custom namespace prefixes](#)

[Pass arguments](#)

[Runtime loading](#)

[Tooling](#)

[XAML Hot Reload](#)

Fundamentals

[Accessibility](#)

[App lifecycle](#)

[Behaviors](#)

[Data binding](#)

[Overview](#)

[Basic bindings](#)

[Binding mode](#)

[String formatting](#)

[Binding path](#)

[Binding value converters](#)

[Relative bindings](#)

- Binding fallbacks
- Multi-bindings
- Commanding
- Compiled bindings

Gestures

- Drag and drop
- Pan
- Pinch
- Swipe
- Tap

Properties

- Bindable properties
- Attached properties

Publish and subscribe to messages

Resource dictionaries

Shell

- Overview
- Create a Shell app
- Flyout
- Tabs
- Pages
- Navigation
- Search
- Lifecycle

Single project

Templates

- Control templates
- Data templates

Triggers

Windows

User interface

- Animation

- [Basic animation](#)
- [Easing functions](#)
- [Custom animation](#)
- [Brushes](#)
 - [Overview](#)
 - [Solid colors](#)
 - [Gradients](#)
 - [Overview](#)
 - [Linear gradients](#)
 - [Radial gradients](#)
- [Controls](#)
 - [Overview](#)
 - [Align and position controls](#)
 - [Handlers](#)
 - [Overview](#)
 - [Create custom controls](#)
 - [Customize controls](#)
- [Layouts](#)
 - [Overview](#)
 - [AbsoluteLayout](#)
 - [BindableLayout](#)
 - [FlexLayout](#)
 - [Grid](#)
 - [HorizontalStackLayout](#)
 - [StackLayout](#)
 - [VerticalStackLayout](#)
- [Pages](#)
 - [ContentPage](#)
 - [FlyoutPage](#)
 - [NavigationPage](#)
 - [TabbedPage](#)
- [Views](#)

Present data

[BlazorWebView](#)

[Border](#)

[BoxView](#)

[Frame](#)

[GraphicsView](#)

[Image](#)

[Label](#)

[ScrollView](#)

[Shapes](#)

[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](#)

[CollectionView](#)

[IndicatorView](#)

[ListView](#)

[Picker](#)

[TableView](#)

[ContentView](#)

[Display a menu bar](#)

[Display pop-ups](#)

[Fonts](#)

[Graphics](#)

[Overview](#)

[Blend modes](#)

[Colors](#)

[Draw graphical objects](#)

[Images](#)

[Paint graphical objects](#)

[Transforms](#)

[Winding modes](#)

[Images](#)

[App icons](#)

[Images](#)

[Splash screen](#)

[Shadows](#)

[Styles](#)

[Style apps using XAML](#)

[Style apps using CSS](#)

[Theming](#)

[Theme an app](#)

[Respond to system theme changes](#)

[Tooling](#)

[Inspect the visual tree](#)

[Visual states](#)

[Platform integration](#)

[Overview](#)

[Application model](#)

[App actions](#)

[App information](#)

[Browser](#)

[Launcher](#)

[Main thread](#)

[Maps](#)

[Permissions](#)

[Version tracking](#)

[Communication](#)

[Contacts](#)

[Email](#)

[Networking](#)

[Phone dialer](#)

[SMS \(messaging\)](#)

[Web authenticator](#)

[Device features](#)

[Battery](#)

[Device display](#)

[Device information](#)

[Device sensors](#)

[Flashlight](#)

[Geocoding](#)

[Geolocation](#)

[Haptic feedback](#)

[Vibration](#)

[Media](#)

[Media picker](#)

[Screenshot](#)

[Text-to-speech](#)

[Unit converters](#)

[Platform-specifics](#)

[Android](#)

[Overview](#)

[Entry input method editor options](#)

[ListView fast scrolling](#)

[Soft keyboard input mode](#)

[SwipeView swipe transition mode](#)

[TabbedPage page swiping](#)

[TabbedPage page transition animations](#)

[TabbedPage toolbar placement](#)

[WebView mixed content](#)

[WebView zoom](#)

[iOS](#)

[Overview](#)

[Cell background color](#)

[DatePicker item selection](#)

[Entry cursor color](#)

[Entry font size](#)

[FlyoutPage shadow](#)

[Large page titles](#)

[ListView group header style](#)

[ListView row animations](#)

[ListView separator style](#)

[Modal page presentation style](#)

[NavigationPage bar translucency](#)

[Picker item selection](#)

[ScrollView content touches](#)

[SearchBar style](#)

[Simultaneous pan gesture recognition](#)

[Slider thumb tap](#)

- [SwipeView swipe transition mode](#)
- [TabPage translucent tabbar](#)
- [TimePicker item selection](#)
- [Windows](#)
 - [Overview](#)
 - [Default image directory](#)
 - [InputView reading order](#)
 - [ListView SelectionMode](#)
 - [RefreshView pull direction](#)
 - [SearchBar spell check](#)
 - [VisualElement access keys](#)
- [Sharing](#)
 - [Clipboard](#)
 - [Share files and text](#)
- [Storage](#)
 - [File picker](#)
 - [File system helpers](#)
 - [Preferences](#)
 - [Secure storage](#)
 - [Configure multi-targeting](#)
 - [Invoke platform code](#)
- [Data & cloud services](#)
 - [Local databases](#)
 - [Web services](#)
 - [Consume a REST-based web service](#)
 - [Connect to local web services](#)
- [Deployment](#)
 - [Overview](#)
 - [Hot restart](#)
 - [Project configuration](#)
 - [Publish](#)
 - [Android](#)

iOS

[Publish an iOS app](#)

[Provision for app store distribution](#)

[Entitlements and capabilities](#)

macOS

[Windows](#)

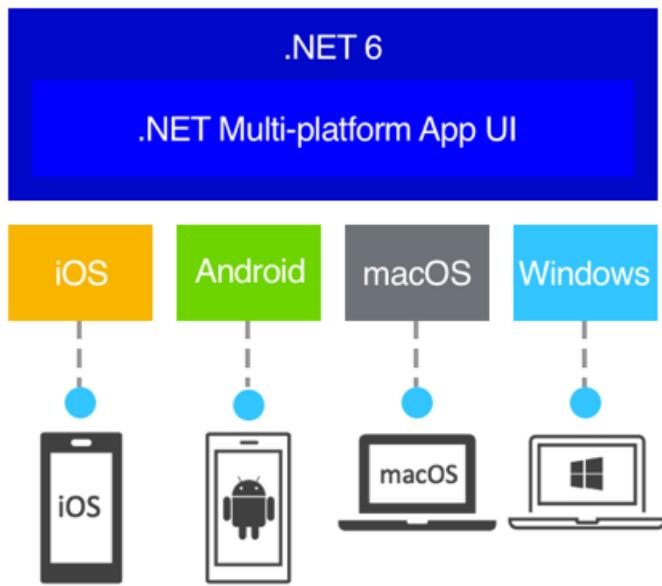
[Troubleshooting](#)

What is .NET MAUI?

9/20/2022 • 5 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) is a cross-platform framework for creating native mobile and desktop apps with C# and XAML.

Using .NET MAUI, you can develop apps that can run on Android, iOS, macOS, and Windows from a single shared code-base.



.NET MAUI is open-source and is the evolution of Xamarin.Forms, extended from mobile to desktop scenarios, with UI controls rebuilt from the ground up for performance and extensibility. If you've previously used Xamarin.Forms to build cross-platform user interfaces, you'll notice many similarities with .NET MAUI. However, there are also some differences. Using .NET MAUI, you can create multi-platform apps using a single project, but you can add platform-specific source code and resources if necessary. One of the key aims of .NET MAUI is to enable you to implement as much of your app logic and UI layout as possible in a single code-base.

Who .NET MAUI is for

.NET MAUI is for developers who want to:

- Write cross-platform apps in XAML and C#, from a single shared code-base in Visual Studio.
- Share UI layout and design across platforms.
- Share code, tests, and business logic across platforms.

How .NET MAUI works

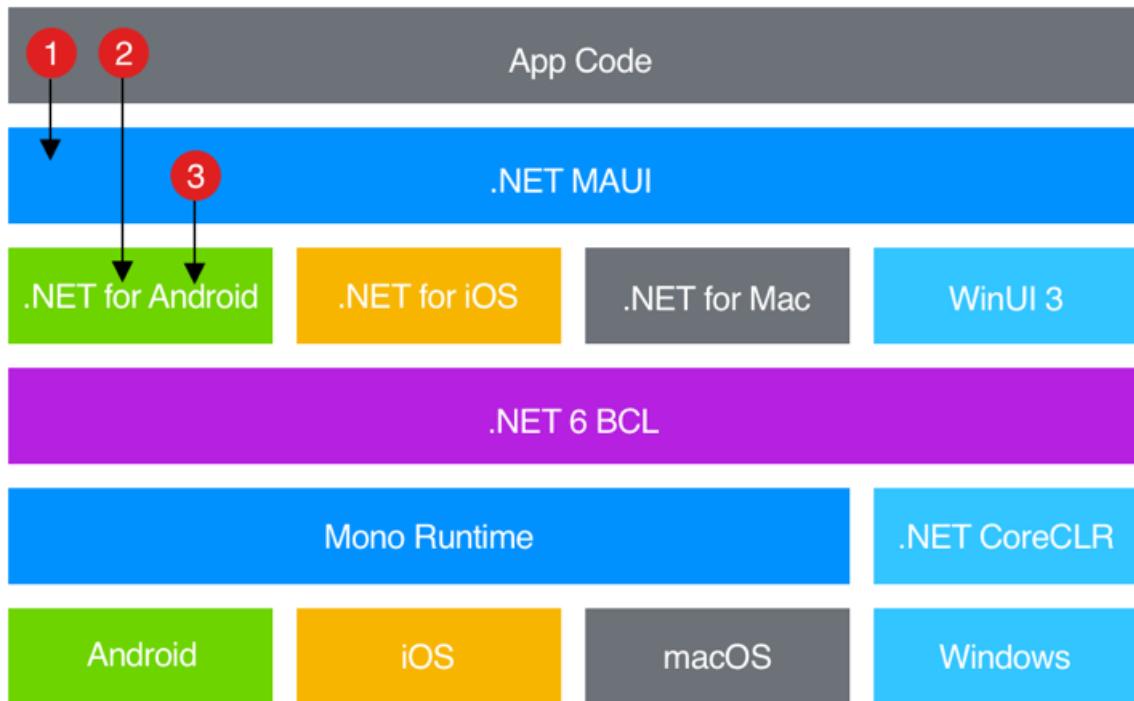
.NET MAUI unifies Android, iOS, macOS, and Windows APIs into a single API that allows a write-once run-anywhere developer experience, while additionally providing deep access to every aspect of each native platform.

.NET 6 provides a series of platform-specific frameworks for creating apps: .NET for Android, .NET for iOS, .NET for macOS, and Windows UI 3 (WinUI 3) library. These frameworks all have access to the same .NET 6 Base Class Library (BCL). This library abstracts the details of the underlying platform away from your code. The BCL depends on the .NET runtime to provide the execution environment for your code. For Android, iOS, and macOS,

the environment is implemented by Mono, an implementation of the .NET runtime. On Windows, .NET CoreCLR provides the execution environment.

While the BCL enables apps running on different platforms to share common business logic, the various platforms have different ways of defining the user interface for an app, and they provide varying models for specifying how the elements of a user interface communicate and interoperate. You can craft the UI for each platform separately using the appropriate platform-specific framework (.NET for Android, .NET for iOS, .NET for macOS, or WinUI 3), but this approach then requires you to maintain a code-base for each individual family of devices.

.NET MAUI provides a single framework for building the UIs for mobile and desktop apps. The following diagram shows a high-level view of the architecture of a .NET MAUI app:



In a .NET MAUI app, you write code that primarily interacts with the .NET MAUI API (1). .NET MAUI then directly consumes the native platform APIs (3). In addition, app code may directly exercise platform APIs (2), if required.

.NET MAUI apps can be written on PC or Mac, and compile into native app packages:

- Android apps built using .NET MAUI compile from C# into intermediate language (IL) which is then just-in-time (JIT) compiled to a native assembly when the app launches.
- iOS apps built using .NET MAUI are fully ahead-of-time (AOT) compiled from C# into native ARM assembly code.
- macOS apps built using .NET MAUI use Mac Catalyst, a solution from Apple that brings your iOS app built with UIKit to the desktop, and augments it with additional AppKit and platform APIs as required.
- Windows apps built using .NET MAUI use Windows UI 3 (WinUI 3) library to create native apps that target the Windows desktop. For more information about WinUI 3, see [Windows UI Library](#).

NOTE

Building apps for iOS and macOS requires a Mac.

What .NET MAUI provides

.NET MAUI provides a collection of controls that can be used to display data, initiate actions, indicate activity, display collections, pick data, and more. In addition to a collection of controls, .NET MAUI also provides:

- An elaborate layout engine for designing pages.
- Multiple page types for creating rich navigation types, like drawers.
- Support for data-binding, for more elegant and maintainable development patterns.
- The ability to customize handlers to enhance the way in which UI elements are presented.
- Cross-platform APIs for accessing native device features. These APIs enable apps to access device features such as the GPS, the accelerometer, and battery and network states. For more information, see [Cross-platform APIs for device features](#).
- Cross-platform graphics functionality, that provides a drawing canvas that supports drawing and painting shapes and images, compositing operations, and graphical object transforms.
- A single project system that uses multi-targeting to target Android, iOS, macOS, and Windows. For more information, see [.NET MAUI Single project](#).
- .NET hot reload, so that you can modify both your XAML and your managed source code while the app is running, then observe the result of your modifications without rebuilding the app. For more information, see [.NET hot reload](#).

Cross-platform APIs for device features

.NET MAUI provides cross-platform APIs for native device features. Examples of functionality provided by .NET MAUI for accessing device features includes:

- Access to sensors, such as the accelerometer, compass, and gyroscope on devices.
- Ability to check the device's network connectivity state, and detect changes.
- Provide information about the device the app is running on.
- Copy and paste text to the system clipboard, between apps.
- Pick single or multiple files from the device.
- Store data securely as key/value pairs.
- Utilize built-in text-to-speech engines to read text from the device.
- Initiate browser-based authentication flows that listen for a callback to a specific app registered URL.

Single project

.NET MAUI single project takes the platform-specific development experiences you typically encounter while developing apps and abstracts them into a single shared project that can target Android, iOS, macOS, and Windows.

.NET MAUI single project provides a simplified and consistent cross-platform development experience, regardless of the platforms being targeted. .NET MAUI single project provides the following features:

- A single shared project that can target Android, iOS, macOS, and Windows.
- A simplified debug target selection for running your .NET MAUI apps.
- Shared resource files within the single project.
- A single app manifest that specifies the app title, id, and version.
- Access to platform-specific APIs and tools when required.
- A single cross-platform app entry point.

.NET MAUI single project is enabled using multi-targeting and the use of SDK-style projects in .NET 6. For more information about .NET MAUI single project, see [.NET MAUI single project](#).

Hot reload

.NET MAUI includes support for .NET hot reload, which enables you to modify your managed source code while the app is running, without the need to manually pause or hit a breakpoint. Then, your code edits can be applied

to your running app without recompilation.

.NET MAUI includes support for XAML hot reload, which enables you to save your XAML files and see the changes reflected in your running app without recompilation. 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.

Supported platforms for .NET MAUI apps

9/20/2022 • 2 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) apps can be written for the following platforms:

- Android 5.0 (API 21) or higher.
- iOS 10 or higher.
- macOS 10.15 or higher, using Mac Catalyst.
- Windows 11 and Windows 10 version 1809 or higher, using [Windows UI Library \(WinUI\) 3](#).

.NET MAUI Blazor apps have the following additional platform requirements:

- Android 7.0 (API 24) or higher is required
- iOS 14 or higher is required.
- macOS 11 or higher, using Mac Catalyst.

.NET MAUI Blazor apps also require an updated platform specific WebView control. For more information, see [Blazor supported platforms](#).

.NET MAUI apps for Android, iOS, and Windows can be built in Visual Studio. However, a networked Mac is required for iOS development.

Additional platform support

.NET MAUI also includes Tizen support, which is provided by Samsung.

Build your first app

9/20/2022 • 7 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to create and run your first .NET Multi-platform App UI (.NET MAUI) app in Visual Studio 2022 on Windows, or Visual Studio 2022 for Mac 17.4 Preview.

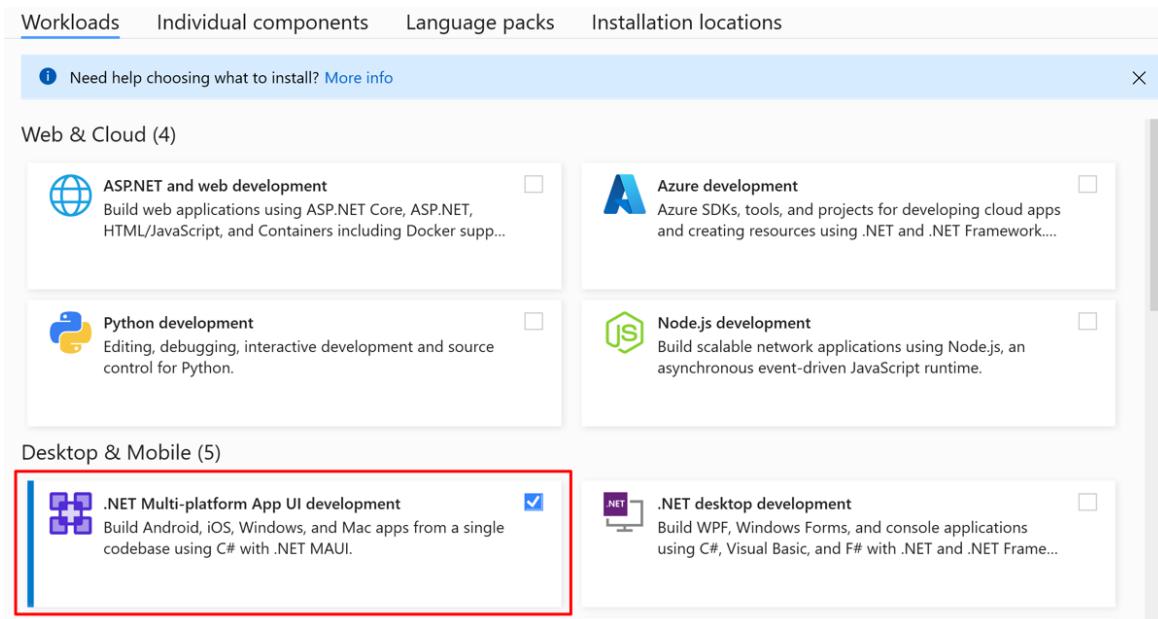
- [Visual Studio](#)
- [Visual Studio for Mac](#)

In this tutorial, you'll create your first .NET MAUI app in Visual Studio 2022, and run it on an Android emulator:

1. To create .NET MAUI apps, you'll need the latest Visual Studio 2022:

- [Download 2022 Community](#)
- [Download 2022 Professional](#)
- [Download 2022 Enterprise](#)

Either install Visual Studio, or modify your installation, and install the .NET Multi-platform App UI development workload with its default optional installation options:



Developing .NET MAUI apps for iOS on Windows requires a Mac build host. If you don't specifically need to target iOS and don't have a Mac, consider getting started with Android or Windows instead.

In this tutorial, you'll create your first .NET MAUI app in Visual Studio, and run it on an iOS simulator:

1. To create .NET MAUI apps, you'll need the latest Visual Studio 2022:

- [Download 2022 Community](#)
- [Download 2022 Professional](#)
- [Download 2022 Enterprise](#)

Either install Visual Studio, or modify your installation, and install the .NET Multi-platform App UI development workload with its default optional installation options:

Workloads Individual components Language packs Installation locations

Need help choosing what to install? [More info](#)

Web & Cloud (4)

-  **ASP.NET and web development**
Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker supp...
-  **Python development**
Editing, debugging, interactive development and source control for Python.
-  **Azure development**
Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET and .NET Framework....
-  **Node.js development**
Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Desktop & Mobile (5)

-  **.NET Multi-platform App UI development**
Build Android, iOS, Windows, and Mac apps from a single codebase using C# with .NET MAUI.
-  **.NET desktop development**
Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...

In this tutorial, you'll create your first .NET MAUI app in Visual Studio 2022, and run it on Windows:

1. To create .NET MAUI apps, you'll need the latest Visual Studio 2022:

- [Download 2022 Community](#)
- [Download 2022 Professional](#)
- [Download 2022 Enterprise](#)

Either install Visual Studio, or modify your installation, and install the .NET Multi-platform App UI development workload with its default optional installation options:

Workloads Individual components Language packs Installation locations

Need help choosing what to install? [More info](#)

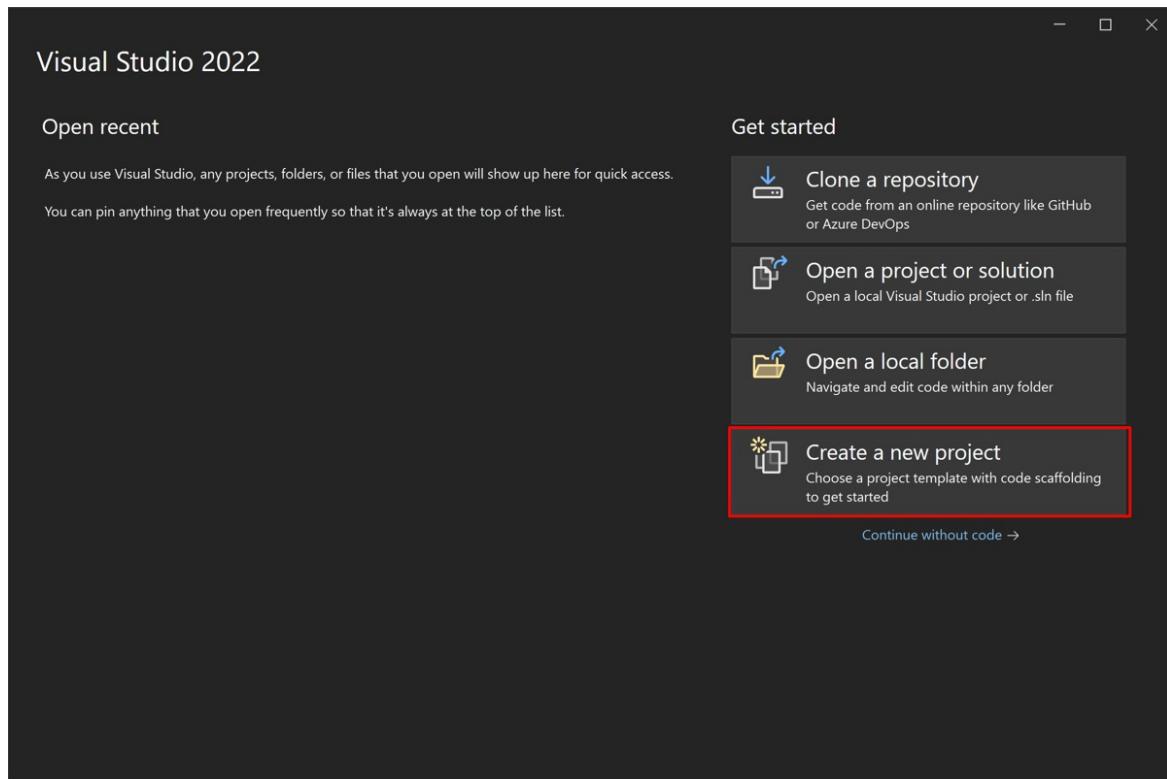
Web & Cloud (4)

-  **ASP.NET and web development**
Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker supp...
-  **Python development**
Editing, debugging, interactive development and source control for Python.
-  **Azure development**
Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET and .NET Framework....
-  **Node.js development**
Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

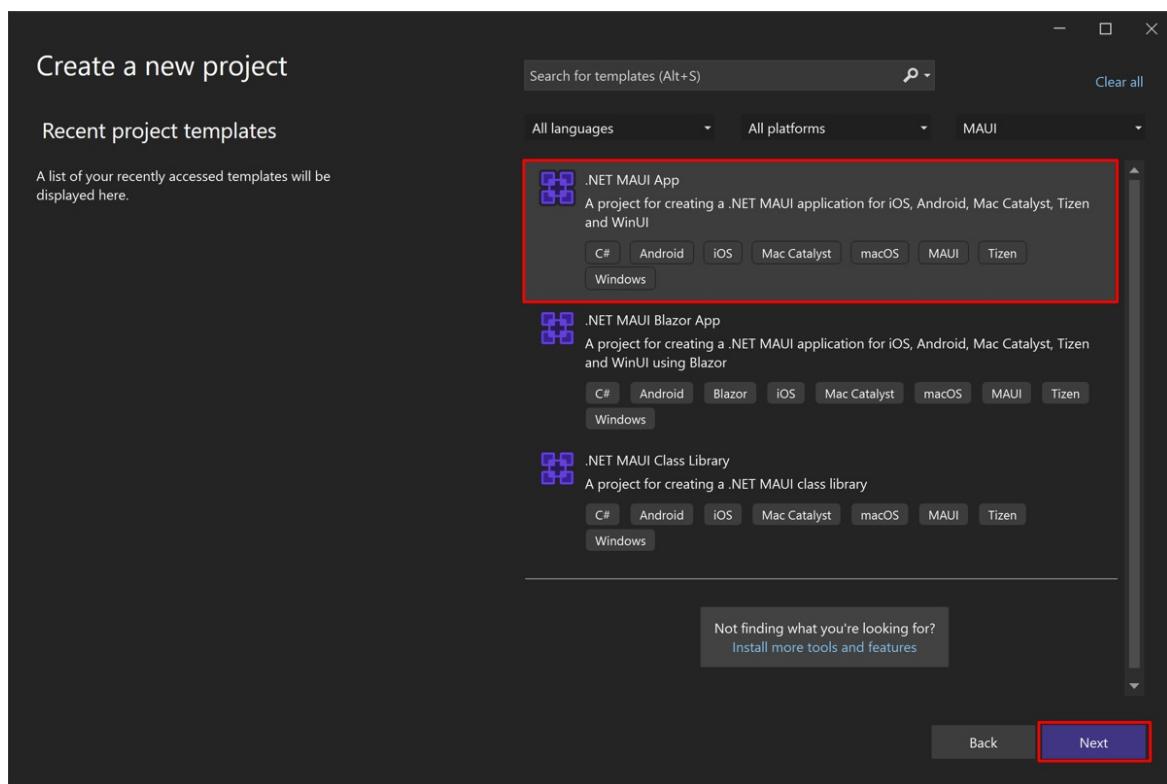
Desktop & Mobile (5)

-  **.NET Multi-platform App UI development**
Build Android, iOS, Windows, and Mac apps from a single codebase using C# with .NET MAUI.
-  **.NET desktop development**
Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET and .NET Frame...

2. Launch Visual Studio 2022, and in the start window click **Create a new project** to create a new project:



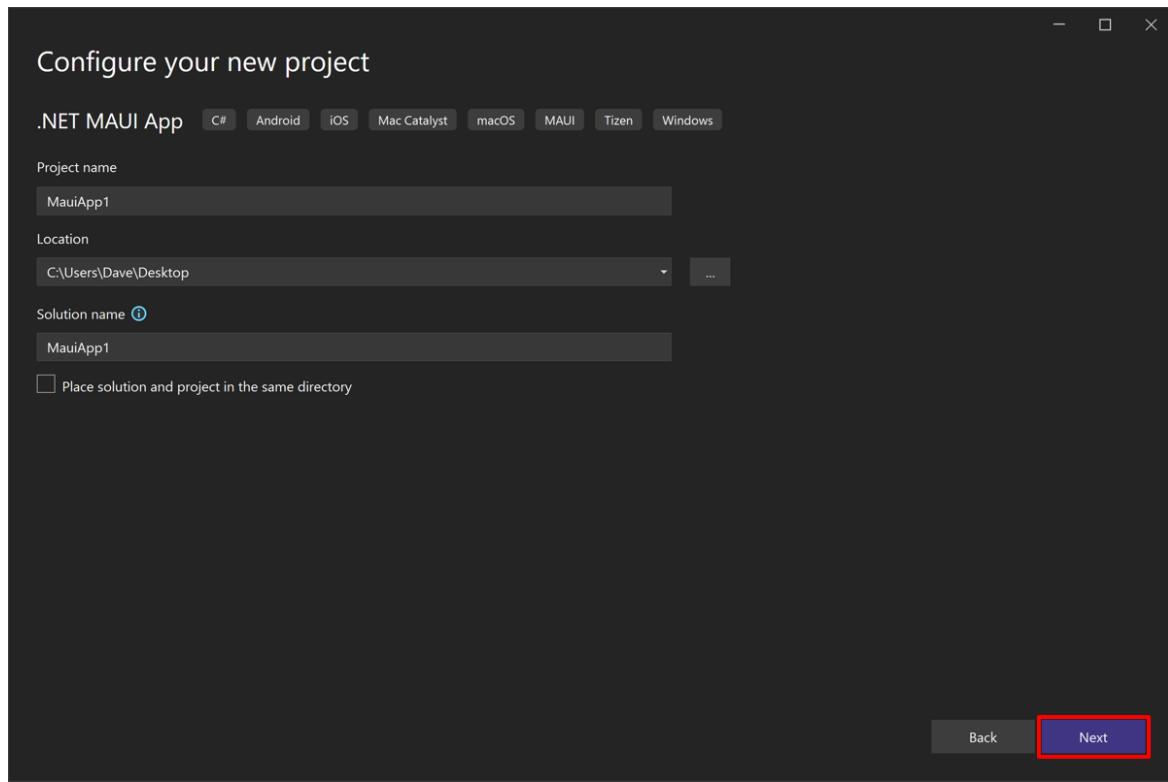
3. In the **Create a new project** window, select **MAUI** in the **All project types** drop-down, select the **.NET MAUI App** template, and click the **Next** button:



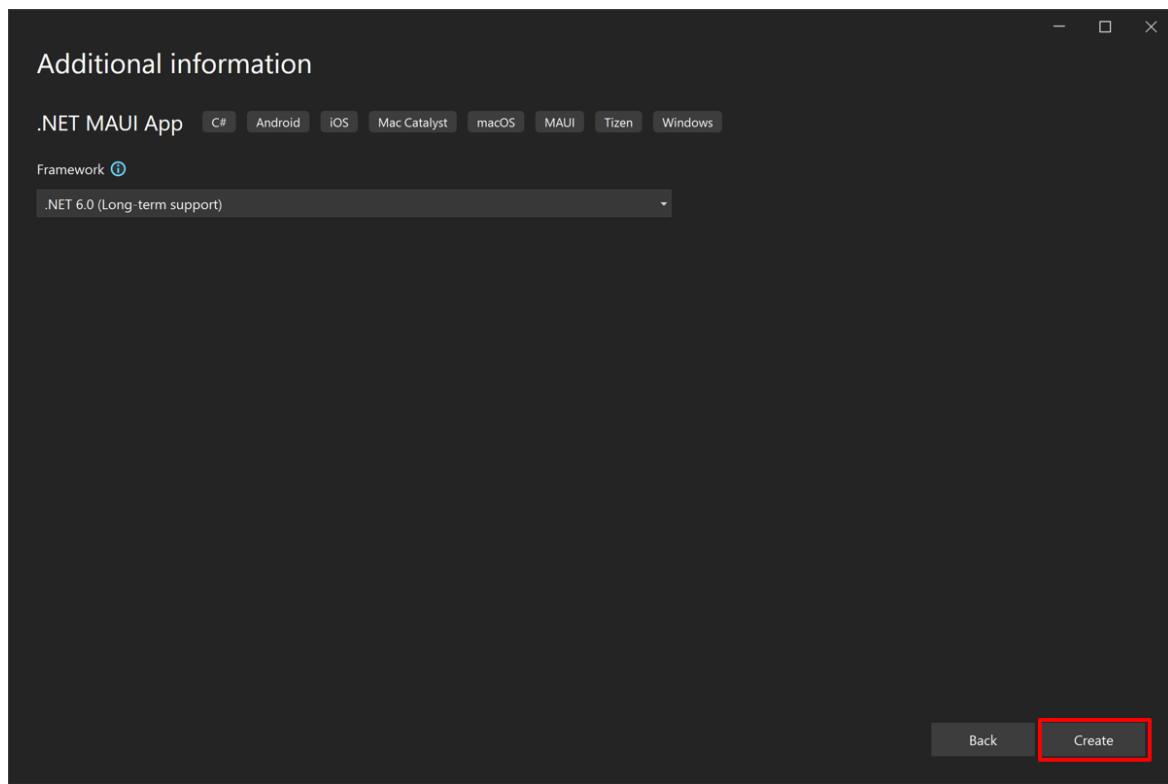
NOTE

The .NET MAUI templates might not appear in Visual Studio if you also have .NET 7 Preview installed. For more information, see [.NET MAUI templates do not appear in Visual Studio](#).

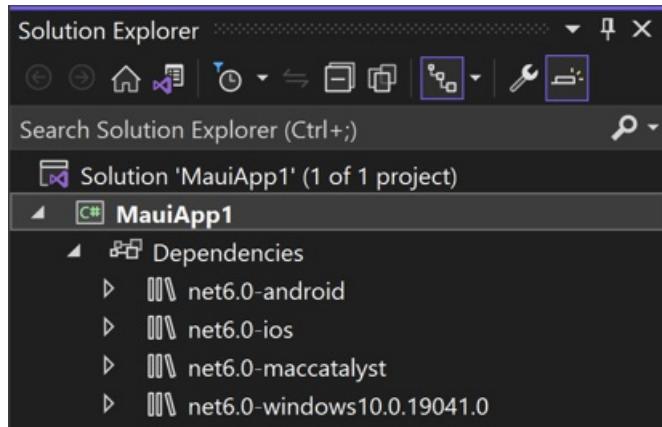
4. In the **Configure your new project** window, name your project, choose a suitable location for it, and click the **Next** button:



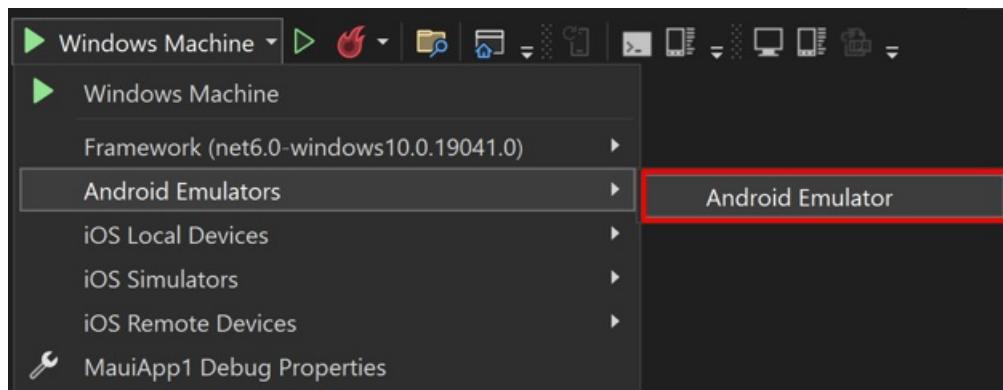
5. In the Additional information window, click the **Create** button:



6. Wait for the project to be created, and its dependencies to be restored:



7. In the Visual Studio toolbar, use the **Debug Target** drop down to select **Android Emulators** and then the **Android Emulator** entry:

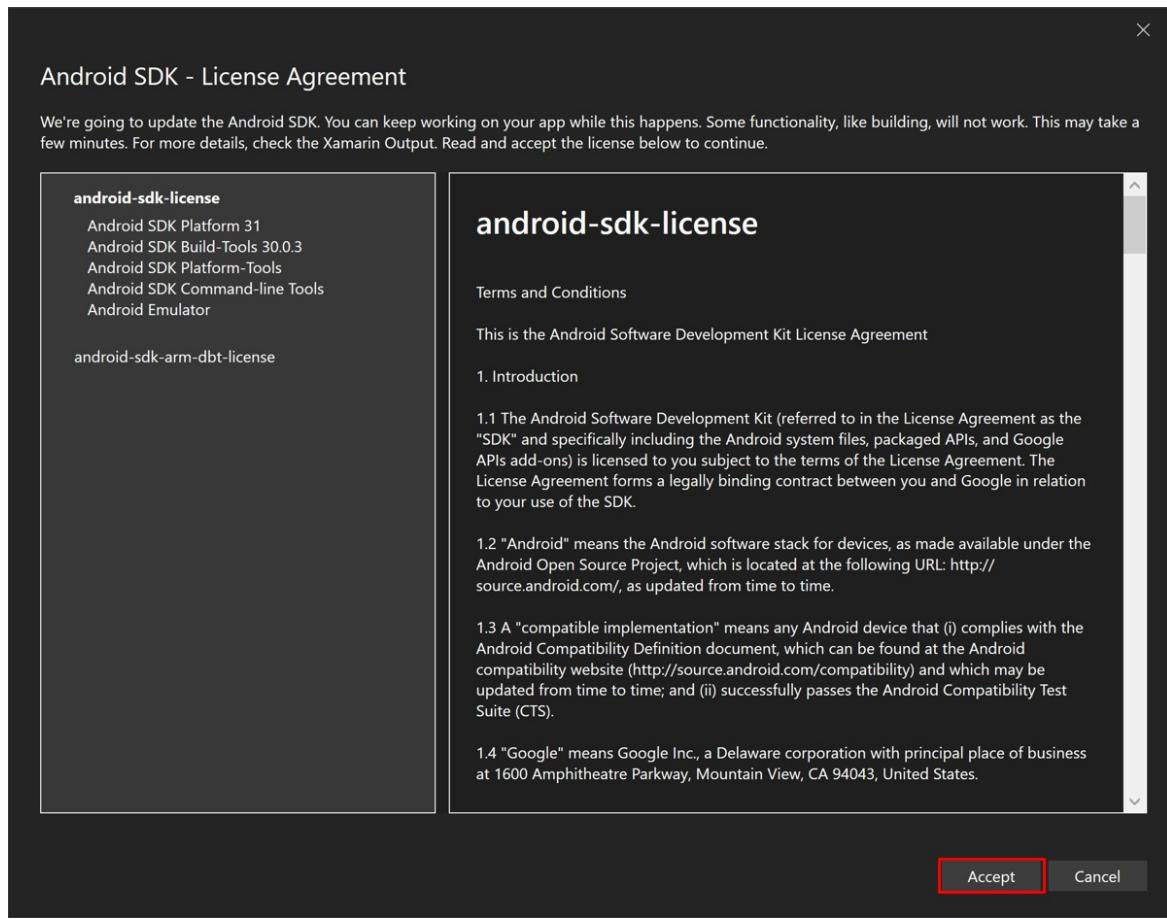


8. In the Visual Studio toolbar, press the **Android Emulator** button:

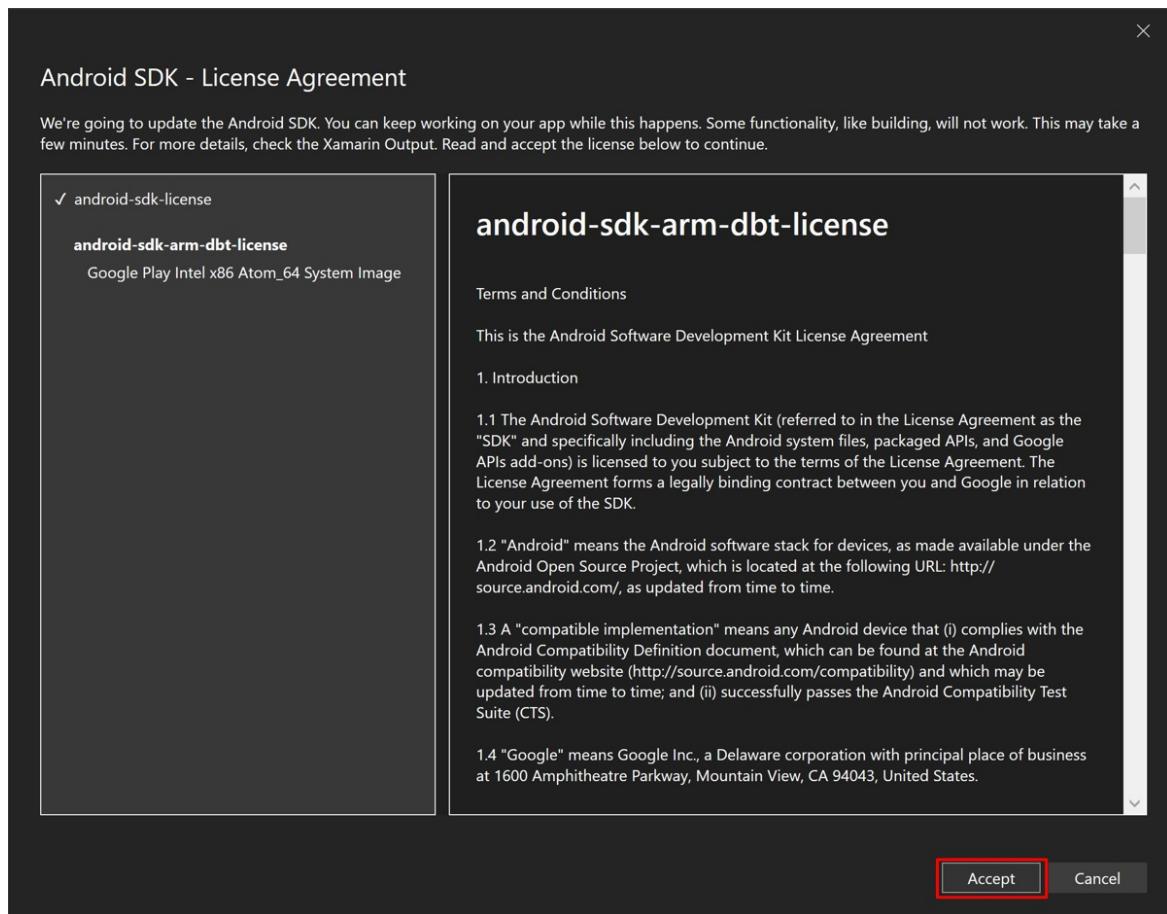


Visual Studio will start to install the default Android SDK and Android Emulator.

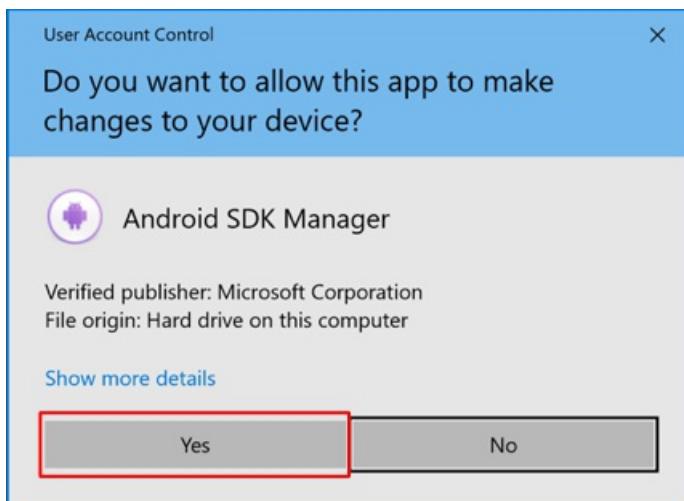
9. In the **Android SDK - License Agreement** window, press the **Accept** button:



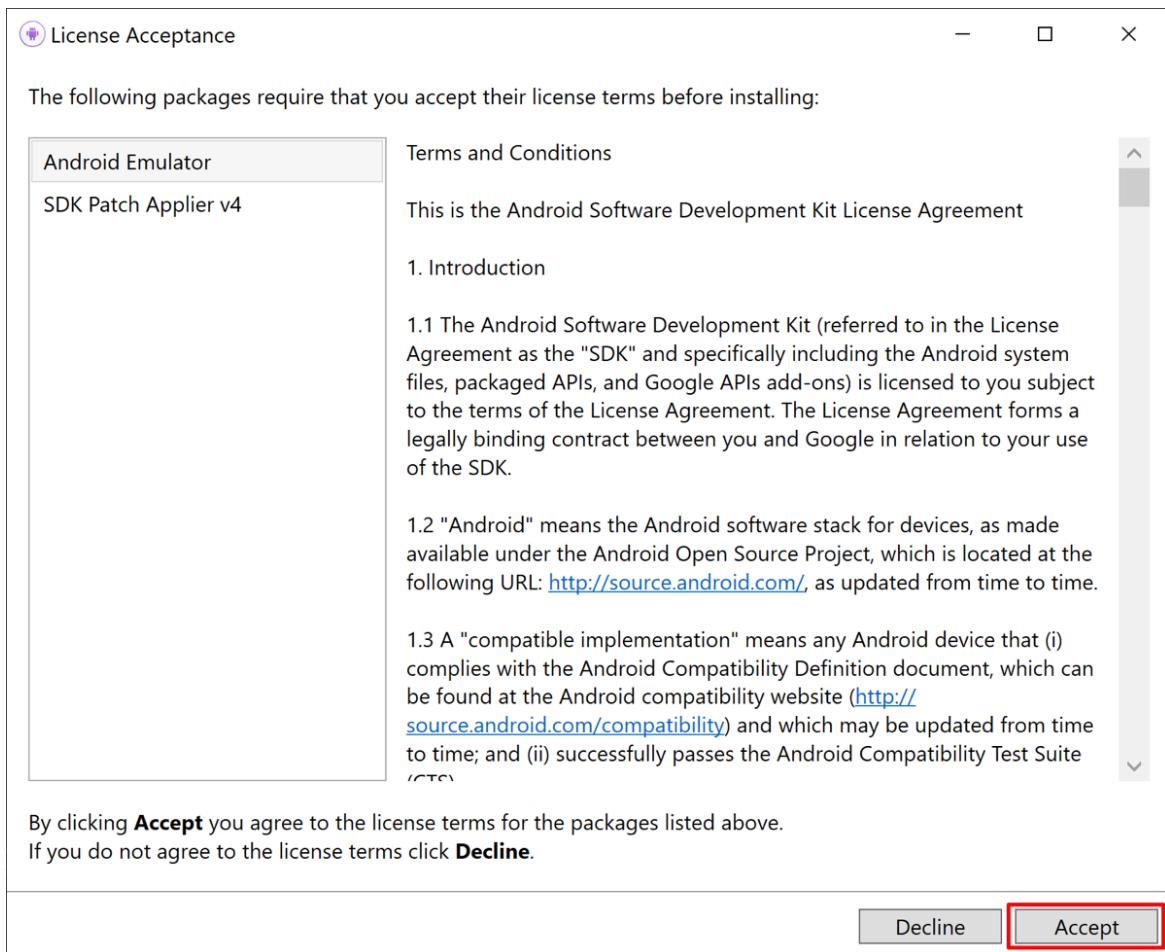
10. In the **Android SDK - License Agreement** window, press the **Accept** button:



11. In the User Account Control dialog, press the **Yes** button:

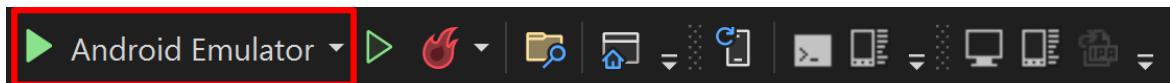


12. In the License Acceptance window, press the **Accept** button:



Wait for Visual Studio to download the Android SDK and Android Emulator.

13. In the Visual Studio toolbar, press the **Android Emulator** button:

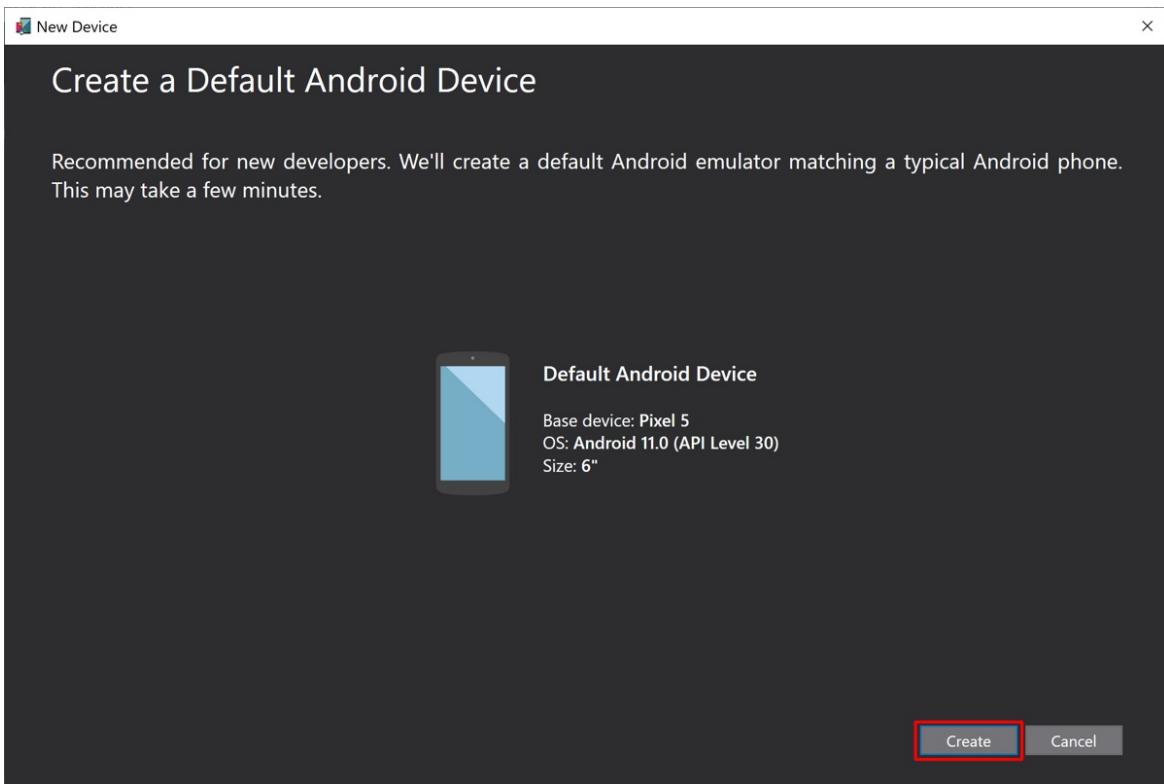


Visual Studio will start to create a default Android emulator.

14. In the User Account Control dialog, press the **Yes** button:

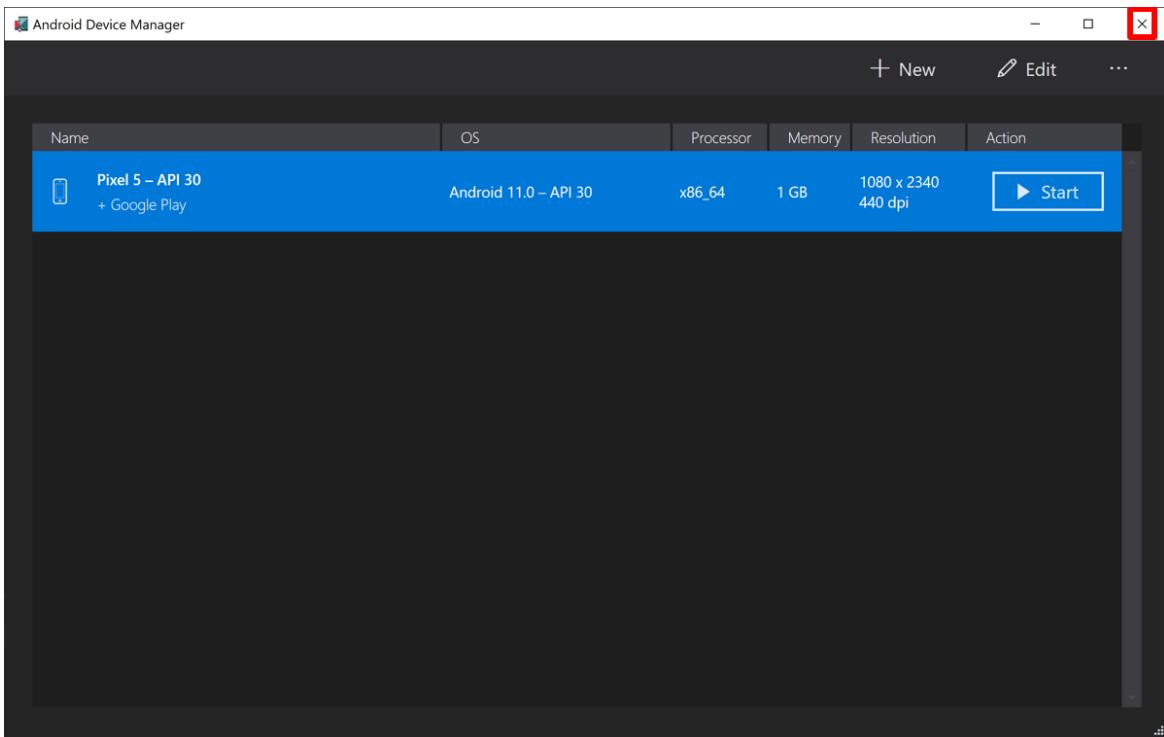


15. In the **New Device** window, press the **Create** button:



Wait for Visual Studio to download, unzip, and create an Android emulator.

16. Close the **Android Device Manager** window:



17. In the Visual Studio toolbar, press the **Pixel 5 - API 30 (Android 11.0 - API 30)** button to build and run the app:



Visual Studio will start the Android emulator, build the app, and deploy the app to the emulator.

WARNING

Hardware acceleration must be enabled to maximize Android emulator performance. Failure to do this will result in the emulator running very slowly. For more information, see [How to enable hardware acceleration with Android emulators \(Hyper-V & HAXM\)](#).

18. In the running app in the Android emulator, press the **Click me** button several times and observe that the count of the number of button clicks is incremented.



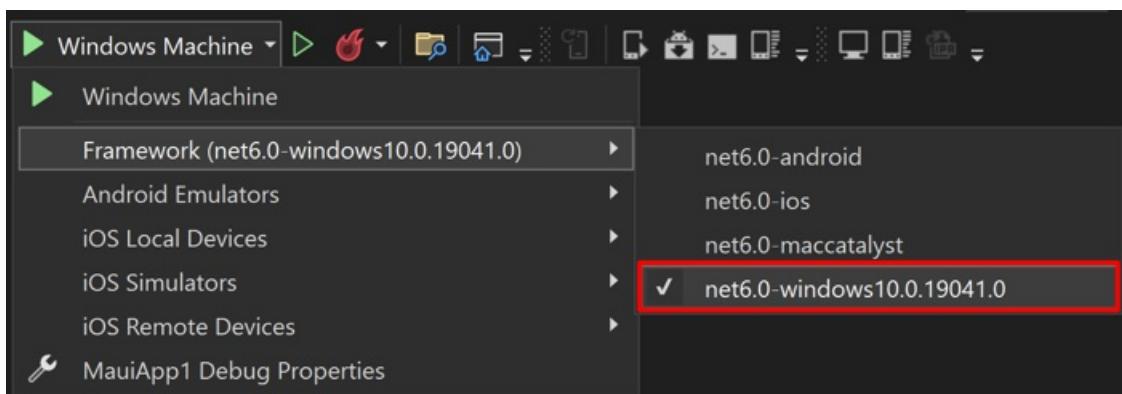
Hello, World!

Welcome to .NET Multi-platform App UI

Clicked 5 times



7. In the Visual Studio toolbar, use the **Debug Target** drop down to select **Framework** and then the **net6.0-windows** entry:



8. In the Visual Studio toolbar, press the **Windows Machine** button to build and run the app:



If you've not enabled Developer Mode, Visual Studio will prompt you to enable it. In the **Enable Developer Mode for Windows** dialog, click **settings for developers** to open the Settings app:

Enable Developer Mode for Windows

X

This device needs to be set up correctly to develop this type of app for Windows. If you don't, then you can't install and test your app before you submit it to the Windows Store.

Go to [settings for developers](#) on your device, and select Developer Mode.

This device is not currently in developer mode.

[Close](#)

In the Settings app, turn on **Developer Mode** and accept the disclaimer:

Developer Mode

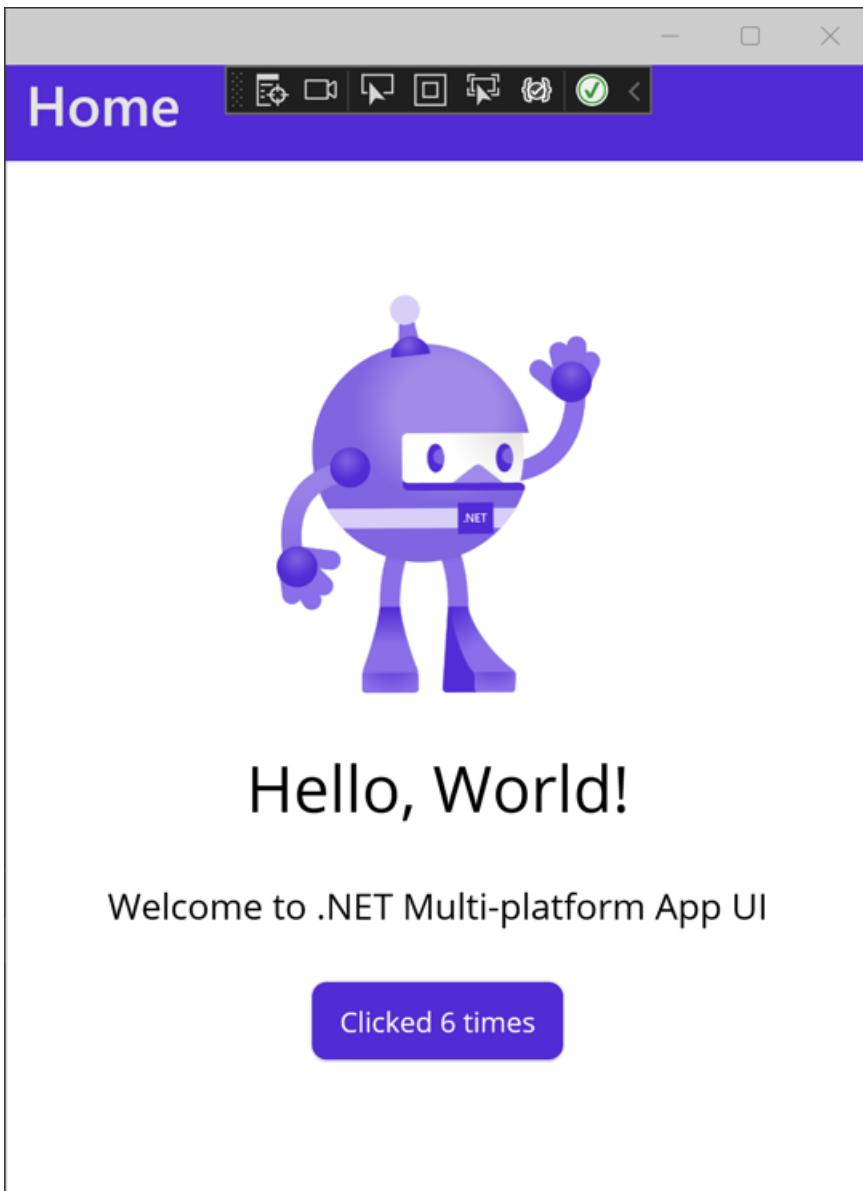
Install apps from any source, including loose files.



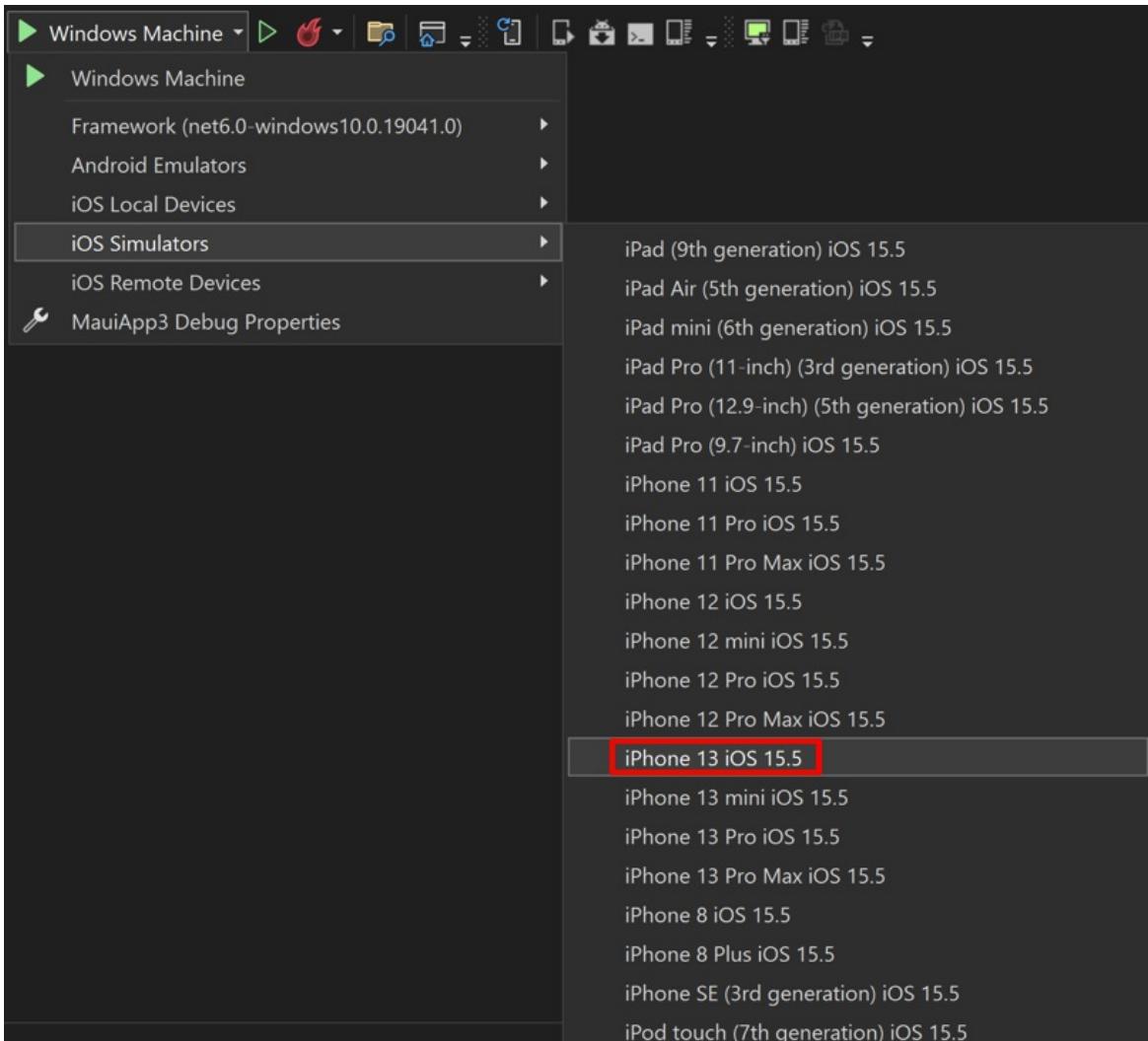
On

Close the Settings app, and then close the **Enable Developer Mode for Windows** dialog.

9. In the running app, press the **Click me** button several times and observe that the count of the number of button clicks is incremented:



7. In Visual Studio, pair the IDE to a Mac Build host. For more information, see [Pair to Mac for iOS development](#).
8. In the Visual Studio toolbar, use the **Debug Target** drop down to select **iOS Simulators** and then a specific iOS simulator:

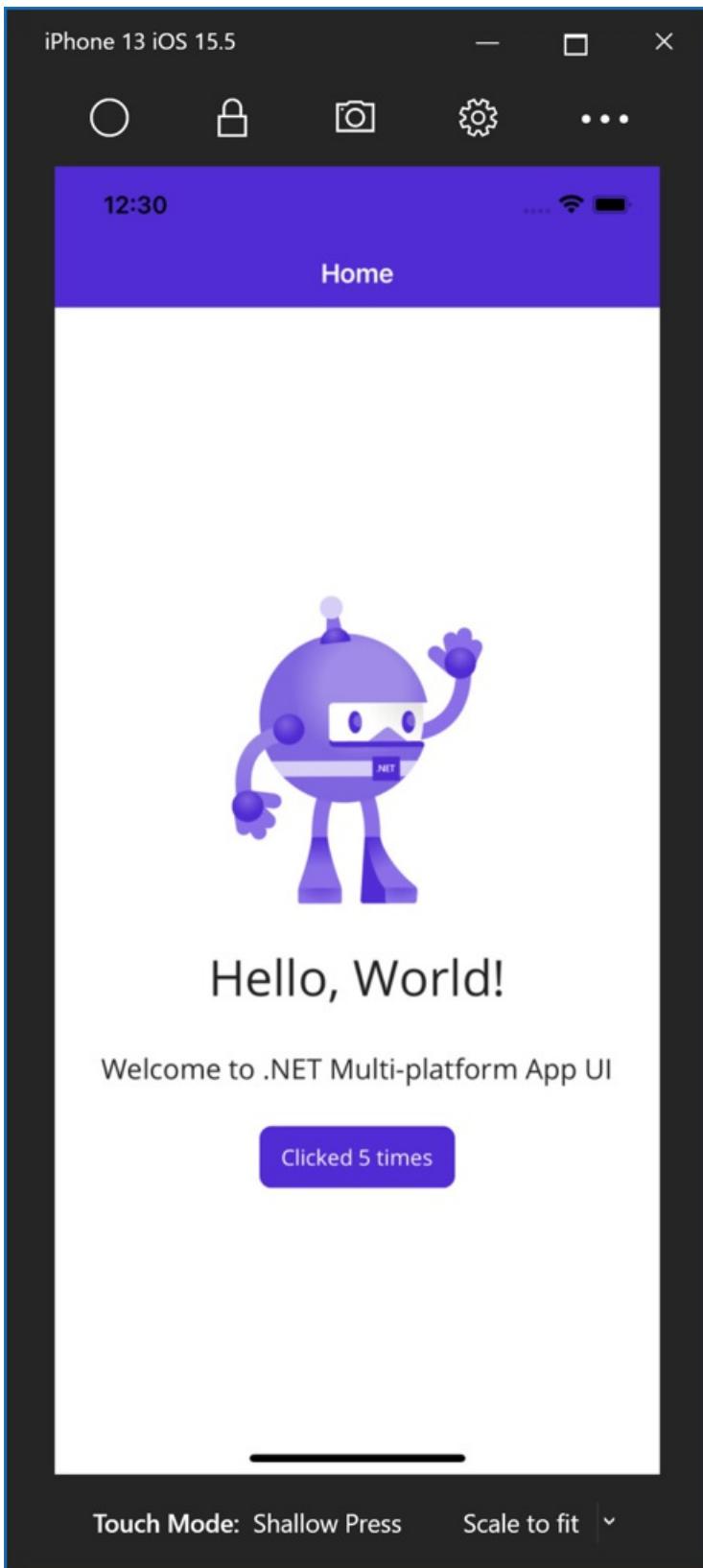


9. In the Visual Studio toolbar, press the Start button for your chosen iOS simulator to build and run your app:



Visual Studio will build the app, start the remote iOS Simulator for Windows, and deploy the app to the remote simulator. For more information about the remote iOS Simulator for Windows, see [Remote iOS Simulator for Windows](#).

10. In the running app, press the **Click me** button several times and observe that the count of the number of button clicks is incremented.



.NET MAUI apps that target Mac Catalyst can only be launched and debugged using Visual Studio 2022 for Mac 17.4 Preview.

Next steps

In this tutorial, you've learnt how to create and run your first .NET Multi-platform App UI (.NET MAUI) app.

To learn the fundamentals of building an app with .NET MAUI, see [Create a .NET MAUI app](#).

Resources for learning .NET MAUI

9/20/2022 • 2 minutes to read • [Edit Online](#)

There are many different resources available for you to use to learn .NET Multi-platform App UI (.NET MAUI). There are Microsoft Learn modules, workshops, videos, and podcasts. Each varies in its depth and the topics it covers.

- [Build mobile and desktop apps with .NET MAUI](#)

Learn how to use .NET MAUI to build apps that run on mobile devices and the desktop using C# and Visual Studio. You'll learn the fundamentals of building an app with .NET MAUI and more advanced topics such as local data storage and invoking REST-based web services.

- [.NET MAUI for beginners](#)

Follow a short video series that teaches you how to get started with .NET MAUI and Visual Studio, to build your very first cross-platform desktop and mobile app.

- [.NET MAUI workshop](#)

Learn how to build a .NET MAUI app that displays a list of monkeys from around the world. You'll start by building the business logic backend that retrieves JSON-encoded data from a REST-based endpoint. You then use .NET MAUI to find the closest monkey to you, and show the monkey on a map. You'll also examine how to display data using different approaches, and then finally fully theme the app.

- [Enterprise application patterns using .NET MAUI](#)

This book provides real world solutions for addressing challenges faced when building an enterprise app using .NET MAUI. The book covers topics such as:

- Model-View-ViewModel (MVVM) pattern
- Dependency injection
- Navigation
- Configuration
- Loose-coupling of components
- Additional enterprise concerns

The content of this book is helpful for anyone looking to build a new app or looking to solve the problems of apps that evolve over time.

- [.NET MAUI samples](#)

Download and explore the code of different example .NET MAUI apps.

- [.NET MAUI podcast](#)

Keep up with the latest news in the world of mobile and desktop development with the official .NET MAUI podcast.

Migrate your app from Xamarin.Forms

9/20/2022 • 2 minutes to read • [Edit Online](#)

You don't need to rewrite your Xamarin.Forms apps to move them to .NET Multi-platform App UI (.NET MAUI). However, you will need to make a small amount of code changes to each app. Similarly, you can use single-project features without merging all of your Xamarin.Forms projects into one project.

To migrate a Xamarin.Forms app to .NET 6 and update the code to .NET MAUI, you'll need to do the following:

- Convert the projects from .NET Framework to .NET SDK style.
- Update namespaces.
- Update any incompatible NuGet packages.
- Address any breaking API changes.
- Run the converted app and verify that it functions correctly.

For more information, see [Migrating from Xamarin.Forms](#) on the .NET MAUI Wiki.

How to enable hardware acceleration with Android emulators (Hyper-V & HAXM)

9/20/2022 • 4 minutes to read • [Edit Online](#)

This article explains how to use your computer's hardware acceleration features to maximize Android emulator performance.

With Visual Studio, you can easily test and debug your .NET MAUI app for Android in situations where an Android device isn't available. However, if hardware acceleration isn't available or enabled, the emulator will run too slow. You can drastically improve the performance of the emulator by enabling hardware acceleration and using **x86-64** or **x86** virtual device images.

SCENARIO	HAXM	WHPX	HYPERVISOR.FRAMEWORK
You have an Intel Processor	X	X	X
You have an AMD Processor		X	
You want to support Hyper-V		X	
You want to support nested Virtualization		Limited	
You want to use technologies like Docker	(with WSL2)	X	X

Accelerating Android emulators on Windows

The following virtualization technologies are available for accelerating the Android emulator:

1. **Microsoft's Hyper-V and the Windows Hypervisor Platform (WHPX).**

[Hyper-V](#) is a virtualization feature of Windows that makes it possible to run virtualized computer systems on a physical host computer.

2. **Intel's Hardware Accelerated Execution Manager (HAXM).**

HAXM is a virtualization engine for computers running Intel CPUs.

For the best experience on Windows, it's recommended you use WHPX to accelerate the Android emulator. If WHPX isn't available on your computer, then HAXM can be used. The Android emulator automatically uses hardware acceleration if the following criteria are met:

- Hardware acceleration is available and enabled on your development computer.
- The emulator is running a system image created for an **x86-64** or **x86**-based virtual device.

IMPORTANT

You can't run a VM-accelerated emulator inside another VM, such as a VM hosted by VirtualBox, VMware, or Docker (unless using WSL2). You must run the Android emulator [directly on your system hardware](#).

For information about launching and debugging with the Android emulator, see [Debugging on the Android Emulator](#).

Accelerating with Hyper-V

Before enabling Hyper-V, read the following section to verify that your computer supports Hyper-V.

Verifying support for Hyper-V

Hyper-V runs on the Windows Hypervisor Platform. To use the Android emulator with Hyper-V, your computer must meet the following criteria to support the Windows Hypervisor Platform:

- Your computer hardware must meet the following requirements:
 - A 64-bit Intel or AMD Ryzen CPU with Second Level Address Translation (SLAT).
 - CPU support for VM Monitor Mode Extension (VT-c on Intel CPUs).
 - Minimum of 4-GB memory.
- In your computer's BIOS, the following items must be enabled:
 - Virtualization Technology (may have a different label depending on motherboard manufacturer).
 - Hardware Enforced Data Execution Prevention.
- Your computer must be running Windows 11 or Windows 10 Version 1909 or later.

To verify that your computer hardware and software is compatible with Hyper-V, open a command prompt and type the following command:

```
systeminfo
```

If all listed Hyper-V requirements have a value of Yes, then your computer can support Hyper-V. For example:

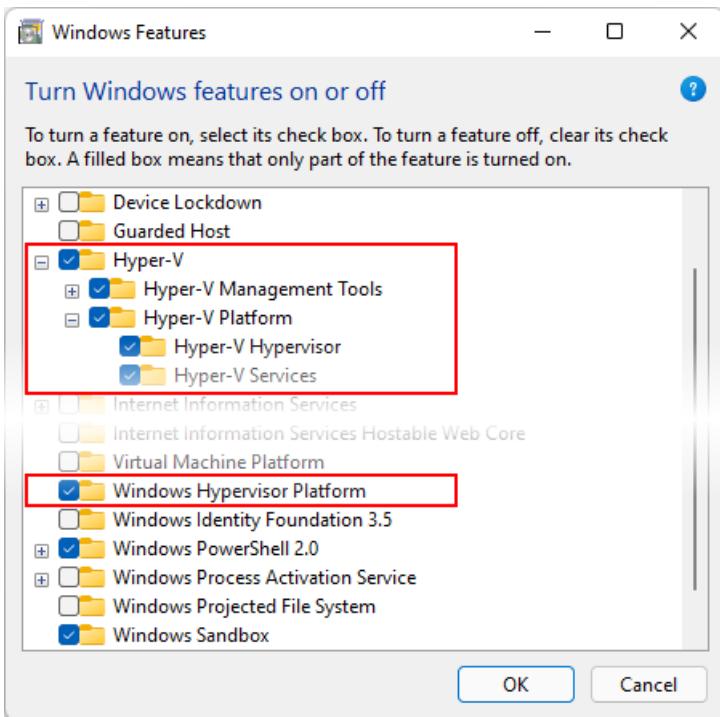
Hyper-V Requirements:	VM Monitor Mode Extensions: Yes
	Virtualization Enabled In Firmware: Yes
	Second Level Address Translation: Yes
	Data Execution Prevention Available: Yes

If the Hyper-V result indicates that a hypervisor is currently running, Hyper-V is already enabled.

Enabling Hyper-V acceleration in Windows and the emulator

If your computer meets the above criteria, use the following steps to accelerate the Android emulator with Hyper-V:

1. Enter **windows features** in the Windows search box and select **Turn Windows features on or off** in the search results. In the **Windows Features** dialog, enable both **Hyper-V** and **Windows Hypervisor Platform**:



After making these changes, reboot your computer.

IMPORTANT

On Windows 10 October 2018 Update (RS5) and higher, you only need to enable Hyper-V, as it will use Windows Hypervisor Platform (WHPX) automatically.

1. Make sure that the virtual device you [created in the Android Device Manager](#) is an **x86-64** or **x86**-based system image. If you use an Arm-based system image, the virtual device won't be accelerated and will run slowly.

After Hyper-V is enabled, you'll be able to run your accelerated Android emulator.

Accelerating with HAXM

IMPORTANT

HAXM is only supported on Intel CPUs.

If your computer doesn't support Hyper-V, you may use HAXM to accelerate the Android emulator. To use HAXM, [disable Device Guard](#).

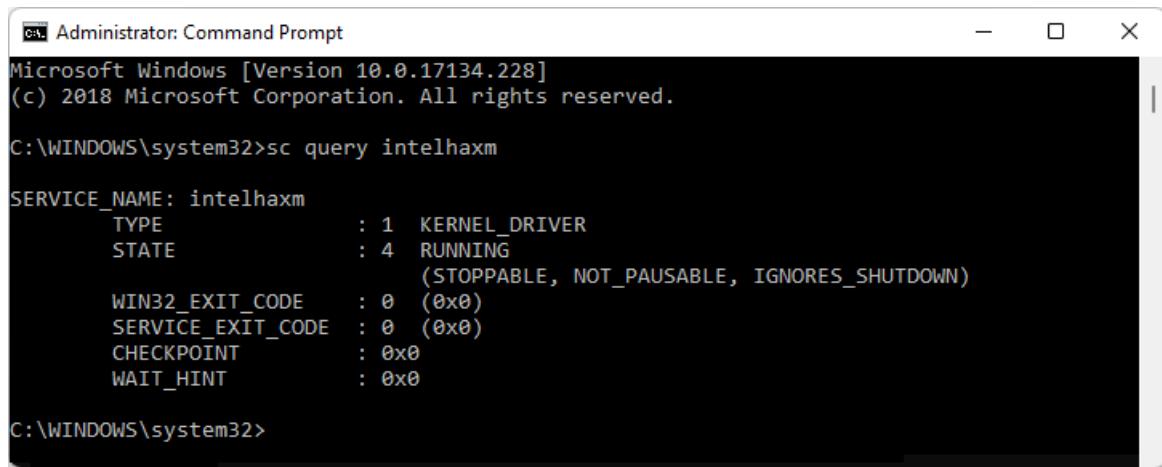
Verifying HAXM support

To determine if your hardware supports HAXM, follow the steps in [Does My Processor Support Intel Virtualization Technology?](#). If your hardware supports HAXM, you can check to see if HAXM is already installed by using the following steps:

1. Open a command prompt window and enter the following command:

```
sc query intelhaxm
```

2. Examine the output to see if the HAXM process is running. If it is, you should see output listing the **intelhaxm** state as **RUNNING**. For example:



```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.17134.228]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>sc query intelhaxm

SERVICE_NAME: intelhaxm
    TYPE               : 1  KERNEL_DRIVER
    STATE              : 4  RUNNING
                           (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
    WIN32_EXIT_CODE    : 0  (0x0)
    SERVICE_EXIT_CODE : 0  (0x0)
    CHECKPOINT        : 0x0
    WAIT_HINT         : 0x0

C:\WINDOWS\system32>
```

If `STATE` isn't set to `RUNNING`, then HAXM isn't installed.

If your computer can support HAXM but HAXM isn't installed, use the steps in the next section to install HAXM.

Installing HAXM

HAXM install packages for Windows are available from the [Intel Hardware Accelerated Execution Manager GitHub releases page](#). Use the following steps to download and install HAXM:

1. From the Intel website, download the latest [HAXM virtualization engine](#) installer for Windows. The advantage of downloading the HAXM installer directly from the Intel website is that you can be assured of using the latest version.
2. Run `intelhaxm-android.exe` to start the HAXM installer. Accept the default values in the installer dialogs.

When you [create a virtual device](#), be sure to select an `x86_64` or `x86`-based system image. If you use an Arm-based system image, the virtual device will not be accelerated and will run slowly.

Troubleshooting

For help with troubleshooting hardware acceleration issues, see the Android emulator [Troubleshooting](#) guide.

Related Links

- [Run Apps on the Android Emulator](#)

Managing virtual devices with the Android Device Manager

9/20/2022 • 6 minutes to read • [Edit Online](#)

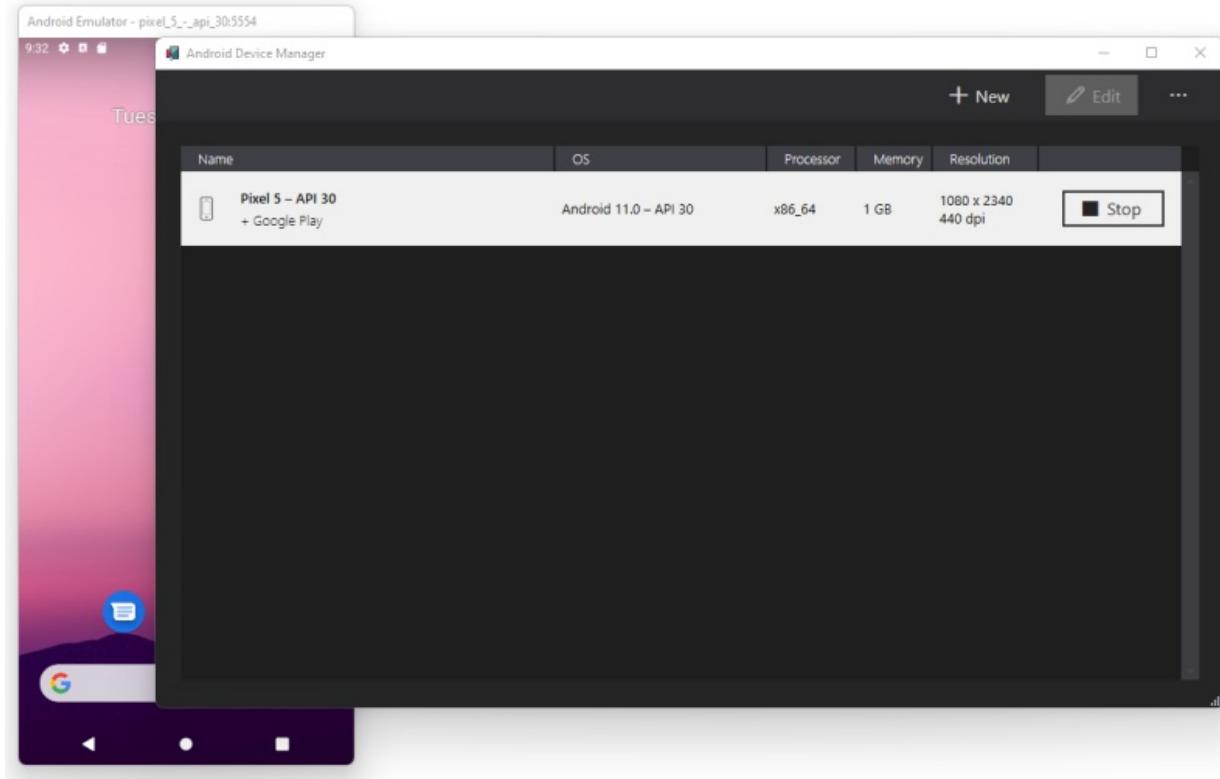
This article explains how to use the Android Device Manager to create and configure Android Virtual Devices (AVDs) that emulate physical Android devices. You can use these virtual devices to run and test your app without having to rely on a physical device.

IMPORTANT

Enable hardware acceleration for the Android devices. For more information, see [Hardware Acceleration for Emulator Performance](#).

Android Device Manager on Windows

You use the Android Device Manager to create and configure an Android Virtual Devices (AVD) that run in the [Android Emulator](#). Each AVD is an emulator configuration that simulates a physical Android device. This makes it possible to run and test your app in a variety of configurations that simulate different physical Android devices.



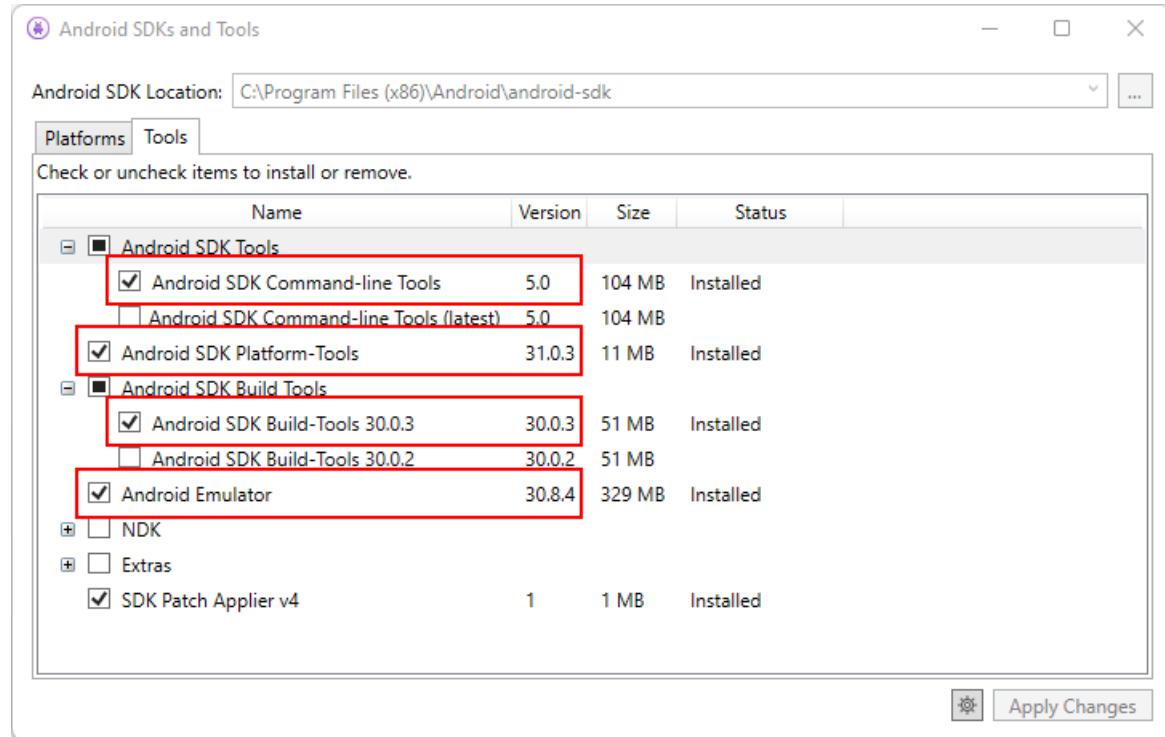
Requirements

To use the Android Device Manager, you'll need the following items:

- Visual Studio 2022: Community, Professional, and Enterprise editions are supported.
- The **Android SDK API Level 30** or later. Be sure to install the Android SDK at its default location if it isn't already installed: `C:\Program Files (x86)\Android\android-sdk`.

- The following packages must be installed:
 - Android SDK Tools 5.0 or later
 - Android SDK Platform-Tools 31.0.3 or later
 - Android SDK Build-Tools 30.0.2 or later
 - Android Emulator 30.8.4 or later

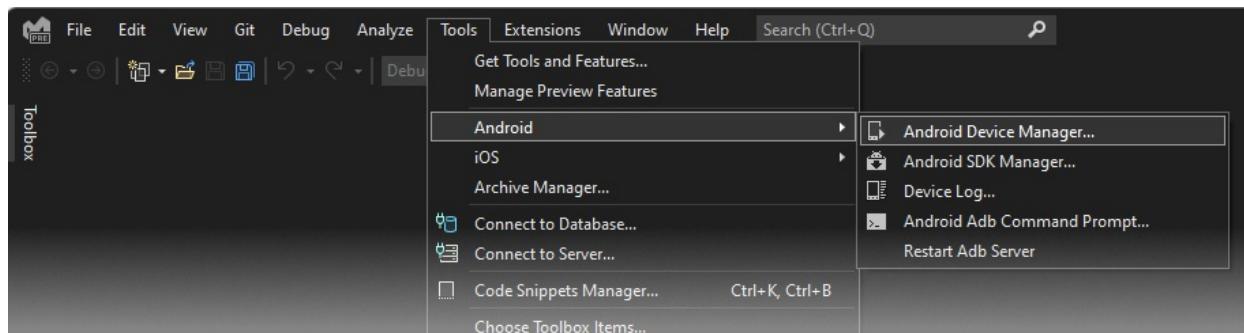
These packages should be displayed with **Installed** status as seen in the following screenshot:



When you install the .NET Multi-Platform App UI development workload in Visual Studio, everything is installed for you. For more information on setting up .NET MAUI with Visual Studio, see [Build your first app](#).

Open the device manager

Open the Android Device Manager in Visual Studio from the Tools menu by pressing Tools > Android > Android Device Manager:



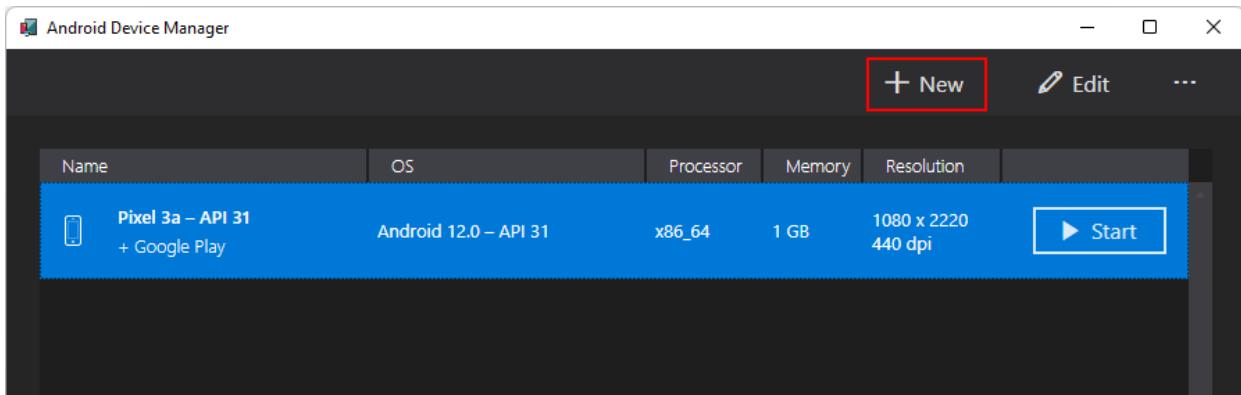
Main screen

When you run the Android Device Manager, it presents a screen that displays all currently configured virtual devices. For each virtual device, the **Name**, **OS** (Android Version), **Processor**, **Memory** size, and screen **Resolution** are displayed:

When you select a device in the list, the **Start** button appears on the right. Press the **Start** button to launch the emulator with this virtual device. If the emulator is running with the selected virtual device, the **Start** button changes to a **Stop** button that you can use to halt the emulator.

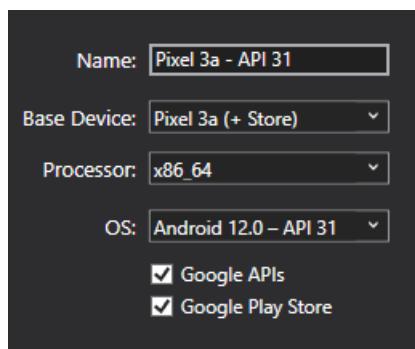
Create a new device

To create a new device, press the **New** button:



The **New Device** window is displayed. To configure the device, follow these steps:

1. Give the device a new name. In the following example, the new device is named *Pixel 3a - API 31*.



2. Select a physical device to emulate by selecting a device in the **Base Device** box.
3. Select a processor type for this virtual device with the **Processor** box.

It's recommended that you choose **x86_64** and enable [hardware acceleration](#).

4. Select the Android version (API level) with the **OS** box.

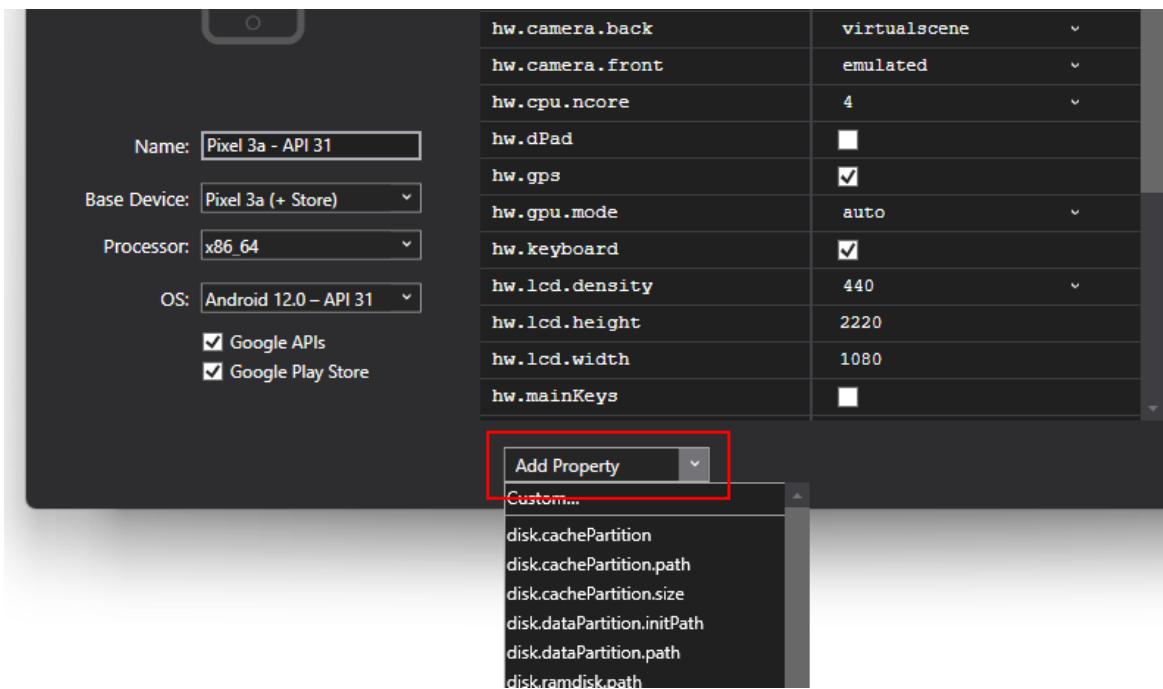
If you select an Android API level that has not yet been installed, the Device Manager will display **A new device will be downloaded** message at the bottom of the screen – it will download and install the necessary files as it creates the new virtual device.

5. If you want to include Google Play Services APIs in your virtual device, select the **Google APIs** option. To include the Google Play Store app on the virtual device, select the **Google Play Store** option

NOTE

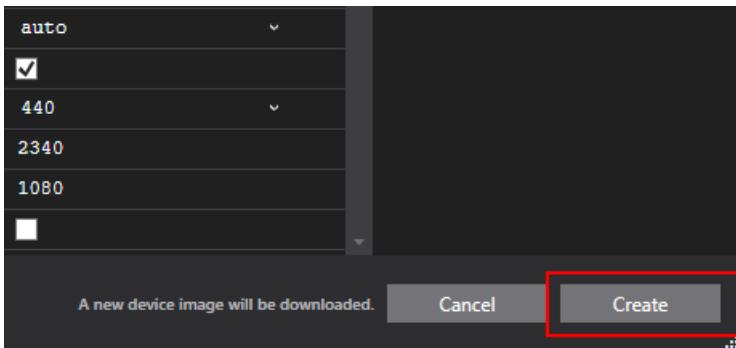
Google Play Store images are available only for some base device types such as Pixel, Pixel 2, Pixel 3, and Nexus 5. This is indicated by the text **(+ Store)** in the image name.

6. Use the property list to change some of the most commonly modified properties. To make changes to properties, see [Editing Android Virtual Device Properties](#).
7. Add any additional properties that you need to explicitly set with the **Add Property** box at the bottom of the window:



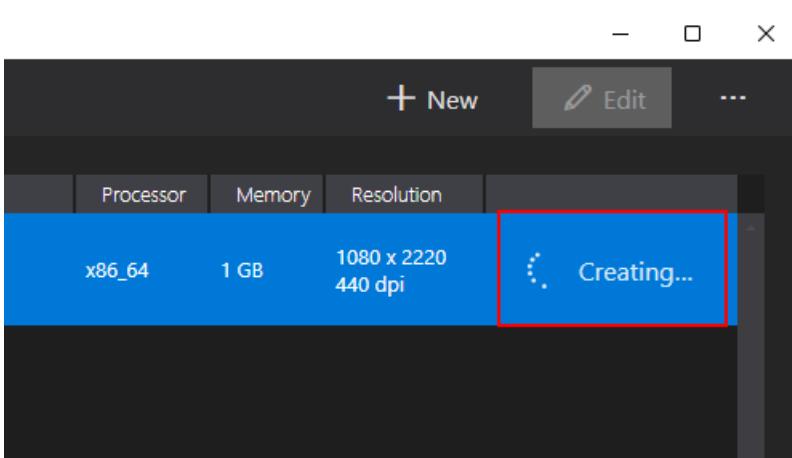
You can also define a custom property by selecting **Custom....**

8. Press the **Create** button to create the new device:



You might get a **License Acceptance** screen when you create the device. Select **Accept** if you agree to the license terms.

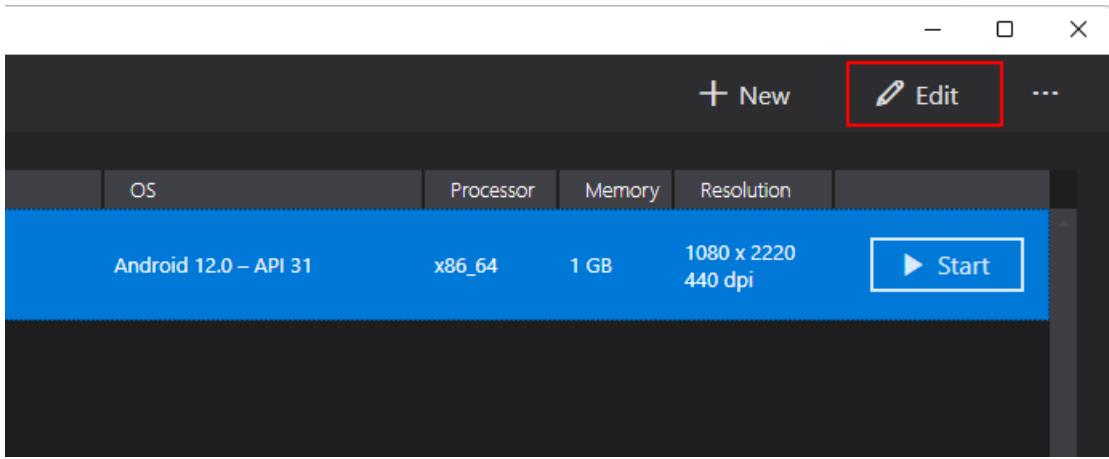
9. The Android Device Manager adds the new device to the list of installed virtual devices while displaying a **Creating** progress indicator during device creation:



10. When the creation process is complete, the new device is shown in the list of installed virtual devices with a **Start** button, ready to launch

Edit device

To edit an existing virtual device, select the device and then press the **Edit** button:



Pressing **Edit** displays the **Device Editor** window for the selected virtual device.

The **Device Editor** window lists the properties of the virtual device under the **Property** column, with the corresponding values of each property in the **Value** column. When you select a property, a detailed description of that property is displayed on the right.

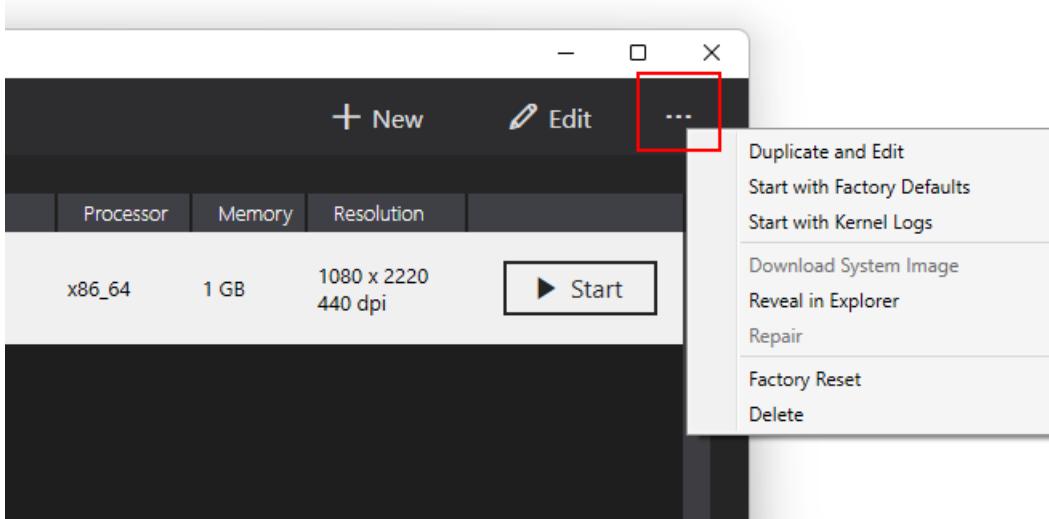
To change a property, edit its value in the **Value** column. For example, in the following screenshot the `hw.lcd.density` property is being changed to 240:

hw.keyboard	✓
hw.lcd.density	440
hw.lcd.height	120
hw.lcd.width	160
hw.mainKeys	240
hw.ramSize	213
hw.sdCard	320
hw.sensors.orientation	420
	480
	560

After you've made the necessary configuration changes, press the **Save** button. For more information about changing virtual device properties, see [Editing Android Virtual Device Properties](#).

Additional options

Additional options for working with devices are available from the **Additional Options (...)** pull-down menu:



The additional options menu contains the following items:

- **Duplicate and Edit** – Duplicates the currently selected device and opens it in the **New Device** screen with a new name that's similar to the existing device. For example, selecting Pixel 3a - API 31 and

pressing **Duplicate and Edit** appends a counter to the name: Pixel 3a - API 31 (1).

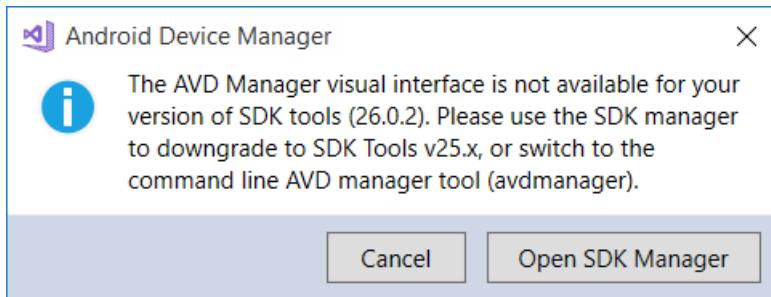
- **Start with Factory Defaults** – Starts the device with a cold boot.
- **Start with Kernel Logs** – Starts the emulator and opens up kernel logs directory.
- **Download System Image** – Downloads the Android OS system image for the device, if it's not already downloaded.
- **Reveal in Explorer** – Opens Windows Explorer and navigates to the folder that holds the files for the virtual device.
- **Repair** – Initiates a repair on the device.
- **Factory Reset** – Resets the selected device to its default settings, erasing any user changes made to the internal state of the device while it was running. This action also erases the current **Fast Boot** snapshot if it exists. This change doesn't alter modifications that you make to the virtual device during creation and editing. A dialog box will appear with the reminder that this reset cannot be undone – press **Factory Reset** to confirm the reset.
- **Delete** – Permanently deletes the selected virtual device. A dialog box will appear with the reminder that deleting a device cannot be undone. Press **Delete** if you are certain that you want to delete the device.

Troubleshooting

The following sections explain how to diagnose and work around problems that may occur when using the Android Device Manager to configure virtual devices.

Wrong version of Android SDK Tools

If you have the wrong Android SDK tools installed, installed, you may see this error dialog on launch:



If you see that error dialog, press **Open SDK Manager** to open the Android SDK Manager. In the Android SDK Manager, go to the **Tools** tab and install the following packages:

- **Android SDK Command-line Tools** 5.0 or later
- **Android SDK Platform-Tools** 31.0.3 or later
- **Android SDK Build-Tools** 30.0.3 or later

Snapshot disables Wi-Fi on Android Oreo

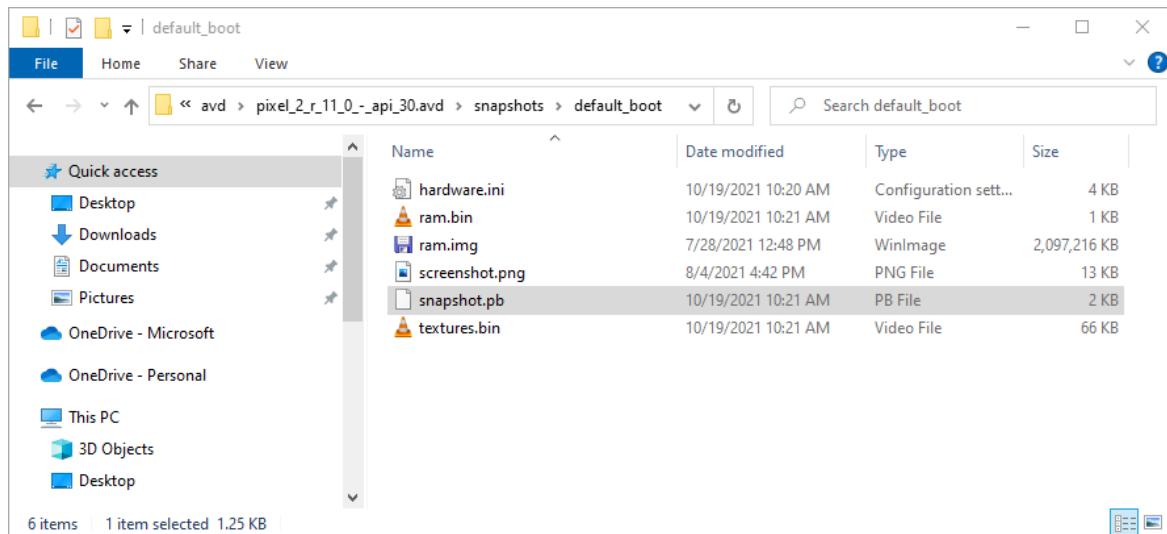
If you've an AVD configured for Android Oreo with simulated Wi-Fi access, restarting the AVD after a snapshot may cause Wi-Fi access to become disabled.

To work around this problem,

1. Open the **Android Device Manager**.
2. Select the AVD in the Android Device Manager.
3. From the **Additional Options (...)** menu, select **Reveal in Explorer**.

4. Navigate to the **snapshots > default_boot** folder.

5. Delete the *snapshot.pb* file:



6. Restart the AVD.

After these changes are made, the AVD will restart in a state that allows Wi-Fi to work again.

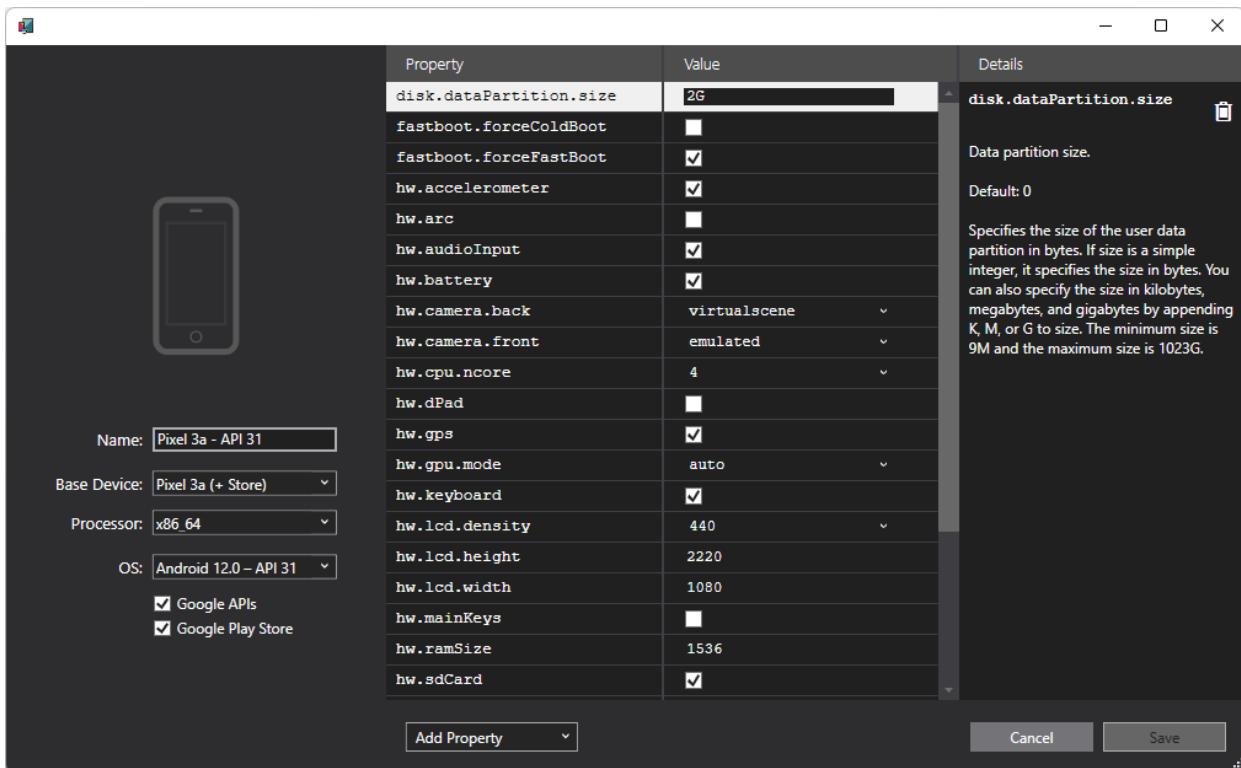
Editing Android virtual device properties

9/20/2022 • 14 minutes to read • [Edit Online](#)

This article explains how to use the Android Device Manager (AVD) to edit the profile properties of an Android virtual device.

Android Device Manager on Windows

The **Android Device Manager** supports the editing of individual Android virtual device profile properties. The **New Device** and **Device Edit** screens list the properties of the virtual device in the first column, with the corresponding values of each property in the second column (as seen in this example):



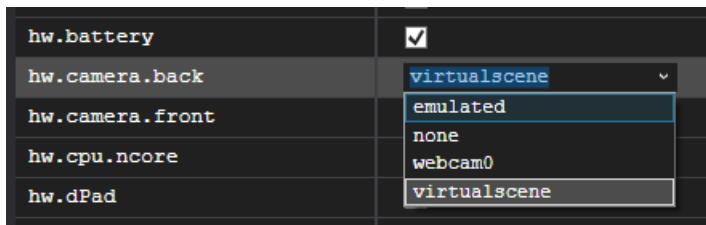
When you select a property, a detailed description of that property is displayed on the right. You can modify hardware profile properties and AVD properties. Hardware profile properties (such as `hw.ramSize` and `hw.accelerometer`) describe the physical characteristics of the emulated device. These characteristics include screen size, the amount of available RAM, whether or not an accelerometer is present. AVD properties specify the operation of the AVD when it runs. For example, AVD properties can be configured to specify how the AVD uses your development computer's graphics card for rendering.

You can change properties by using the following guidelines:

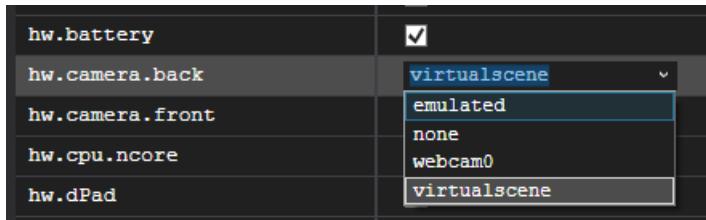
- To change a boolean property, click the check mark to the right of the boolean property:

fastboot.forceFastBoot	<input checked="" type="checkbox"/>
hw.accelerometer	<input checked="" type="checkbox"/>
hw.arc	<input type="checkbox"/>

- To change an *enum* (enumerated) property, click the down-arrow to the right of the property and choose a new value.



- To change a string or integer property, double-click the current string or integer setting in the value column and enter a new value.



The following table provides a detailed explanation of the properties listed in the **New Device** and **Device Editor** screens:

PROPERTY	DESCRIPTION	OPTIONS
<code>abi.type</code>	ABI type – Specifies the ABI (application binary interface) type of the emulated device. The x86 option is for the instruction set commonly referred to as "x86" or "IA-32." The x86_64 option is for the 64-bit x86 instruction set. The armeabi-v7a option is for the ARM instruction set with v7-a ARM extensions. The arm64-v8a option is for the ARM instruction set that supports AArch64.	x86, x86_64, armeabi-v7a, arm64-v8a
<code>disk.cachePartition</code>	Cache partition – Determines whether the emulated device will use a <code>/cache</code> partition on the device. The <code>/cache</code> partition (which is initially empty) is the location where Android stores frequently accessed data and app components. If set to no , the emulator will not use a <code>/cache</code> partition and the other <code>disk.cache</code> settings will be ignored.	yes, no
<code>disk.cachePartition.path</code>	Cache partition path – Specifies a cache partition image file on your development computer. The emulator will use this file for the <code>/cache</code> partition. Enter an absolute path or a path relative to the emulator's data directory. If not set, the emulator creates an empty temporary file called <code>cache.img</code> on your development computer. If the file does not exist, it is created as an empty file. This option is ignored if <code>disk.cachePartition</code> is set to no .	

PROPERTY	DESCRIPTION	OPTIONS
<code>disk.cachePartition.size</code>	<p>Cache partition size – The size of the cache partition file (in bytes). Normally you do not need to set this option unless the app will be downloading very large files that are larger than the default cache size of 66 megabytes. This option is ignored if <code>disk.cachePartition</code> is set to <code>no</code>. If this value is an integer, it specifies the size in bytes. You can also specify the size in kilobytes, megabytes, and gigabytes by appending <code>K</code>, <code>M</code>, or <code>G</code> to the value. The minimum size is <code>9M</code> and the maximum size is <code>1023G</code>.</p>	
<code>disk.dataPartition.initPath</code>	<p>Initial path to the data partition – Specifies the initial contents of the data partition. After wiping user data, the emulator copies the contents of the specified file to user data (by default, <code>userdata-qemu.img</code>) instead of using <code>userdata.img</code> as the initial version.</p>	
<code>disk.dataPartition.path</code>	<p>Path to the data partition – Specifies the user data partition file. To configure a persistent user data file, enter a filename and a path on your development computer. If the file doesn't exist, the emulator creates an image from the default file <code>userdata.img</code>, stores it in the filename specified by <code>disk.dataPartition.path</code>, and persists user data to it when the emulator shuts down. If you don't specify a path, the default file is named <code>userdata-qemu.img</code>. The special value <code><temp></code> causes the emulator to create and use a temporary file. If <code>disk.dataPartition.initPath</code> is set, its content will be copied to the <code>disk.dataPartition.path</code> file at boot-time. Note that this option cannot be left blank.</p>	
<code>disk.dataPartition.size</code>	<p>Data partition size – Specifies the size of the user data partition in bytes. If this value is an integer, it specifies the size in bytes. You can also specify the size in kilobytes, megabytes, and gigabytes by appending <code>K</code>, <code>M</code>, or <code>G</code> to the value. The minimum size is <code>9M</code> and the maximum size is <code>1023G</code>.</p>	

PROPERTY	DESCRIPTION	OPTIONS
<code>disk.ramdisk.path</code>	Ramdisk path – Path to the boot partition (ramdisk) image. The ramdisk image is a subset of the system image that is loaded by the kernel before the system image is mounted. The ramdisk image typically contains boot-time binaries and initialization scripts. If this option is not specified, the default is <code>ramdisk.img</code> in the emulator system directory.	
<code>disk.snapStorage.path</code>	Snapshot storage path – Path to the snapshot storage file where all snapshots are stored. All snapshots made during execution will be saved to this file. Only snapshots that are saved to this file can be restored during the emulator run. If this option is not specified, the default is <code>snapshots.img</code> in the emulator data directory.	
<code>disk.systemPartition.initPath</code>	System partition init path – Path to the read-only copy of the system image file; specifically, the partition containing the system libraries and data corresponding to the API level and any variant. If this path is not specified, the default is <code>system.img</code> in the emulator system directory.	
<code>disk.systemPartition.path</code>	System partition path – Path to the read/write system partition image. If this path is not set, a temporary file will be created and initialized from the contents of the file specified by <code>disk.systemPartition.initPath</code> .	
<code>disk.systemPartition.size</code>	System partition size – The ideal size of the system partition (in bytes). The size is ignored if the actual system partition image is larger than this setting; otherwise, it specifies the maximum size that the system partition file can grow to. If this value is an integer, it specifies the size in bytes. You can also specify the size in kilobytes, megabytes, and gigabytes by appending K , M , or G to the value. The minimum size is 9M and the maximum size is 1023G .	
<code>hw.accelerometer</code>	Accelerometer – Determines whether the emulated device contains an accelerometer sensor. The accelerometer helps the device determine orientation (used for auto-rotation). The accelerometer reports the acceleration of the device along three sensor axes.	yes, no

PROPERTY	DESCRIPTION	OPTIONS
<code>hw.audioInput</code>	Audio recording support – Determines whether the emulated device can record audio.	yes, no
<code>hw.audioOutput</code>	Audio playback support – Determines whether the emulated device can play audio.	yes, no
<code>hw.battery</code>	Battery support – Determines whether the emulated device can run on a battery.	yes, no
<code>hw.camera</code>	Camera support – Determines whether the emulated device has a camera.	yes, no
<code>hw.camera.back</code>	Back-facing camera – Configures the back-facing camera (the lens faces away from the user). If you are using a webcam on your development computer to simulate the back-facing camera on the emulated device, this value must be set to <code>webcamn</code> , where n selects the webcam (if you have only one webcam, choose <code>webcam0</code>). If set to <code>emulated</code> , the emulator simulates the camera in software. To disable the back-facing camera, set this value to <code>none</code> . If you enable the back-facing camera, be sure to also enable <code>hw.camera</code> .	emulated, none, webcam0
<code>hw.camera.front</code>	Front-facing camera – Configures the front-facing camera (the lens faces towards the user). If you are using a webcam on your development computer to simulate the front-facing camera on the emulated device, this value must be set to <code>webcamn</code> , where n selects the webcam (if you have only one webcam, choose <code>webcam0</code>). If set to <code>emulated</code> , the emulator simulates a camera in software. To disable the front-facing camera, set this value to <code>none</code> . If you enable the front-facing camera, be sure to also enable <code>hw.camera</code> .	emulated, none, webcam0
<code>hw.camera.maxHorizontalPixels</code>	Maximum horizontal camera pixels – Configures the maximum horizontal resolution of the emulated device's camera (in pixels).	
<code>hw.camera.maxVerticalPixels</code>	Maximum vertical camera pixels – Configures the maximum vertical resolution of the emulated device's camera (in pixels).	

PROPERTY	DESCRIPTION	OPTIONS
<code>hw.cpu.arch</code>	CPU architecture – The CPU architecture to be emulated by the virtual device. If you are using Intel HAXM for hardware acceleration, select x86 for a 32-bit CPU. Select x86_64 for a 64-bit HAXM-accelerated device. (Be sure to install the corresponding Intel x86 system image in the SDK Manager; for example, Intel x86 Atom or Intel x86 Atom_64.) To simulate an ARM CPU, select arm for 32-bit or select arm64 for a 64-bit ARM CPU. Keep in mind that ARM-based virtual devices will run much slower than those that are x86-based because hardware acceleration is not available for ARM.	x86, x86_64, arm, arm64
<code>hw.cpu.model</code>	CPU model – This value is normally left unset (it will be set to a value that is derived from <code>hw.cpu.arch</code> if it is not explicitly set). However, it can be set to an emulator-specific string for experimental use.	
<code>hw.dPad</code>	DPad keys – Determines whether the emulated device supports directional pad (DPad) keys. A DPad typically has four keys to indicate directional control.	yes, no
<code>hw.gps</code>	GPS support – Determines whether the emulated device has a GPS (Global Positioning System) receiver.	yes, no
<code>hw.gpu.enabled</code>	GPU emulation – Determines whether the emulated device supports GPU emulation. When enabled, GPU emulation uses Open GL for Embedded Systems (OpenGL ES) for rendering both 2D and 3D graphics on the screen, and the associated GPU Emulation Mode setting determines how the GPU emulation is implemented.	yes, no

PROPERTY	DESCRIPTION	OPTIONS
hw.gpu.mode	GPU emulation mode – Determines how GPU emulation is implemented by the emulator. If you select auto, the emulator will choose hardware and software acceleration based on your development computer setup. If you select host, the emulator will use your development computer's graphics processor to perform GPU emulation for faster rendering. If your GPU is not compatible with the emulator and you are on Windows, you can try angle instead of host. The angle mode uses DirectX to provide performance similar to host. If you select mesa, the emulator will use the Mesa 3D software library to render graphics. Select mesa if you have problems rendering via your development computer's graphics processor. The swiftshader mode can be used to render graphics in software with slightly less performance than using your computer's GPU. The off option (disable graphics hardware emulation) is a deprecated option that can cause improper rendering for some items and is therefore not recommended.	auto, host, mesa, angle, swiftshader, off
hw.gsmModem	GSM modem support – Determines whether the emulated device includes a modem that supports the GSM (Global System for Mobile Communications) telephony radio system.	yes, no
hw.initialOrientation	Initial screen orientation – Configures the initial orientation of the screen on the emulated device (portrait or landscape mode). In portrait mode, the screen is taller than it is wide. In landscape mode, the screen is wider than it is tall. When running the emulated device, you can change the orientation if both portrait and landscape are supported in the device profile.	portrait, landscape
hw.keyboard	Keyboard support – Determines whether the emulated device supports a QWERTY keyboard.	yes, no

PROPERTY	DESCRIPTION	OPTIONS
hw.keyboard.charmap	Keyboard charmap name – The name of the hardware charmap for this device. NOTE: This should always be the default <code>qwerty2</code> unless you have modified the system image accordingly. This name is sent to the kernel at boot time. Using an incorrect name will result in an unusable virtual device.	
hw.keyboard.lid	Keyboard lid support – If keyboard support is enabled, this setting determines whether the QWERTY keyboard can be closed/hidden or opened/visible. This setting will be ignored if <code>hw.keyboard</code> is set to false. NOTE: the default value is false if the emulated device targets API level 12 or higher.	yes, no
hw.lcd.backlight	LCD backlight – Determines whether an LCD backlight is simulated by the emulated device.	yes, no
hw.lcd.density	LCD density – The density of the emulated LCD display, measured in density-independent pixels, or dp (dp is a virtual pixel unit). When the setting is 160 dp, each dp corresponds to one physical pixel. At runtime, Android uses this value to select and scale the appropriate resources/assets for correct display rendering.	120, 160, 240, 213, 320
hw.lcd.depth	LCD color depth – The color bit-depth of the emulated framebuffer that holds the bitmap for driving the LCD display. This value can be 16 bits (65,536 possible colors) or 32 bits (16,777,216 colors plus transparency). The 32-bit setting can make the emulator run slightly slower but with better color accuracy.	16, 32
hw.lcd.height	LCD pixel height – The number of pixels that make up the vertical dimension of the emulated LCD display.	
hw.lcd.width	LCD pixel width – The number of pixels that make up the horizontal dimension of the emulated LCD display.	

PROPERTY	DESCRIPTION	OPTIONS
<code>hw.mainKeys</code>	Hardware Back/Home keys – Determines whether the emulated device supports hardware Back and Home navigation buttons. You can set this value to yes if the buttons are implemented only in software. If <code>hw.mainKeys</code> is set to yes , the emulator will not display navigation buttons on the screen, but you can use the emulator side panel to "press" these buttons.	yes, no
<code>hw.ramSize</code>	Device RAM Size – The amount of physical RAM on the emulated device, in megabytes. The default value will be computed from the screen size or the skin version. Increasing the size can provide faster emulator operation, but at the expense of demanding more resources from your development computer.	
<code>hw.screen</code>	Touch screen type – Defines the type of screen on the emulated device. A multi-touch screen can track two or more fingers on the touch interface. A touch screen can detect only single-finger touch events. A no-touch screen does not detect touch events.	touch, multi-touch, no-touch
<code>hw.sdCard</code>	SDCard support – Determines whether the emulated device supports insertion and removal of virtual SD (Secure Digital) cards. The emulator uses mountable disk images stored on your development computer to simulate the partitions of actual SD card devices (see <code>hw.sdCard.path</code>).	yes, no
<code>sdcard.size</code>	SDCard size – Specifies the size of the virtual SD card file at the location specified by <code>hw.sdCard.path</code> . available on the device (in bytes). If this value is an integer, it specifies the size in bytes. You can also specify the size in kilobytes, megabytes, and gigabytes by appending K , M , or G to the value. The minimum size is 9M and the maximum size is 1023G .	
<code>hw.sdCard.path</code>	SDCard Image Path – Specifies the filename and path to an SD card partition image file on your development computer. For example, this path could be set to <code>C:\sd\sdcard.img</code> on Windows.	

PROPERTY	DESCRIPTION	OPTIONS
<code>hw.sensors.magnetic_field</code>	Magnetic Field Sensor – Determines whether the emulated device supports a magnetic field sensor. The magnetic field sensor (also known as magnetometer) reports the ambient geomagnetic field as measured along three sensor axes. Enable this setting for apps that need access to a compass reading. For example, a navigation app might use this sensor to detect which direction the user faces.	yes, no
<code>hw.sensors.orientation</code>	Orientation Sensor – Determines whether the emulated device provides orientation sensor values. The orientation sensor measures degrees of rotation that a device makes around all three physical axes (x, y, z). Note that the orientation sensor was deprecated as of Android 2.2 (API level 8).	yes, no
<code>hw.sensors.proximity</code>	Proximity Sensor – Determines whether the emulated device supports a proximity sensor. This sensor measures the proximity of an object relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	yes, no
<code>hw.sensors.temperature</code>	Temperature Sensor – Determines whether the emulated device supports a temperature sensor. This sensor measures the temperature of the device in degrees Celsius (°C).	yes, no
<code>hw.touchScreen</code>	Touch-screen support – Determines whether the emulated device supports a touch screen. The touch screen is used for direct manipulation of objects on the screen.	yes, no
<code>hw.trackBall</code>	Trackball support – Determines whether the emulated device supports a trackball.	yes, no
<code>hw.useext4</code>	EXT4 file system support – Determines whether the emulated device uses the Linux EXT4 file system for partitions. Because the file system type is now auto-detected, this option is deprecated and ignored.	no

PROPERTY	DESCRIPTION	OPTIONS
<code>kernel.newDeviceNaming</code>	Kernel new device naming – Used to specify whether the kernel requires a new device naming scheme. This is typically used with Linux 3.10 kernels and later. If set to autodetect , the emulator will automatically detect whether the kernel requires a new device naming scheme.	autodetect, yes, no
<code>kernel.parameters</code>	Kernel parameters – Specifies the string of Linux kernel boot parameters. By default, this setting is left blank.	
<code>kernel.path</code>	Kernel path – Specifies the path to the Linux kernel. If this path is not specified, the emulator looks in the emulator system directory for kernel-ranchu.	
<code>kernel.supportsYaffs2</code>	YAFFS2 partition support – Determines whether the kernel supports YAFFS2 (Yet Another Flash File System 2) partitions. Typically, this applies only to kernels before Linux 3.10. If set to autodetect the emulator will automatically detect whether the kernel can mount YAFFS2 file systems.	autodetect, yes, no
<code>skin.name</code>	Skin name – The name for an Android emulator skin. A skin is a collection of files that defines the visual and control elements of an emulator display; it describes what the window of the AVD will look like on your development computer. A skin describes screen size, buttons, and the overall design, but it does not affect the operation of your app.	
<code>skin.path</code>	Skin path – Path to the directory that contains the emulator skin files specified in <code>skin.name</code> . This directory contains hardware.ini layout files, and image files for the display elements of the skin.	
<code>skin.dynamic</code>	Skin dynamic – Whether or not the skin is dynamic. The emulator skin is a dynamic skin if the emulator is to construct a skin of a given size based on a specified width and height.	no

For more information about these properties, see [Hardware Profile Properties](#).

Debug on the Android Emulator

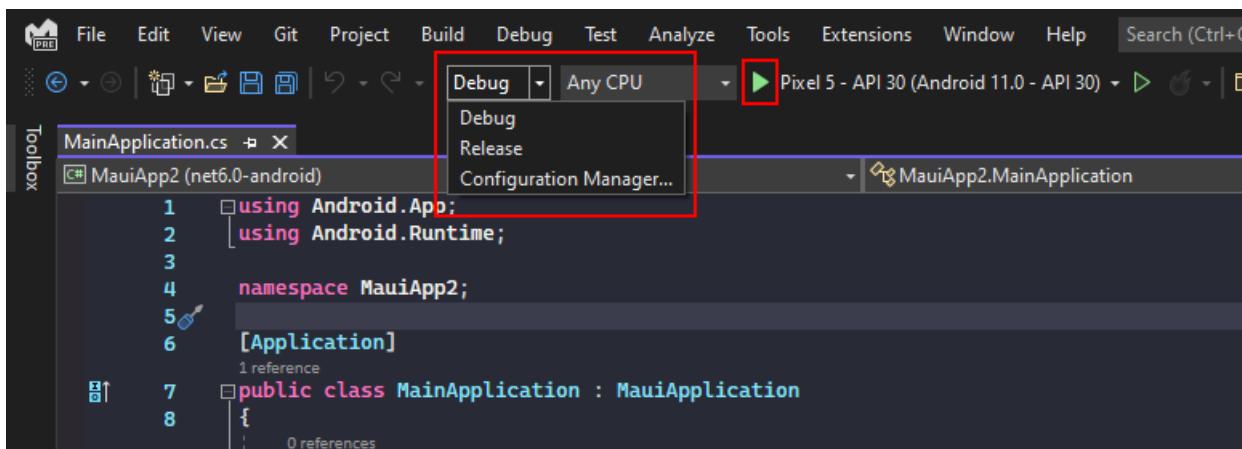
9/20/2022 • 2 minutes to read • [Edit Online](#)

The Android Emulator, installed as part of the **.NET Multi-Platform App UI development** workload, can be run in various configurations to simulate different Android devices. Each one of these configurations is created as a *virtual device*. In this article, you'll learn how to launch the emulator from Visual Studio and run your app in a virtual device. For more information about how to create and configure a virtual device, see [Managing virtual devices with the Android Device Manager](#).

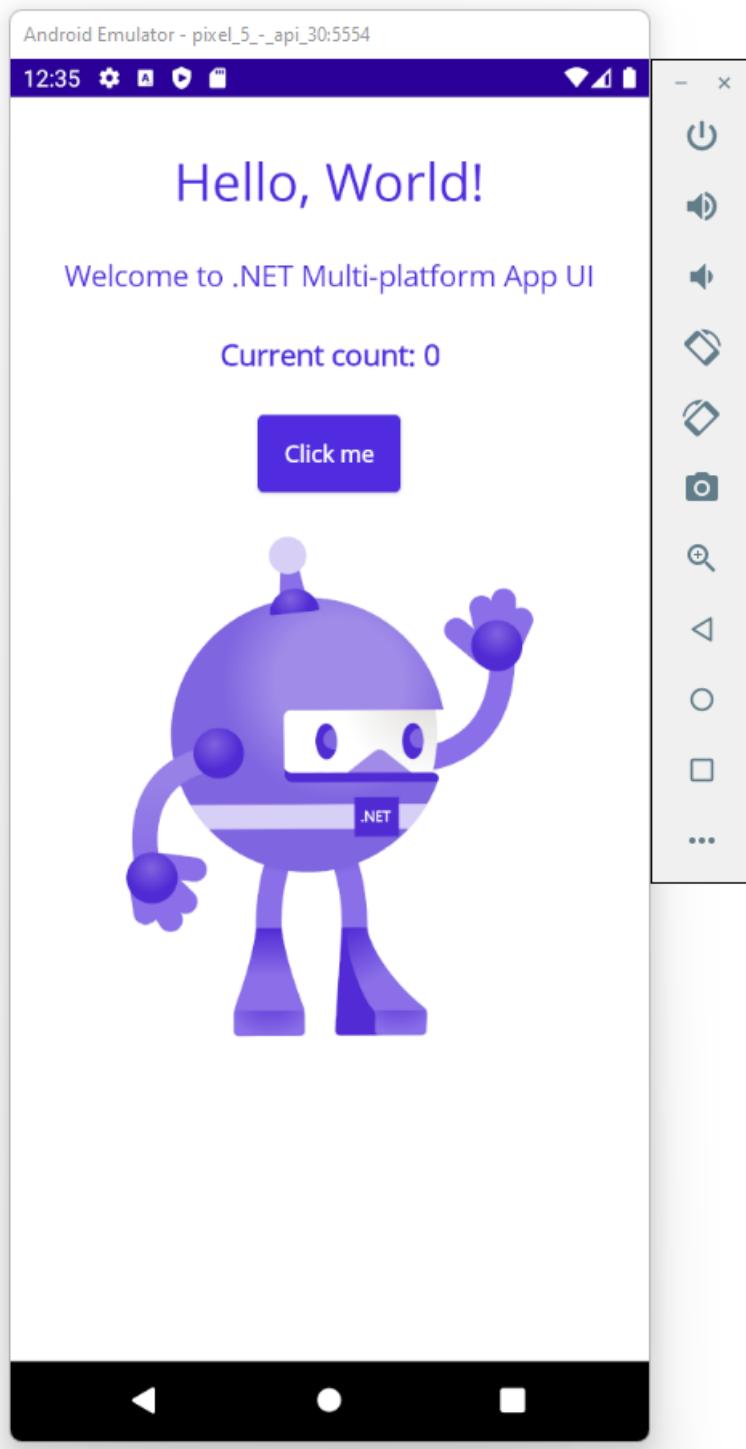
Launching the Emulator

Near the top of Visual Studio, there's the **Solution Configurations** drop-down menu that can be used to select **Debug** or **Release** mode. Choosing **Debug** causes the debugger to attach to the application process running inside the emulator after the app starts. Choosing **Release** mode disables the debugger. When in release mode, you'll need to rely on app logging for debugging.

After you've chosen a virtual device from the **Debug Target** device drop-down menu, select either **Debug** or **Release** mode, then select the Play button to run the application:



After the emulator starts, Visual Studio deploys the app to the virtual device. An example screenshot of the Android Emulator is displayed below. In this example, the emulator is running the .NET MAUI template app.

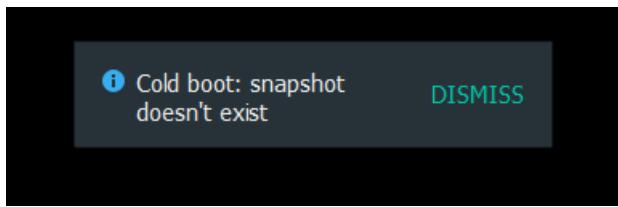


When you're finished debugging and running your app, you can leave the emulator running. The first time a .NET MAUI app is run in the emulator, the .NET MAUI shared runtime for the targeted API level is installed, followed by the app. The runtime installation may take a few moments to install. If you leave the emulator running, later debugging sessions start faster as the runtime is already present on the device. If the device is restarted, the runtime will be redeployed to the device.

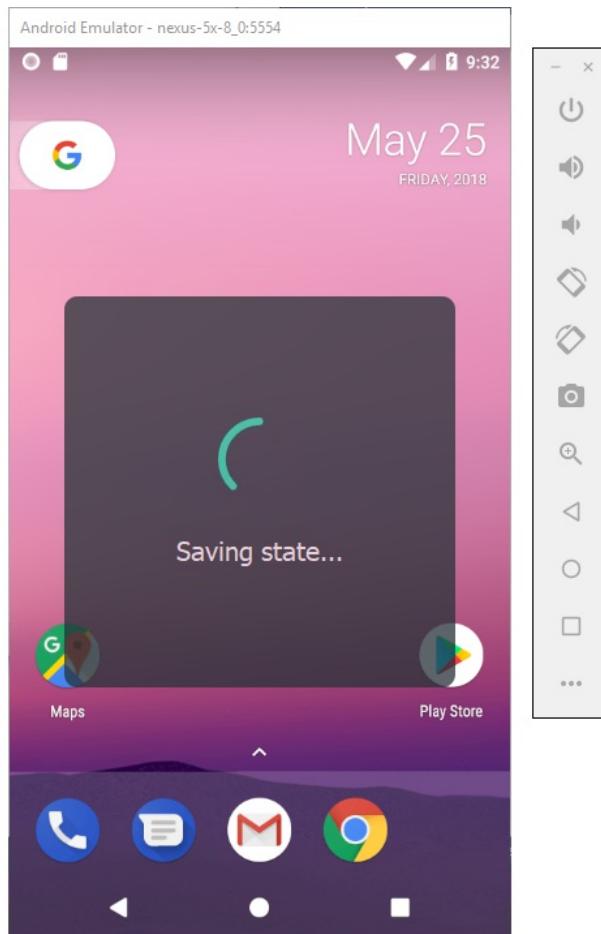
Fast boot

The Android Emulator includes a feature named Fast Boot which is enabled by default. This feature is configured by each device's emulator settings. With this feature enabled, a snapshot of the virtual device is saved when the emulator is closed. The snapshot is quickly restored the next time the device is started.

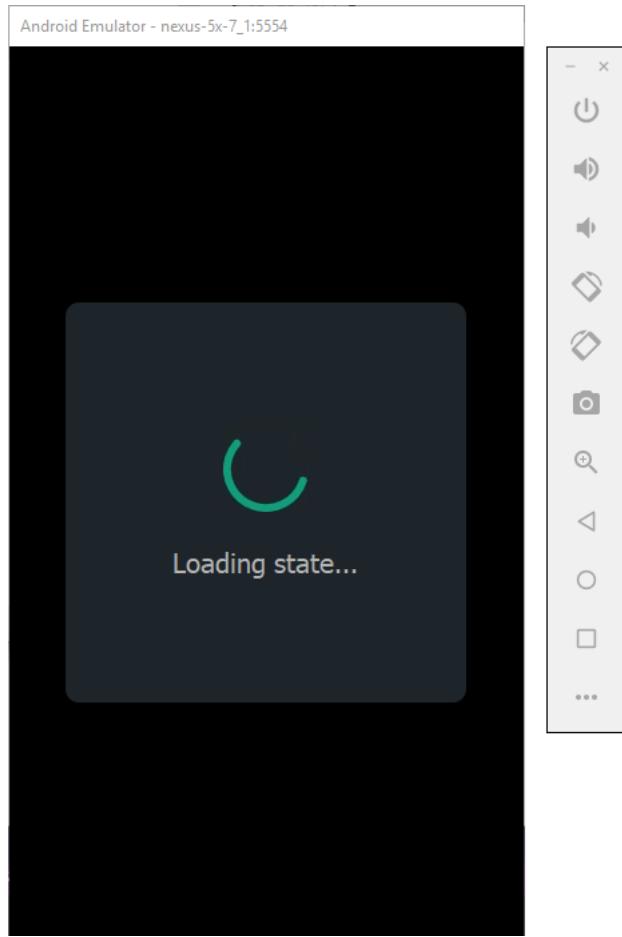
The first time a virtual device is started, a cold boot of the virtual device takes place without a speed improvement because a snapshot hasn't yet been created:



When you exit out of the emulator, Fast Boot saves the state of the emulator in a snapshot:



The next time the virtual device starts, it loads much faster because the emulator simply restores the state at which you closed the emulator.



Troubleshooting

For tips and workarounds for common emulator problems, see [Android Emulator Troubleshooting](#).

For more information about using the Android Emulator, see the following Android Developer articles:

- [Navigating on the Screen](#)
- [Performing Basic Tasks in the Emulator](#)
- [Working with Extended Controls, Settings, and Help](#)
- [Run the emulator with Quick Boot](#)

Android emulator troubleshooting

9/20/2022 • 10 minutes to read • [Edit Online](#)

This article describes the most common warning messages and issues that occur while configuring and running the Android Emulator. Also, it describes solutions for resolving these errors and various troubleshooting tips to help you diagnose emulator problems.

Deployment issues on Windows

Some error messages may be displayed by the emulator when you deploy your app. The most common errors and solutions are explained here.

Deployment errors

If you see an error about a failure to install the APK on the emulator or a failure to run the Android Debug Bridge (**adb**), verify that the Android SDK can connect to your emulator. To verify emulator connectivity, use the following steps:

1. Launch the emulator from the **Android Device Manager** (select your virtual device and select **Start**).
2. Open a command prompt and go to the folder where **adb** is installed. If the Android SDK is installed at its default location, **adb** is located at *C:\Program Files (x86)\Android\android-sdk\platform-tools\adb.exe*; if not, modify this path for the location of the Android SDK on your computer.
3. Type the following command:

```
adb devices
```

4. If the emulator is accessible from the Android SDK, the emulator should appear in the list of attached devices. For example:

```
List of devices attached  
emulator-5554    device
```

5. If the emulator doesn't appear in this list, start the **Android SDK Manager**, apply all updates, then try launching the emulator again.

MMIO access error

If the message **An MMIO access error has occurred** is displayed, restart the emulator.

Missing Google Play Services

If the emulated Android device doesn't have Google Play Services or Google Play Store installed, you probably created a virtual device that excluded these packages. When you [create a virtual device](#), be sure to select one or both of the following options:

- **Google APIs**—includes Google Play Services in the virtual device.
- **Google Play Store**—includes Google Play Store in the virtual device.

For example, this virtual device will include Google Play Services and Google Play Store:



NOTE

Google Play Store images are available only for some base device types such as Pixel, Pixel 2, Nexus 5, and Nexus 5X.

Performance issues

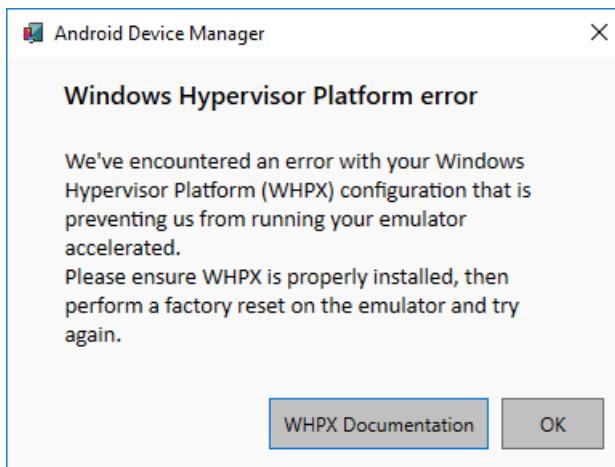
Performance issues are typically caused by one of the following problems:

- The emulator is running without hardware acceleration.
- The virtual device running in the emulator using an Arm-based image.

The following sections cover these scenarios in more detail.

Hardware acceleration isn't enabled

When you start a virtual device, and you don't have hardware acceleration enabled, the Device Manager displays an error dialog similar to the following image:



To fix this error, follow the troubleshooting steps in the [Hardware acceleration issues](#) section.

Hardware acceleration issues

When using hardware acceleration, you may run into configuration problems or conflicts with other software on your computer. The first step in troubleshooting is verifying that hardware acceleration is enabled. You can use

the Android's SDK to check this setting. Open a command prompt and entering the following command:

```
"C:\Program Files (x86)\Android\android-sdk\emulator\emulator-check.exe" accel
```

This command assumes that the Android SDK is installed at the default location of *C:\Program Files (x86)\Android\android-sdk*. If the Android SDK is installed elsewhere, modify the preceding command to the correct location.

Hardware acceleration not available

If Hyper-V is available, a message like the following example will be returned from the **emulator-check.exe accel** command:

```
HAXM isn't installed, but Windows Hypervisor Platform is available.
```

If HAXM is available, a message like the following example will be returned:

```
HAXM version 6.2.1 (4) is installed and usable.
```

If hardware acceleration isn't available, a message like the following example will be displayed (the emulator looks for HAXM if it's unable to find Hyper-V):

```
HAXM isn't installed on this machine
```

If hardware acceleration isn't available, see [Enabling Hyper-V acceleration](#) to learn how to enable hardware acceleration on your computer.

Incorrect BIOS settings

If the BIOS hasn't been configured properly to support hardware acceleration, a message similar to the following example will be displayed when you run the **emulator-check.exe accel** command:

```
VT feature disabled in BIOS/UEFI
```

To correct this problem, reboot into your computer's BIOS and enable the following options:

- Virtualization Technology (may have a different label depending on motherboard manufacturer).
- Hardware Enforced Data Execution Prevention.

If problems still occur because of issues related to Hyper-V and HAXM, see the following section.

Hyper-V issues

In some cases, enabling both **Hyper-V** and **Windows Hypervisor Platform** in the **Turn Windows features on or off** dialog may not properly enable Hyper-V. To verify that Hyper-V is enabled, use the following steps:

1. Enter **PowerShell** in the Windows search box.
2. Right-click **Windows PowerShell** in the search results and select **Run as administrator**.
3. In the PowerShell console, enter the following command:

```
Get-WindowsOptionalFeature -FeatureName Microsoft-Hyper-V-All -Online
```

If Hyper-V isn't enabled, a message similar to the following example will be displayed to indicate that the

state of Hyper-V is **Disabled**:

```
FeatureName      : Microsoft-Hyper-V-All
DisplayName      : Hyper-V
Description       : Provides services and management tools for creating and running virtual machines
and their resources.
RestartRequired   : Possible
State            : Disabled
CustomProperties :
```

4. In the PowerShell console, enter the following command:

```
Get-WindowsOptionalFeature -FeatureName HypervisorPlatform -Online
```

If the Hypervisor isn't enabled, a message similar to the following example will be displayed to indicate that the state of HypervisorPlatform is **Disabled**:

```
FeatureName      : HypervisorPlatform
DisplayName      : Windows Hypervisor Platform
Description       : Enables virtualization software to run on the Windows hypervisor
RestartRequired   : Possible
State            : Disabled
CustomProperties :
```

If Hyper-V or HypervisorPlatform aren't enabled, use the following PowerShell commands to enable them:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All
Enable-WindowsOptionalFeature -Online -FeatureName HypervisorPlatform -All
```

After these commands complete, reboot.

For more information about enabling Hyper-V (including techniques for enabling Hyper-V using the Deployment Image Servicing and Management tool), see [Install Hyper-V](#).

HAXM issues

HAXM issues are often the result of conflicts with other virtualization technologies, incorrect settings, or an out-of-date HAXM driver.

HAXM process isn't running

If HAXM is installed, you can verify that the HAXM process is running by opening a command prompt and entering the following command:

```
sc query intelhaxm
```

If the HAXM process is running, you should see output similar to the following result:

```
SERVICE_NAME: intelhaxm
  TYPE            : 1  KERNEL_DRIVER
  STATE           : 4  RUNNING
                    (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
  WIN32_EXIT_CODE : 0  (0x0)
  SERVICE_EXIT_CODE : 0  (0x0)
  CHECKPOINT     : 0x0
  WAIT_HINT      : 0x0
```

If **STATE** isn't set to **RUNNING**, see [How to Use the Intel Hardware Accelerated Execution Manager](#) to resolve the problem.

HAXM virtualization conflicts

HAXM can conflict with other technologies that use virtualization, such as Hyper-V, Windows Device Guard, and some antivirus software:

- **Hyper-V**—If you're using a version of Windows before the [Windows 10 April 2018 update \(build 1803\)](#) and Hyper-V is enabled, follow the steps in [Disabling Hyper-V](#) so that HAXM can be enabled.
- **Device Guard**—Device Guard and Credential Guard can prevent Hyper-V from being disabled on Windows machines. To disable Device Guard and Credential Guard, see [Disabling Device Guard](#).
- **Antivirus Software**—If you're running antivirus software that uses hardware-assisted virtualization (such as Avast), disable or uninstall this software, reboot, and retry the Android emulator.

Incorrect BIOS settings for HAXM

On Windows, HAXM won't work unless virtualization technology (Intel VT-x) is enabled in the BIOS. If VT-x is disabled, you'll get an error similar to the following when you attempt to start the Android Emulator:

This computer meets the requirements for HAXM, but Intel Virtualization Technology (VT-x) isn't turned on.

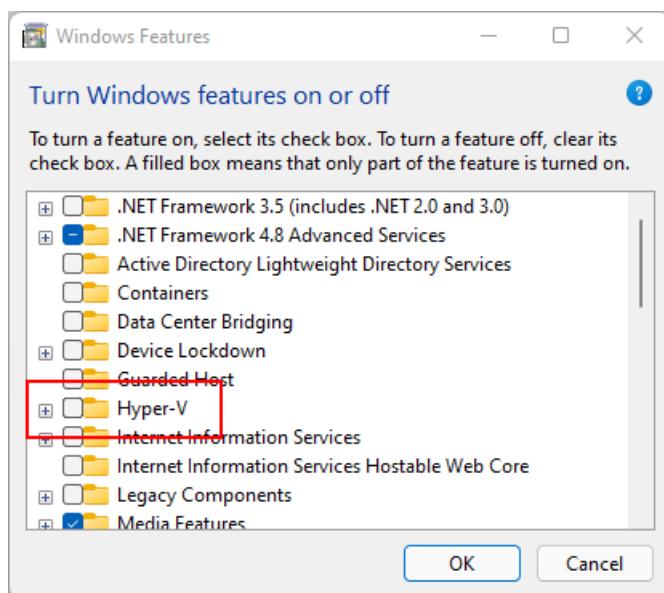
To correct this error, boot the computer into the BIOS, enable both VT-x and SLAT (Second-Level Address Translation) and restart the computer.

Disabling Hyper-V

If you're using a version of Windows before the [Windows 10 April 2018 Update \(build 1803\)](#) and Hyper-V is enabled, you must disable Hyper-V and reboot your computer to install and use HAXM. If you're using [Windows 10 April 2018 Update \(build 1803\)](#) or later, Android Emulator version 27.2.7 or later can use Hyper-V (instead of HAXM) for hardware acceleration, so it isn't necessary to disable Hyper-V.

You can disable Hyper-V from the Control Panel by following these steps:

1. Enter **windows features** in the Windows search box and select **Turn Windows features on or off** in the search results.
2. Uncheck **Hyper-V**:



3. Restart the computer.

Alternately, you can use the following PowerShell command to disable the Hyper-V Hypervisor:

```
Disable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V-Hypervisor
```

Intel HAXM and Microsoft Hyper-V can't both be active at the same time. Unfortunately, there's no way to switch between Hyper-V and HAXM without restarting your computer.

It's possible that the preceding steps won't succeed in disabling Hyper-V if Device Guard and Credential Guard are enabled. If you're unable to disable Hyper-V, or it seems to be disabled but HAXM installation still fails, use the steps in the next section to disable Device Guard and Credential Guard.

Disabling Device Guard

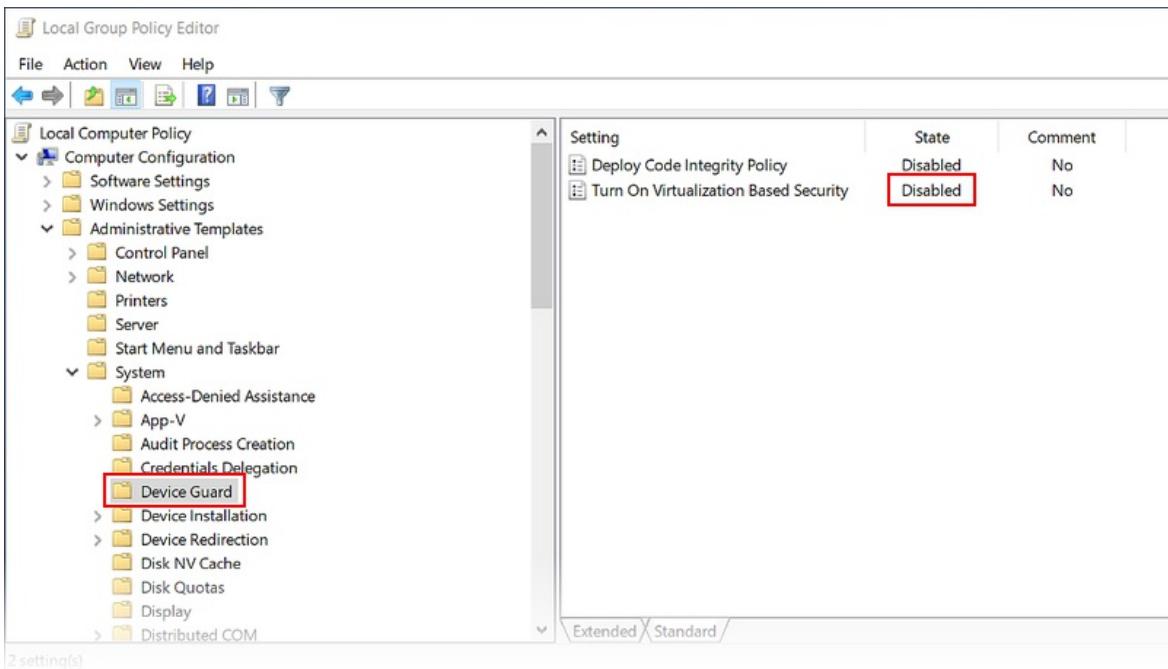
Device Guard and Credential Guard can prevent Hyper-V from being disabled on Windows machines. This situation is often a problem for domain-joined machines that are configured and controlled by an owning organization. On Windows 10, use the following steps to see if **Device Guard** is running:

1. Enter **System info** in the Windows search box and select **System Information** in the search results.
2. In the **System Summary**, look to see if **Device Guard Virtualization based security** is present and is in the **Running** state:



If Device Guard is enabled, use the following steps to disable it:

1. Ensure that **Hyper-V** is disabled (under **Turn Windows Features on or off**) as described in the previous section.
2. In the Windows Search Box, enter **gpedit.msc** and select the **Edit group policy** search result. These steps launch the **Local Group Policy Editor**.
3. In the **Local Group Policy Editor**, navigate to **Computer Configuration > Administrative Templates > System > Device Guard**:



4. Change Turn On Virtualization Based Security to Disabled (as shown above) and exit the Local Group Policy Editor.
5. In the Windows Search Box, enter cmd. When Command Prompt appears in the search results, right-click Command Prompt and select Run as Administrator.
6. Copy and paste the following commands into the command prompt window (if drive Z: is in use, pick an unused drive letter to use instead):

```
mountvol Z: /s
copy %WINDIR%\System32\SecConfig.efi Z:\EFI\Microsoft\Boot\SecConfig.efi /Y
bcdedit /create {0cb3b571-2f2e-4343-a879-d86a476d7215} /d "DebugTool" /application osloader
bcdedit /set {0cb3b571-2f2e-4343-a879-d86a476d7215} path "\EFI\Microsoft\Boot\SecConfig.efi"
bcdedit /set {bootmgr} bootsequence {0cb3b571-2f2e-4343-a879-d86a476d7215}
bcdedit /set {0cb3b571-2f2e-4343-a879-d86a476d7215} loadoptions DISABLE-LSA-ISO,DISABLE-VBS
bcdedit /set {0cb3b571-2f2e-4343-a879-d86a476d7215} device partition=Z:
mountvol Z: /d
```

7. Restart your computer. On the boot screen, you should see a prompt similar to the following message:

Do you want to disable Credential Guard?

Press the indicated key to disable Credential Guard as prompted.

8. After the computer reboots, check again to ensure that Hyper-V is disabled (as described in the previous steps).

If Hyper-V is still not disabled, the policies of your domain-joined computer may prevent you from disabling Device Guard or Credential Guard. In this case, you can request an exemption from your domain administrator to allow you to opt out of Credential Guard. Alternately, you can use a computer that isn't domain-joined if you must use HAXM.

More troubleshooting tips

The following suggestions are often helpful in diagnosing Android emulator issues.

Starting the emulator from the command line

If the emulator isn't already running, you can start it from the command line (rather than from within Visual

Studio) to view its output. Typically, Android emulator AVD images are stored at the following location:
%userprofile%\android\avd.

You can launch the emulator with an AVD image from this location by passing in the folder name of the AVD. For example, this command launches an AVD named **Pixel_API_27**:

```
"C:\Program Files (x86)\Android\android-sdk\emulator\emulator.exe" -partition-size 2000 -no-boot-anim -verbose -feature WindowsHypervisorPlatform -avd pixel_5_-_api_30 -prop monodroid.avdname=pixel_5_-_api_30
```

This command assumes that the Android SDK is installed at the default location of *C:\Program Files (x86)\Android\android-sdk*. If the Android SDK is installed elsewhere, modify the preceding command to the correct location.

When you run this command, it produces many lines of output while the emulator starts up. Specifically, lines such as the following example are printed if hardware acceleration is enabled and working properly. In this example, HAXM is used for hardware acceleration:

```
emulator: CPU Acceleration: working  
emulator: CPU Acceleration status: HAXM version 6.2.1 (4) is installed and usable.
```

Viewing Device Manager logs

Often you can diagnose emulator problems by viewing the Device Manager logs. These logs are written to the following location: %userprofile%\AppData\Local\Xamarin\Logs\16.0.

You can view each *DeviceManager.log* file by using a text editor such as Notepad. The following example log entry indicates that HAXM wasn't found on the computer:

```
Component Intel x86 Emulator Accelerator (HAXM installer) r6.2.1 [Extra: (Intel Corporation)] not present on the system
```

Set up Android device for debugging

9/20/2022 • 3 minutes to read • [Edit Online](#)

While the [Android emulator](#) is a great way to rapidly develop and test your app, you'll want to test your apps on a real Android device. To run on a device, you'll need to enable developer mode on the device and connect it to your computer.

IMPORTANT

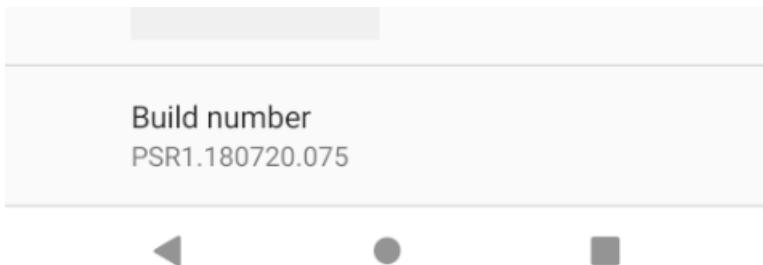
The steps in this article are written generically, to work on as many devices as possible. If you can't find these settings on your device, consult your device manufacturer's documentation.

Enable developer mode on the device

A device must enable Developer mode in order to deploy and test an Android app. Developer mode is enabled by following these steps:

1. Go to the **Settings** screen.
2. Select **About phone**.
3. Tap **Build Number** seven times until **You are now a developer!** is visible.

Depending on the UI your device is running, the **About phone** option may be in a different location. Consult your device documentation if you can't find **About phone**.



Enable USB debugging

After enabling developer mode on your device, enable USB debugging by following these steps:

1. Go to the **Settings** screen.
2. Select **Developer options**.
3. Turn on the **USB debugging** option.

Depending on the UI your device is running, the **USB debugging** option may be in a different location. Consult your device documentation if you can't find **USB debugging**.

Connect the device to the computer

The final step is to connect the device to the computer. The easiest and most reliable way is to do so over USB.

You'll receive a prompt to trust the computer on your device if you haven't used it for debugging before. You can also check **Always allow from this computer** to prevent requiring this prompt each time you connect the device.

Allow USB debugging?

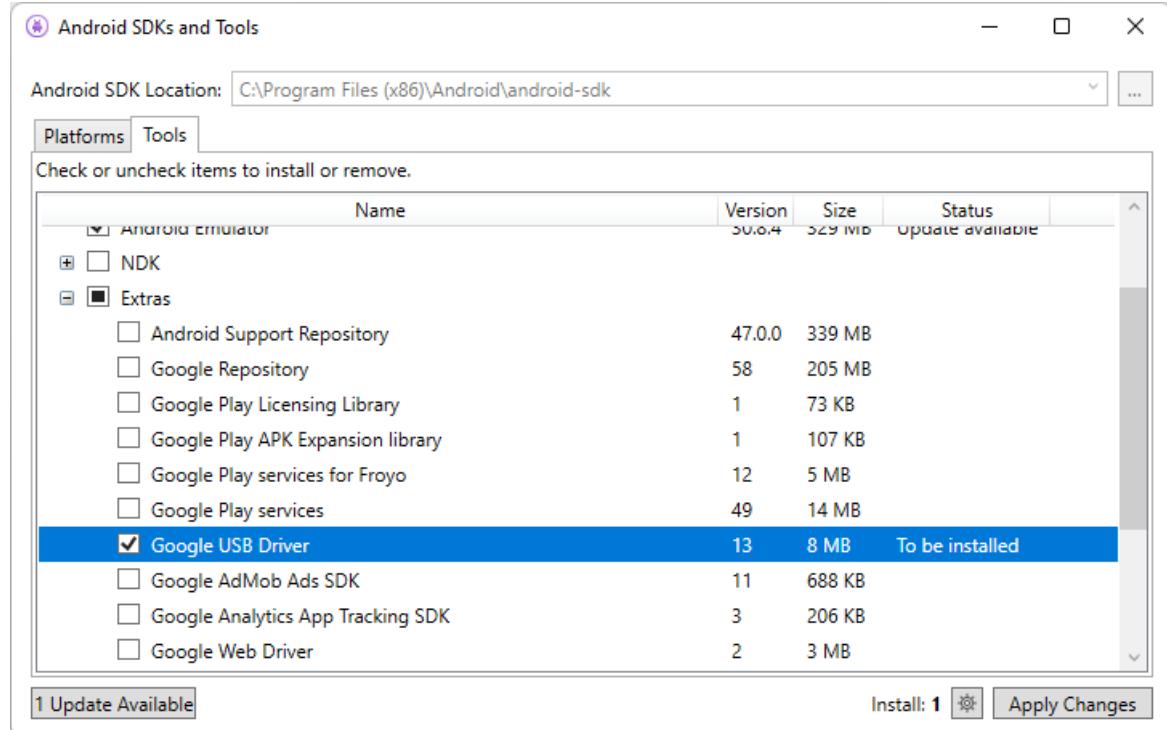
The computer's RSA key fingerprint is:

 Always allow from this computer

[Cancel](#)

[OK](#)

If your computer isn't recognizing the device when it's plugged in, try installing a driver for the device. Consult your device manufacturer's support documentation. You can also try installing the Google USB Driver through the Android SDK Manager:



Enable WiFi debugging

It's possible to debug an android device over WiFi, without keeping the device physically connected to the computer. This technique requires more effort, but could be useful when the device is too far from the computer to remain constantly plugged-in via a cable.

Connecting over WiFi

By default, the [Android Debug Bridge](#) (adb) is configured to communicate with an Android device via USB. It's possible to reconfigure it to use TCP/IP instead of USB. To do this, both the device and the computer must be on the same WiFi network.

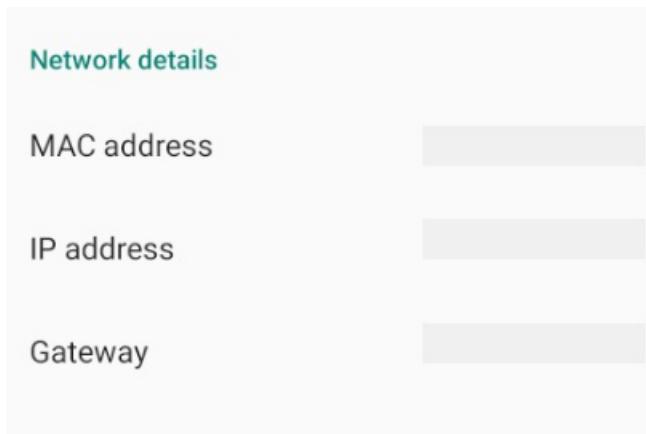
First, enable Wireless debugging on your Android device:

1. Follow the steps in the [Enable developer mode on the device](#) section.
2. Follow the steps in the [Enable USB debugging](#) section.
3. Go to the **Settings** screen.
4. Select **Developer options**.
5. Turn on the **Wireless debugging** option.

Depending on the UI your device is running, the **Wireless debugging** option may be in a different location. Consult your device documentation if you can't find **Wireless debugging**.

Next, use adb to connect to your device, first through a USB connection:

1. Determine the IP address of your Android device. One way to find out the IP address is to look under **Settings > Network & internet > Wi-Fi**, then tap on the WiFi network that the device is connected to, and then tap on **Advanced**. This will open a dropdown showing information about the network connection, similar to what is seen in the screenshot below:



On some versions of Android the IP address won't be listed there but can be found instead under **Settings > About phone > Status**.

2. In Visual Studio, open the adb command prompt by selecting the menu option: **Tools > Android > Android Adb Command Prompt....**
3. In the command prompt, use the `adb tcpip 5555` command to tell the device to listen to TCP/IP connections on port 5555.

```
adb tcpip 5555
```

4. Disconnect the USB cable from your device.

5. Connect to the device's IP address with port 5555:

```
adb connect 192.168.1.28:5555
```

When this command finishes, the Android device is connected to the computer via WiFi.

When you're finished debugging via WiFi, you can reset ADB back to USB mode with the following command:

```
adb usb
```

To see the devices connected to the computer, use the `adb devices` command:

```
adb devices
```

Build an iOS app with .NET CLI

9/20/2022 • 2 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to create and run a .NET Multi-platform App UI (.NET MAUI) app on iOS using .NET Command Line Interface (CLI) on macOS:

1. To create .NET MAUI apps, you'll need to download and run the [installer](#) for the latest .NET 6 runtime. You'll also need to download and install the latest version of [Xcode 13](#), which is also available from the App Store app on your Mac.
2. On your Mac, open **Terminal** and check that you have the latest .NET 6 runtime installed:

```
dotnet --version
```

3. In **Terminal**, install the latest public build of .NET MAUI:

```
sudo dotnet workload install maui --source https://api.nuget.org/v3/index.json
```

This command will install the latest released version of .NET MAUI, including the required platform SDKs.

4. In **Terminal**, create a new .NET MAUI app using .NET CLI:

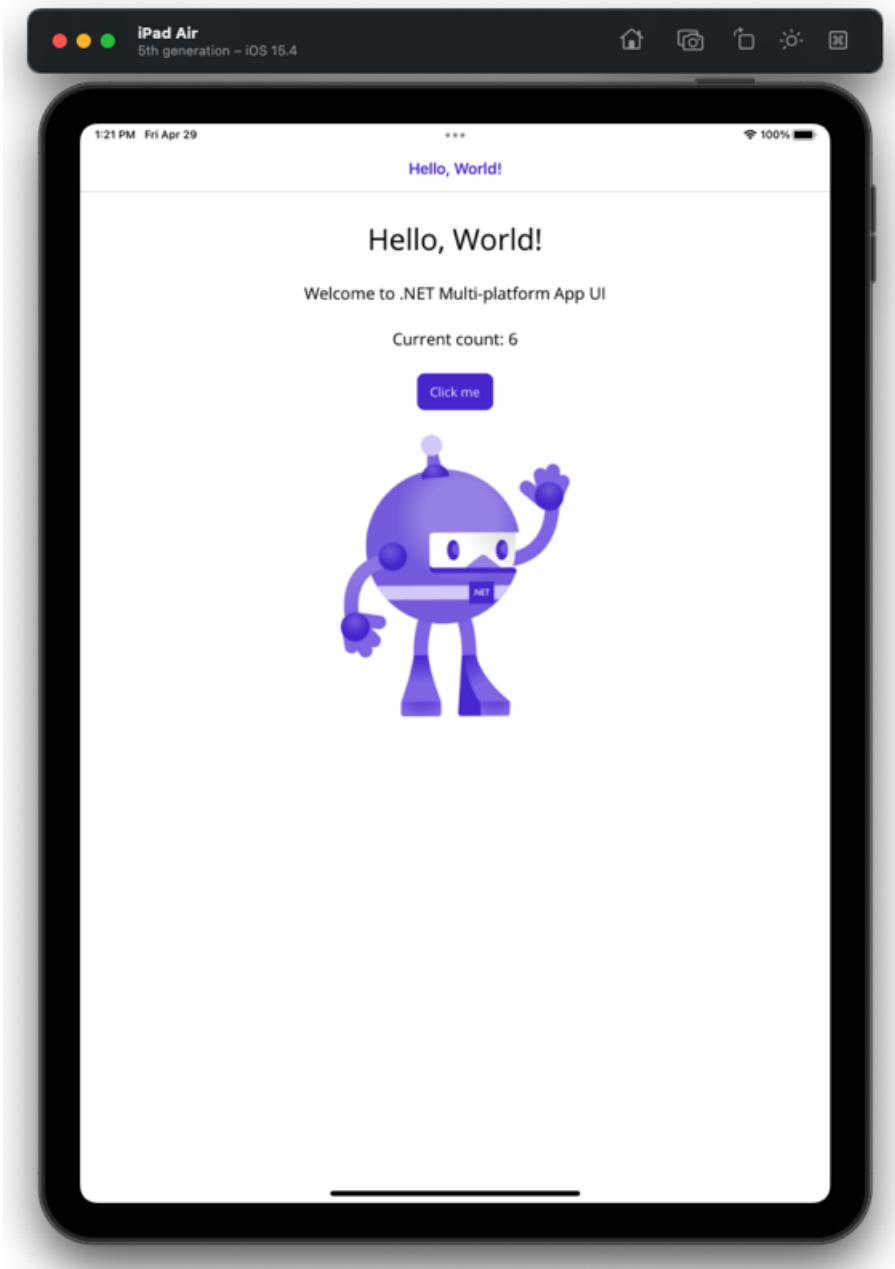
```
dotnet new maui -n "MyMauiApp"
```

5. In **Terminal**, change directory to *MyMauiApp*, and build and run the app:

```
cd MyMauiApp  
dotnet build -t:Run -f net6.0-ios
```

The `dotnet build` command will restore the project the dependencies, build the app, and launch it in the default simulator.

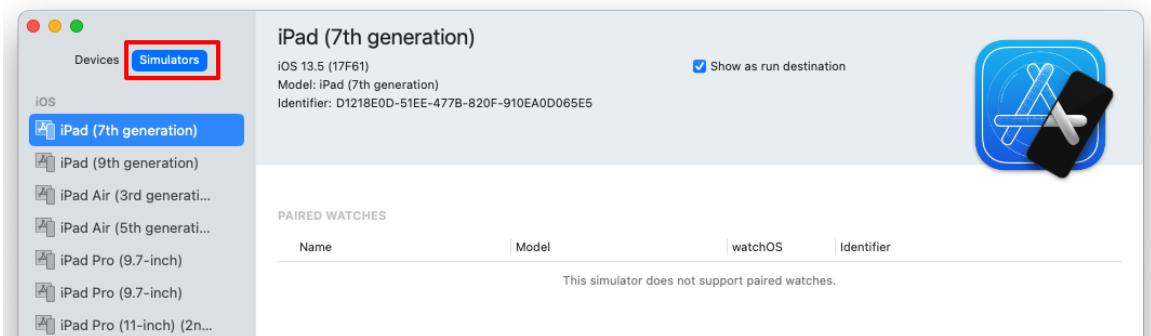
6. In the default simulator, press the **Click me** button several times and observe that the count of the number of button clicks is incremented.



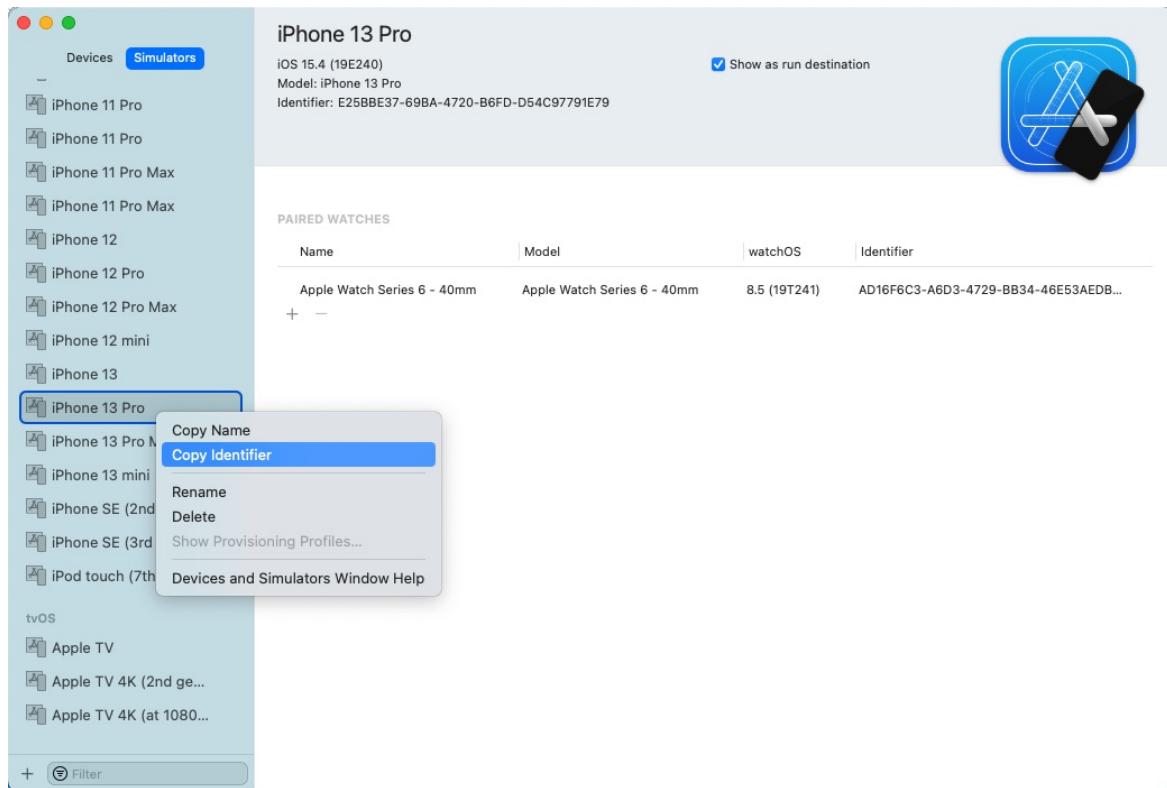
Launch the app on a specific simulator

A .NET MAUI iOS app can be launched on a specific iOS simulator by providing its unique device id (UDID):

1. On your Mac, open **Xcode**, select the **Windows > Devices and Simulators** menu item, and then the **Simulators** tab.



2. Right-click on your chosen simulator, and select **Copy Identifier** to copy the UDID to the clipboard.



Alternatively, you can retrieve a list of UDID values by executing the `simctl list` command:

```
/Applications/Xcode.app/Contents/Developer/usr/bin/simctl list
```

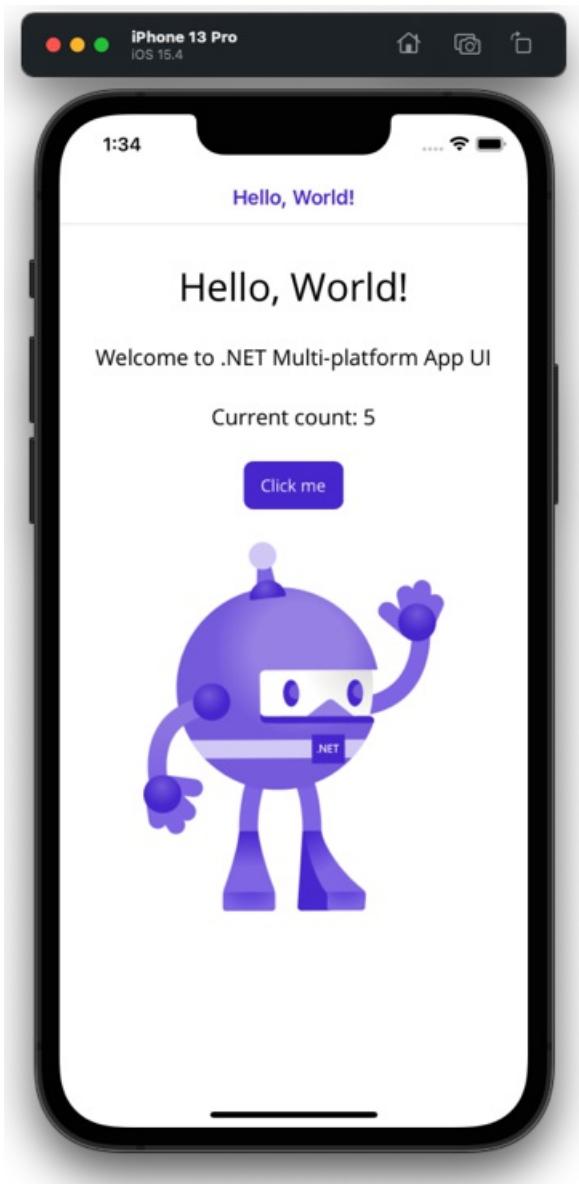
3. In Terminal, build the app and run it on your chosen simulator by specifying the `_DeviceName` MSBuild property using the `-p` `MSBuild` option:

```
dotnet build -t:Run -f net6.0-ios -p:_DeviceName=:v2:udid=insert_UDID_here
```

For example, use the following command to build the app and run it on the iPhone 13 Pro simulator:

```
dotnet build -t:Run -f net6.0-ios -p:_DeviceName=:v2:udid=E25BBE37-69BA-4720-B6FD-D54C97791E79
```

4. In your chosen simulator, press the **Click me** button several times and observe that the count of the number of button clicks is incremented.



Pair to Mac for iOS development

9/20/2022 • 6 minutes to read • [Edit Online](#)

Building native iOS applications using .NET Multi-platform App UI (.NET MAUI) requires access to Apple's build tools, which only run on a Mac. Because of this, Visual Studio 2022 must connect to a network-accessible Mac to build .NET MAUI iOS apps.

Visual Studio 2022's Pair to Mac feature discovers, connects to, authenticates with, and remembers Mac build hosts so that you can work productively on Windows.

Pair to Mac enables the following software development workflow:

- You can write .NET MAUI iOS code in Visual Studio 2022.
- Visual Studio 2022 opens a network connection to a Mac build host and uses the build tools on that machine to compile and sign the iOS app.
- There's no need to run a separate application on the Mac – Visual Studio 2022 invokes Mac builds securely over SSH.
- Visual Studio 2022 is notified of changes as soon as they happen. For example, when an iOS device is plugged into the Mac or becomes available on the network, the iOS Toolbar updates instantly.
- Multiple instances of Visual Studio 2022 can connect to the Mac simultaneously.
- It's possible to use the Windows command-line to build iOS apps.

NOTE

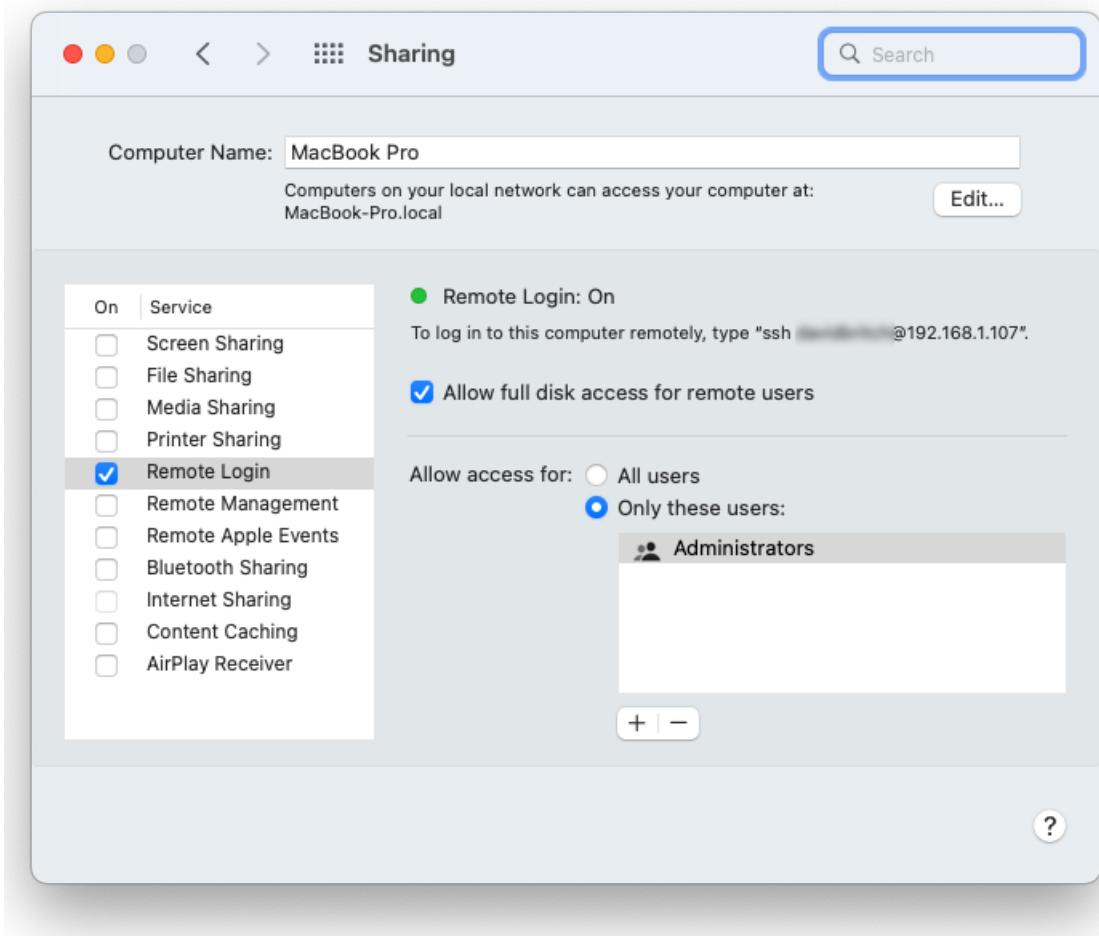
Before following the instructions in this article, on a Mac, [install Xcode](#). Then manually open Xcode, after installation, so that it can add additional components. In addition, you should also install either the latest [Visual Studio 2022 for Mac Preview](#) or [Mono](#).

If you would prefer not to install Visual Studio 2022 for Mac, Visual Studio 2022 can automatically configure the Mac build host. However, you must still install and run Xcode, and install Mono.

Enable remote login on the Mac

To set up the Mac build host, first enable remote login:

1. On the Mac, open **System Preferences** and go to the **Sharing** pane.
2. Check **Remote Login** in the **Service** list.



Make sure that it's configured to allow access for **All users**, or that your Mac username or group is included in the list of allowed users.

3. If prompted, configure the macOS firewall. If you have set the macOS firewall to block incoming connections, you may need to allow `mono-sgen` to receive incoming connections. An alert appears to prompt you if so.
4. If it's on the same network as the Windows machine, the Mac should now be discoverable by Visual Studio 2022. If the Mac is still not discoverable, try [manually adding a Mac](#).

Connect to the Mac from Visual Studio 2022

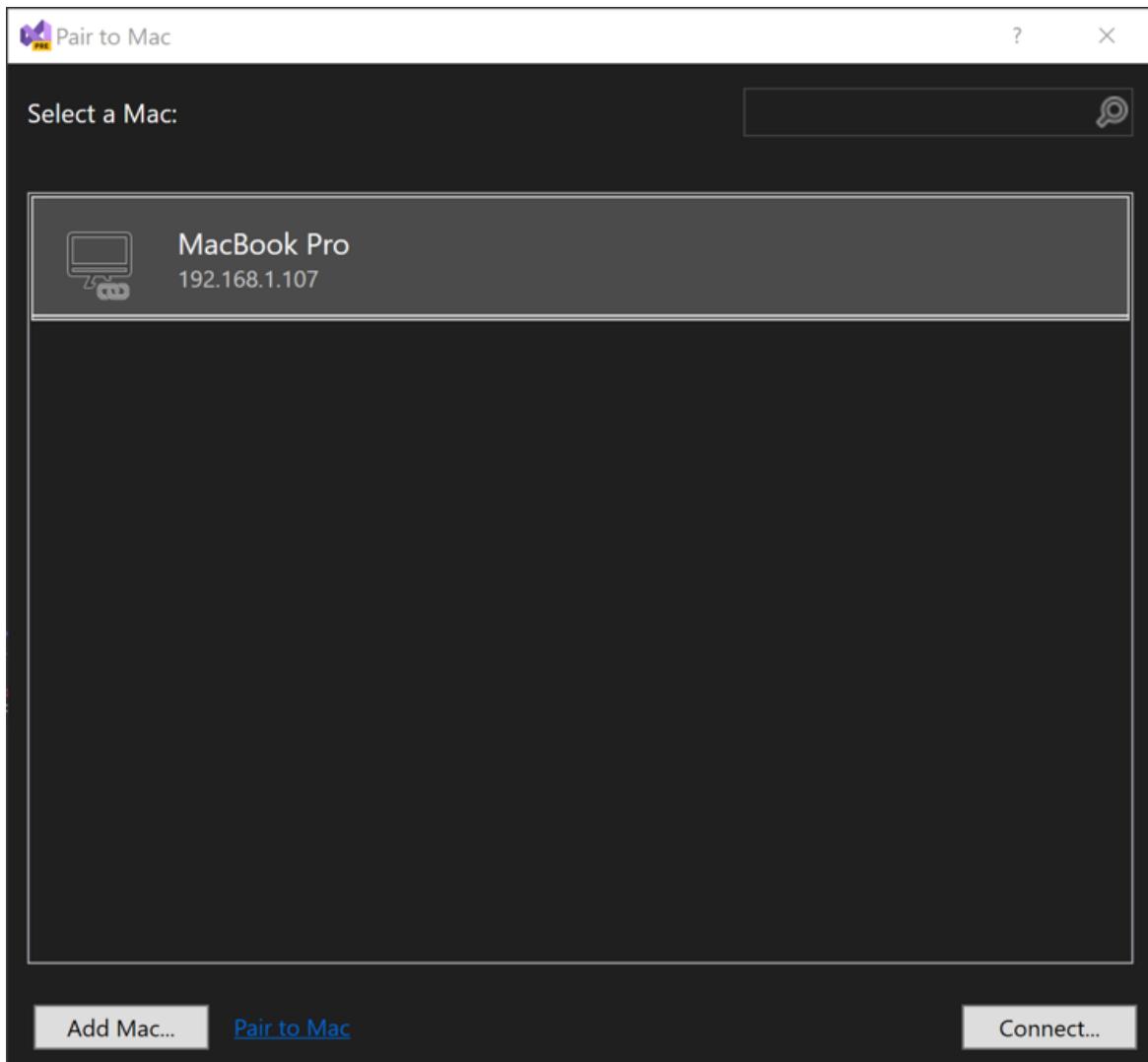
After enabling remote login on the Mac, connect Visual Studio 2022 to the Mac:

1. In Visual Studio 2022, open an existing .NET MAUI project or create a new one.
2. Open the **Pair to Mac** dialog with the **Pair to Mac** button iOS toolbar:

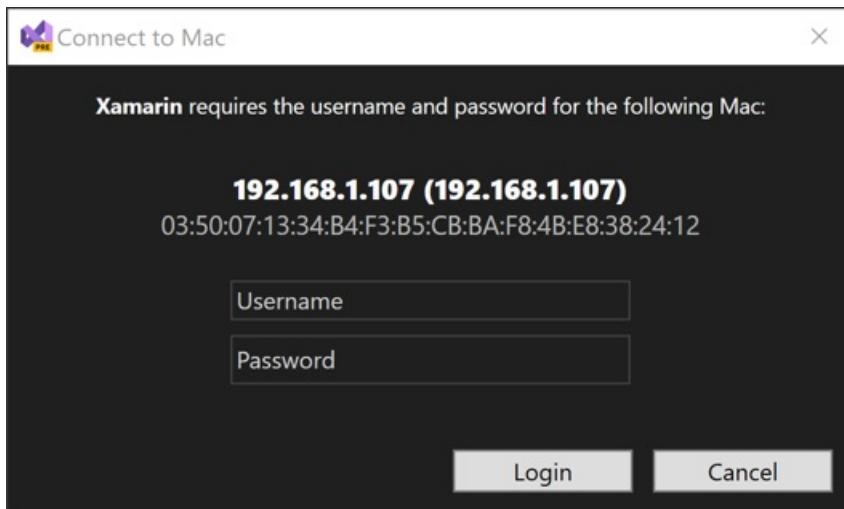


Alternatively, select **Tools > iOS > Pair to Mac**.

The **Pair to Mac** dialog displays a list of all previously connected and currently available Mac build hosts:



3. Select a Mac in the list and select Connect.
4. Enter your username and password. The first time you connect to any particular Mac, you're prompted to enter your username and password for that machine:



TIP

When logging in, use your system username.

Pair to Mac uses these credentials to create a new SSH connection to the Mac. If it succeeds, a key is added to the `authorized_keys` file on the Mac. Subsequent connections to the same Mac will log in

automatically.

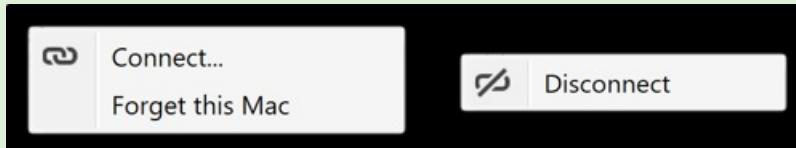
5. Pair to Mac automatically configures the Mac. Visual Studio 2022 installs or updates pre-requisites on a connected Mac build host as needed. However, Xcode must still be installed manually.
6. Examine the connection status icon. When Visual Studio 2022 is connected to a Mac, that Mac's item in the **Pair to Mac** dialog displays an icon indicating that it's currently connected:



There can be only one connected Mac at a time.

TIP

Right-clicking any Mac in the **Pair to Mac** list brings up a context menu that allows you to **Connect...**, **Forget this Mac**, or **Disconnect**:



If you choose **Forget this Mac**, your credentials for the selected Mac will be forgotten. To reconnect to that Mac, you will need to re-enter your username and password.

If you've successfully paired to a Mac build host, you're ready to build .NET MAUI iOS apps in Visual Studio 2022. For more information, see [Build your first app](#).

If you haven't been able to pair a Mac, try [manually adding a Mac](#).

Manually add a Mac

If you don't see a specific Mac listed in the **Pair to Mac** dialog, add it manually:

1. Open **System Preferences > Sharing > Remote Login** on your Mac to locate your Mac's IP address:



Alternatively, use the command line. In **Terminal**, issue the following command:

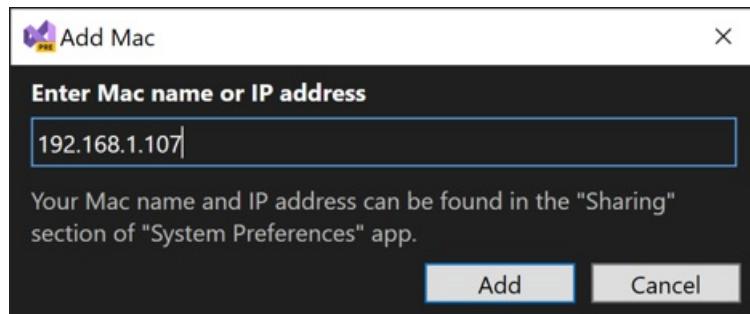
```
ipconfig getifaddr en0
```

Depending on your network configuration, you may need to use an interface name other than `en0`, for example, `en1` or `en2`.

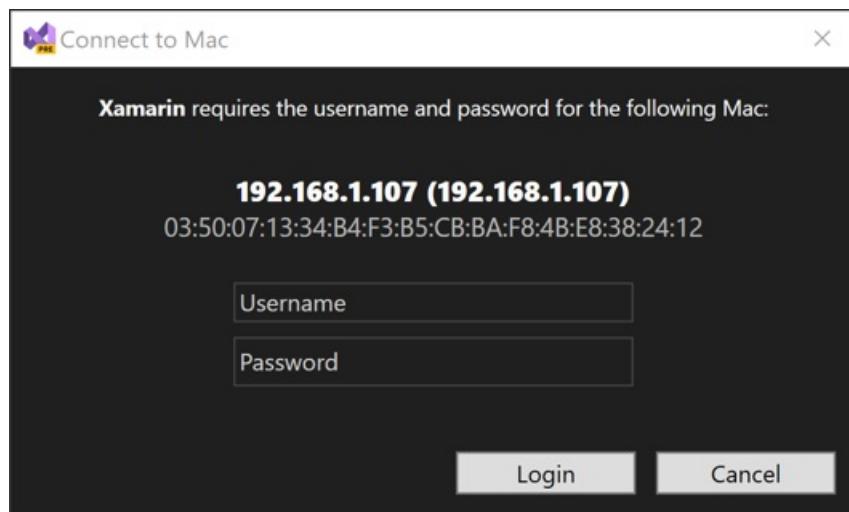
2. In Visual Studio 2022's **Pair to Mac** dialog, select **Add Mac...**:



3. Enter the Mac's IP address and select **Add**:



4. Enter your username and password for the Mac:



TIP

When logging in, use your system username.

5. Select **Login** to connect Visual Studio 2022 to the Mac over SSH and add it to the list of known machines.

Automatic Mac provisioning

Pair to Mac automatically provisions a Mac with the software necessary for building .NET MAUI iOS apps. This includes .NET and various Xcode-related tools (but not Xcode itself).

IMPORTANT

- Pair to Mac can't install Xcode. You must manually install it on the Mac build host. It's required for .NET MAUI iOS development.
- Automatic Mac provisioning requires that remote login is enabled on the Mac, and the Mac must be network-accessible to the Windows machine.
- Automatic Mac provisioning requires sufficient free space on the Mac to install .NET.

In addition, Pair to Mac performs required software installations and updates to the Mac, when Visual Studio 2022 connects to it.

Xcode tools and license

Pair to Mac will also check to determine whether Xcode has been installed and its license accepted. While Pair to Mac doesn't install Xcode, it does prompt for license acceptance.

In addition, Pair to Mac will install or update various packages distributed with Xcode. The installation of these packages happens quickly and without a prompt.

Troubleshooting automatic Mac provisioning

If you encounter any trouble using automatic Mac provisioning, take a look at the Visual Studio 2022 IDE logs, stored in %LOCALAPPDATA%\Xamarin\Logs\17.0. These logs may contain error messages to help you better diagnose the failure or get support.

Build iOS apps from the Windows command-line

Pair to Mac supports building .NET MAUI apps from the command line. Navigate to the folder that holds the source of your .NET MAUI iOS app and execute the following command:

```
dotnet build -f:net6.0-ios /p:ServerAddress={macOS build host IP address} /p:ServerUser={macOS username}  
/p:ServerPassword={macOS password} /p: TcpPort=58181 /p:_DotNetRootRemoteDirectory=/Users/{macOS  
username}/Library/Caches/Xamarin/XMA/SDKs/dotnet/
```

The parameters passed to `dotnet` in the above example are:

- `ServerAddress` – the IP address of the Mac build host.
- `ServerUser` – the username to use when logging in to the Mac build host. Use your system username rather than your full name.
- `ServerPassword` – the password to use when logging in to the Mac build host.
- `_DotNetRootRemoteDirectory` - the folder on the Mac build host that contains the .NET SDK.

The first time Pair to Mac logs in to a Mac build host from either Visual Studio 2022 or the command-line, it sets up SSH keys. With these keys, future logins won't require a username or password. Newly created keys are stored in %LOCALAPPDATA%\Xamarin\MonoTouch.

If the `ServerPassword` parameter is omitted from a command-line build invocation, Pair to Mac attempts to log in to the Mac build host using the saved SSH keys.

Remote iOS Simulator for Windows

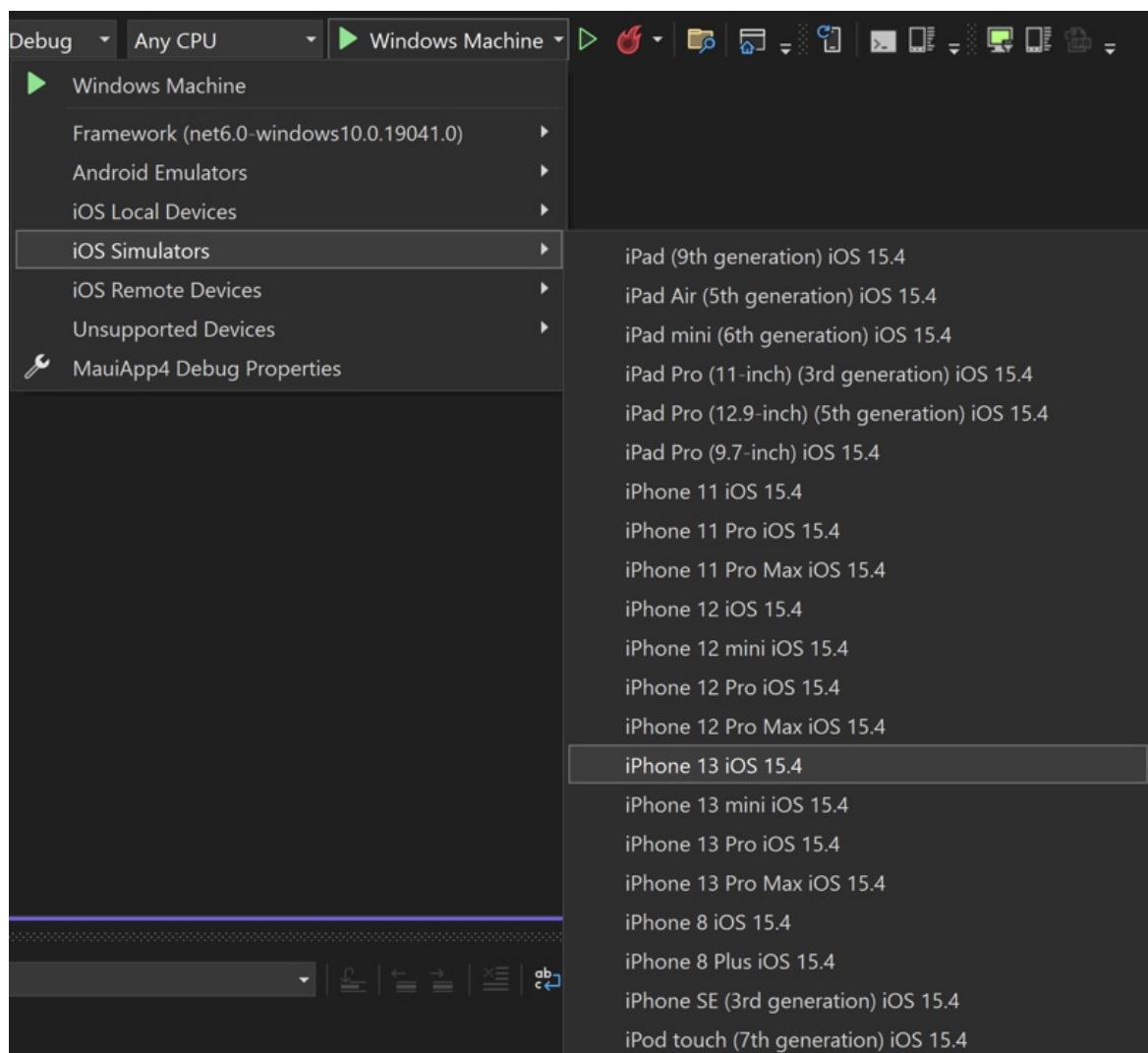
9/20/2022 • 2 minutes to read • [Edit Online](#)

The remote iOS Simulator for Windows allows you to test your apps on an iOS simulator displayed in Windows alongside Visual Studio 2022.

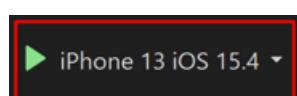
Get started

The remote iOS Simulator for Windows is installed automatically as part of the .NET Multi-platform App UI development workload in Visual Studio 2022. To use it, follow these steps:

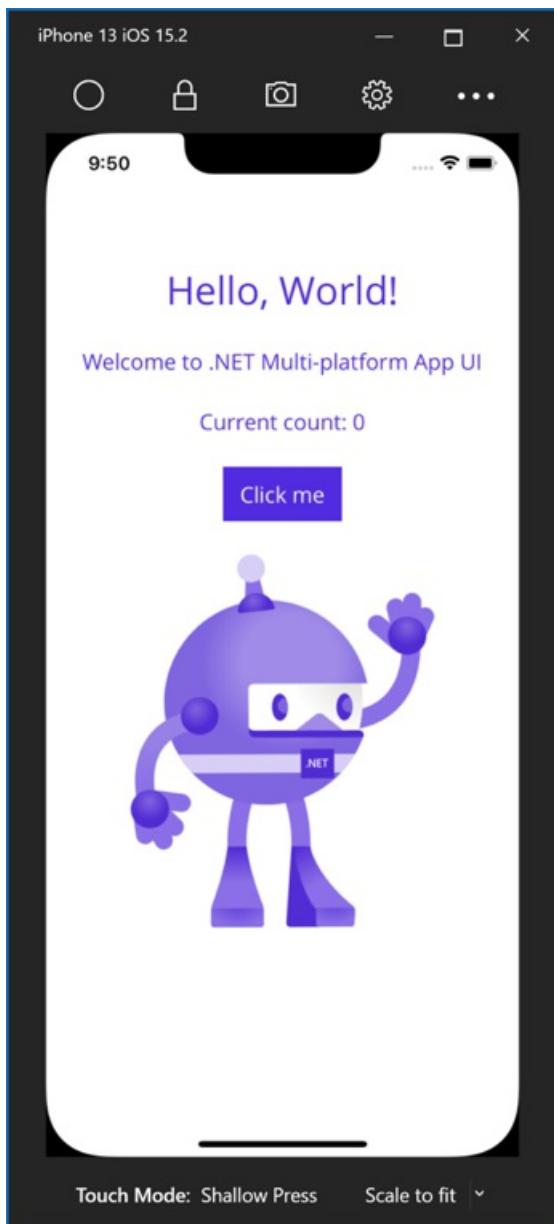
1. Launch Visual Studio 2022 and create or load a .NET MAUI app project.
2. In Visual Studio 2022, pair the IDE to a Mac Build host if you haven't previously. For more information, see [Pair to Mac for iOS development](#).
3. In the Visual Studio toolbar, use the **Debug Target** drop down to select **iOS Simulators** and then a specific iOS simulator:



4. In the Visual Studio toolbar, press the green Start button for your chosen iOS simulator:

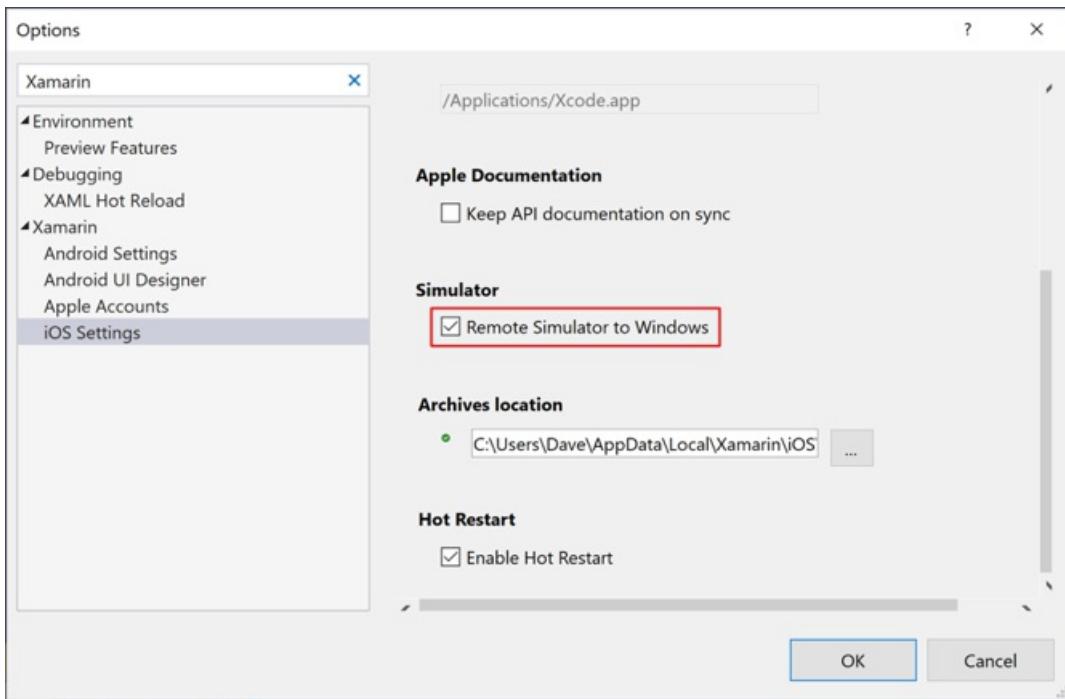


Visual Studio will build the app, start the remote iOS simulator for Windows, and deploy the app to the simulator:



Enable the remote iOS simulator for Windows

The remote iOS simulator for Windows is enabled by default. However, if it's been previously disabled it can be enabled in Visual Studio by navigating to **Tools > Options > Xamarin > iOS Settings** and ensuring that **Remote Simulator to Windows** is checked:



NOTE

When the remote simulator is disabled in Visual Studio, debugging a .NET MAUI iOS app will open the iOS Simulator on the connected Mac build host.

Simulator window toolbar

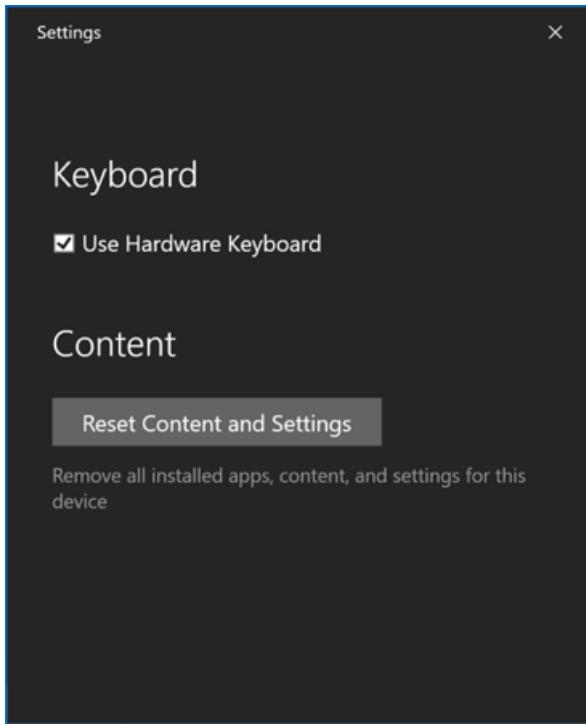
The toolbar at the top of the simulator's window displays five buttons:



The buttons are as follows:

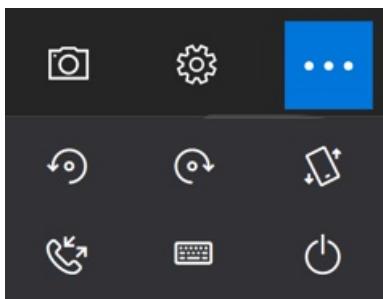
- **Home** – simulates the home button on an iOS device.
- **Lock** – locks the simulator (swipe to unlock).
- **Take Screenshot** – saves a screenshot of the simulator to `\Users\{User}\Pictures\Xamarin\iOS Simulator`.
- **Settings** – displays keyboard and other settings.
- **Other options** – displays various simulator options such as rotation, and shake gesture.

Clicking the toolbar's **Settings** button (the gear icon) opens the **Settings** window:



These settings allow you to enable the hardware keyboard and reset the content and settings for the simulator.

Clicking the toolbar's **Other options** button (the ellipsis icon) reveals additional buttons such as rotation, shake gestures, and rebooting:



NOTE

Right-clicking anywhere in the remote iOS simulator window will display all the toolbar buttons as a context menu.

Touchscreen support

Many Windows computers have touch screens. Since the remote iOS Simulator for Windows supports touch interactions, you can test your app with the same pinch, swipe, and multi-finger touch gestures that you use with physical iOS devices.

Similarly, the remote iOS Simulator for Windows treats Windows Stylus input as Apple Pencil input.

Sound handling

Sounds played by the simulator will come from the host Mac's speakers. iOS sounds are not heard on Windows.

Troubleshooting

In some circumstances, an Xcode configuration problem can result in the remote iOS Simulator for Windows getting stuck in a Connecting to Mac...Checking Server...Connected... loop. When this occurs, you need to remove and reset the Simulators on your Mac build host:

- Ensure that Xamarin Mac Agent (XMA) and Xcode aren't running.
- Delete your `~/Library/Developer/CoreSimulator/Devices` folder.
- Run `killall -9 com.apple.CoreSimulator.CoreSimulatorService`.
- Run `xcrun simctl list devices`.

Logs

If you experience issues with the remote iOS Simulator, you can view the logs in the following locations:

- **Mac** – `~/Library/Logs/Xamarin/Simulator.Server`
- **Windows** – `%LOCALAPPDATA%\Xamarin\Logs\Xamarin.Simulator`

Build a Mac Catalyst app with .NET CLI

9/20/2022 • 2 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to create and run a .NET Multi-platform App UI (.NET MAUI) app on Mac Catalyst using .NET Command Line Interface (CLI) on macOS:

1. To create .NET MAUI apps, you'll need to download and run the [installer](#) for the latest .NET 6 runtime. You'll also need to download and install the latest version of [Xcode 13](#), which is also available from the App Store app on your Mac.
2. On your Mac, open **Terminal** and check that you have the latest .NET 6 runtime installed:

```
dotnet --version
```

3. In **Terminal**, install the latest public build of .NET MAUI:

```
sudo dotnet workload install maui --source https://api.nuget.org/v3/index.json
```

This command will install the latest released version of .NET MAUI, including the required platform SDKs.

4. In **Terminal**, create a new .NET MAUI app using .NET CLI:

```
dotnet new maui -n "MyMauiApp"
```

5. In **Terminal**, change directory to *MyMauiApp*, and build and run the app:

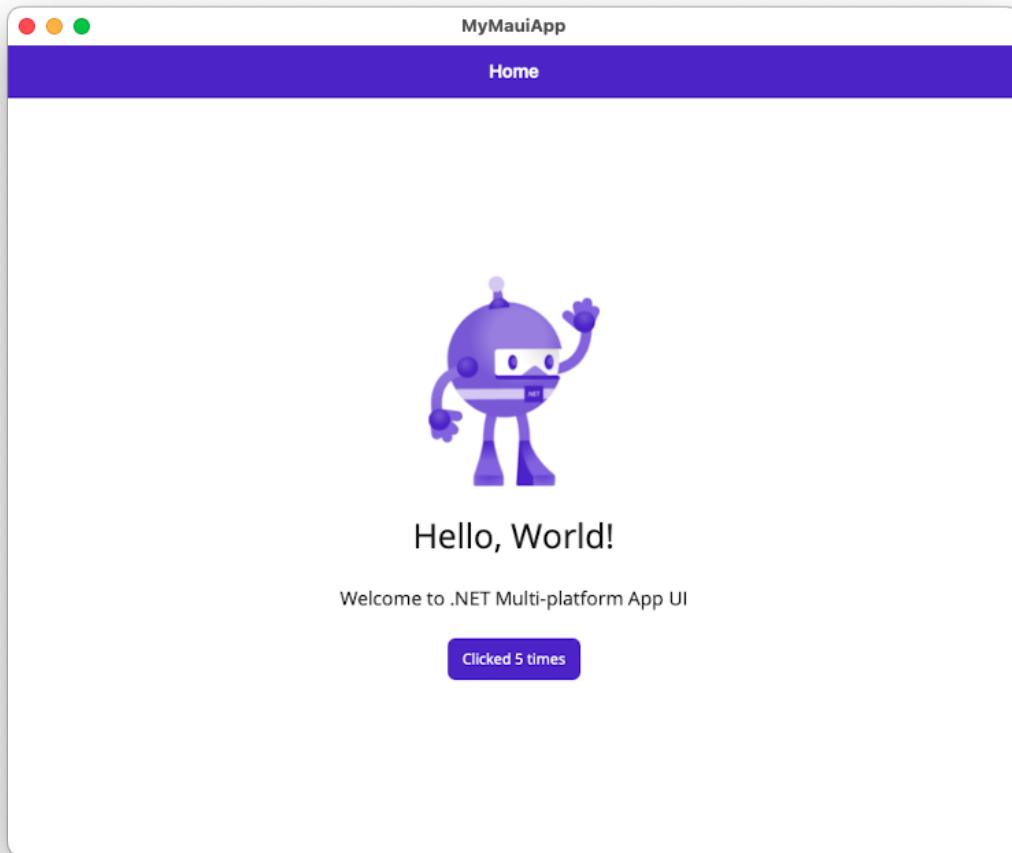
```
cd MyMauiApp  
dotnet build -t:Run -f net6.0-maccatalyst
```

The `dotnet build` command will restore the project dependencies, build the app, and launch it.

If you see a build error and a warning that the Xcode app bundle could not be found, you may need to run the following command:

```
xcode-select --reset
```

6. In the running app, press the **Click me** button several times and observe that the count of the number of button clicks is incremented.



Deploy and debug your .NET MAUI app on Windows

9/20/2022 • 2 minutes to read • [Edit Online](#)

You can use your local Windows development computer to deploy and debug a .NET Multi-platform App UI (.NET MAUI) app. This article describes how to configure Windows to debug a .NET MAUI app.

Configure Windows

You must enable Developer Mode in Windows. Both Windows 10 and Windows 11 are supported.

Developer Mode

Install apps from any source, including loose files.



On

Windows 11

Developer Mode is enabled in **Settings** app, under **Privacy & security > For developers**. To enable Developer Mode in Windows 11:

1. Open the Start Menu.
2. Type **Developer settings** in the search box and select it.
3. Turn on **Developer Mode**.
4. If you receive a warning message about Developer Mode, read it, and select **Yes** if you understand the warning.

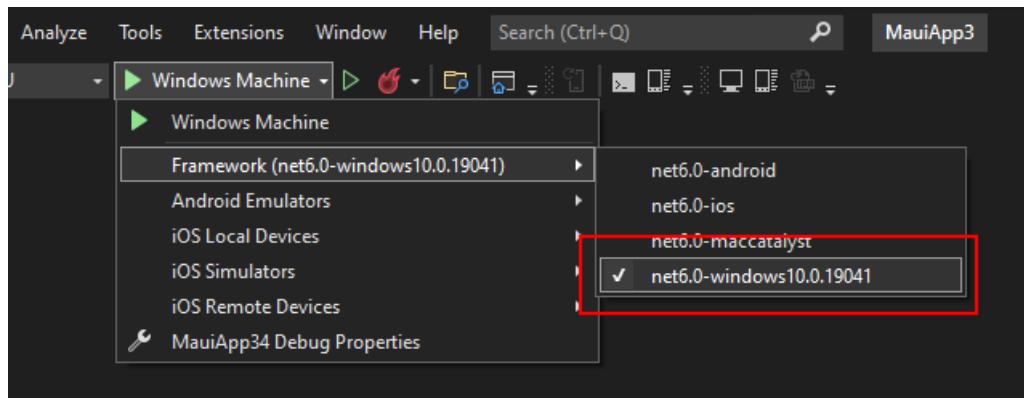
Windows 10

Developer Mode is enabled in **Settings** app, under **Update & Security > For developers**. To enable Developer Mode in Windows 10:

1. Open the Start Menu.
2. Search for **Developer settings**, select it.
3. Turn on **Developer Mode**.
4. If you receive a warning message about Developer Mode, read it, and select **Yes** if you understand the warning.

Target Windows

In Visual Studio, set the **Debug Target** to **Framework (...) > net6.0-windows**. There is a version number in the item entry, which may or may not match the following screenshot:



XAML

9/20/2022 • 2 minutes to read • [Edit Online](#)

The eXtensible Application Markup Language (XAML) is an XML-based language that's an alternative to programming code for instantiating and initializing objects, and organizing those objects in parent-child hierarchies.

XAML allows developers to define user interfaces in .NET Multi-platform App UI (.NET MAUI) apps using markup rather than code. XAML is not required in a .NET MAUI app, but it is the recommended approach to developing your UI because it's often more succinct, more visually coherent, and has tooling support. XAML is also well suited for use with the Model-View-ViewModel (MVVM) pattern, where XAML defines the view that is linked to viewmodel code through XAML-based data bindings.

Within a XAML file, you can define user interfaces using all the .NET MAUI views, layouts, and pages, as well as custom classes. The XAML file can be either compiled or embedded in the app package. Either way, the XAML is parsed at build time to locate named objects, and at runtime the objects represented by the XAML are instantiated and initialized.

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.

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.
- 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, although this restriction can sometimes be overcome.
- XAML generally cannot call methods, although this restriction can sometimes be overcome.

There is no visual designer for producing XAML in .NET MAUI apps. All XAML must be hand-written, but you can use XAML hot reload to view your UI as you edit it.

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.

Get started with XAML

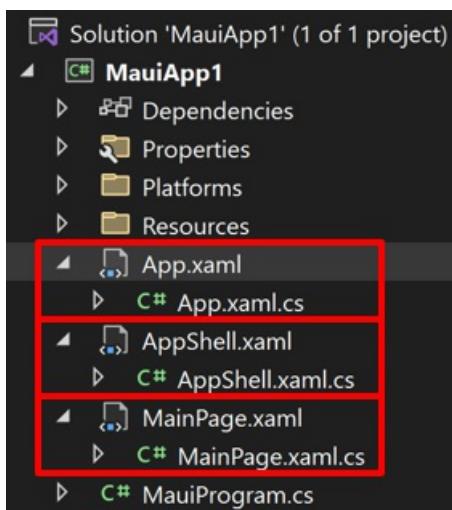
9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

In a .NET Multi-platform App UI (.NET MAUI) app, 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.

Anatomy of a XAML file

A new .NET MAUI app contains three XAML files, and their associated code-behind files:



The first file pairing is *App.xaml*, a XAML file, and *App.xaml.cs*, a C# *code-behind* file associated with the XAML file. Both *App.xaml* and *App.xaml.cs* contribute to a class named `App` that derives from `Application`. The second file pairing is *AppShell.xaml* and *AppShell.xaml.cs*, which contribute to a class named `AppShell` that derives from `Shell`. Most other classes with XAML files contribute to a class that derives from `ContentPage`, and define the UI of a page. This is true of the *MainPage.xaml* and *MainPage.xaml.cs* files.

The *MainPage.xaml* file has the following structure:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MyMauiApp.MainPage">
    ...
</ContentPage>
```

The two XML namespace (`xmlns`) declarations refer to URIs on microsoft.com. However, there's no content at these URLs, 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 .NET MAUI, 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. .NET MAUI supports the [2009 XAML specification](#).

At the end of the 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 `MyMauiApp` namespace. This means that this XAML file defines a new class named `MainPage` in the `MyMauiApp` 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 a XAML file is instead simply instantiated from existing classes and initialized.

The `MainPage.xaml.cs` file looks similar to this:

```
namespace MyMauiApp;

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

The `MainPage` class derives from `ContentPage`, and is a partial class definition.

When Visual Studio builds the project, a source generator generates new C# source that contains the definition of the `InitializeComponent` method that's called from the `MainPage` constructor and adds it to the compilation object.

At runtime, code in the `MauiProgram` class bootstraps the app and executes the `App` class constructor, which instantiates `AppShell`. The `AppShell` class instantiates the first page of the app to be displayed, which is `MainPage`. The `MainPage` constructor calls `InitializeComponent`, which 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.

NOTE

The `AppShell` class uses .NET MAUI Shell to set the first page of the app to be displayed. However, Shell is beyond the scope of this introduction to XAML. For more information, see [.NET MAUI Shell](#).

Set page content

A `ContentPage` should contain a single child, that can be a view or a layout with child views. The child of the `ContentPage` is automatically set as the value of the `ContentPage.Content` property.

The following example shows a `ContentPage` containing a `Label`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.HelloXamlPage"
    Title="Hello XAML Page">
    <Label Text="Hello, XAML!">
        VerticalOptions="Center"
        HorizontalTextAlignment="Center"
        Rotation="-15"
        FontSize="18"
        FontAttributes="Bold"
        TextColor="Blue" />
</ContentPage>

```

From the example above the relationship between classes, properties, and XML should be evident. A .NET MAUI 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`, and `Rotation` is of type `double`. 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 .NET MAUI that derive from `TypeConverter`. For the example above, several .NET MAUI converters are automatically applied to convert string values to their correct type:

- `LayoutOptionsConverter` for the `VerticalOptions` property. This converter converts the names of public static fields of the `LayoutOptions` structure to values of type `LayoutOptions`.
- `ColorTypeConverter` for the `TextColor` property. This converter converts the names of public static fields of the `Colors` structure or hexadecimal RGB values, with or without an alpha channel.

Page navigation

When you run a .NET MAUI app, the `MainPage` is typically displayed. To see a different page you can either set that as the new startup page in the `AppShell.xaml` file, or navigate to the new page from `MainPage`.

To implement navigation, 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;
}

```

When you compile and deploy the new version of this app, a button appears on the screen. Pressing it navigates

to `HelloXamlPage`:

Hello, XAML!

You can navigate back to `MainPage` using the navigation bar that appears on each platform.

NOTE

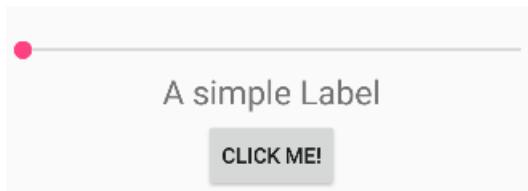
An alternative to this navigation model is to use .NET MAUI Shell. For more information, see [.NET MAUI Shell overview](#).

XAML and code interactions

The child of most `ContentPage` derivatives is a layout, such as a `StackLayout` or a `Grid`, and the layout can contain multiple children. In XAML, these parent-child relationships are established with normal XML hierarchy:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.XamlPlusCodePage"
    Title="XAML + Code Page">
    <StackLayout>
        <Slider VerticalOptions="Center" />
        <Label Text="A simple Label"
            FontSize="18"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <Button Text="Click Me!"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
    </StackLayout>
</ContentPage>
```

This XAML file is syntactically complete, and produces the following UI:



However, while you can interact with the `Slider` and `Button`, the UI isn't updated. The `Slider` should cause the `Label` to display the current value, and the `Button` should do something.

Displaying a `Slider` value using a `Label` can be achieved entirely in XAML with a *data binding*. However, it's 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`:

```

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

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

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

```

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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.XamlPlusCodePage"
    Title="XAML + Code Page">
    <StackLayout>
        <Slider VerticalOptions="Center"
            ValueChanged="OnSliderValueChanged" />
        <Label x:Name="valueLabel"
            Text="A simple Label"
            FontSize="18"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <Button Text="Click Me!"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            Clicked="OnButtonClicked" />
    </StackLayout>
</ContentPage>

```

Notice that assigning a handler to an event has the same syntax as assigning a value to a property. In addition, for the `ValueChanged` event handler of the `Slider` to use the `Label` to display the current value, the handler needs to reference that object from code. Therefore, the `Label` needs a name, which is specified with the `x:Name` attribute. 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.

The `ValueChanged` event handler can now set the `Label` to display the new `Slider` value, which is available from the event arguments:

```

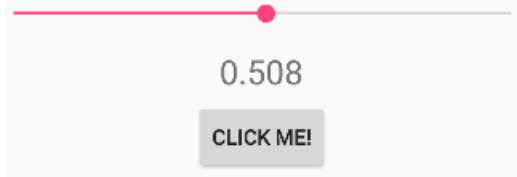
void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
    valueLabel.Text = args.NewValue.ToString("F3");
}

```

Alternatively, 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");
}
```

The result is that any manipulation of the `Slider` causes its value to be displayed in the `Label`:



In the example above the `Button` simulates a response to a `Clicked` event by displaying an alert with the `Text` of the button. Therefore, the event handler can 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 `OnButtonClicked` 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.

Next steps

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*.

[Essential XAML syntax](#)

Essential XAML syntax

9/20/2022 • 5 minutes to read • [Edit Online](#)

 [Browse 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 .NET Multi-platform App UI (.NET MAUI) XAML, properties of classes are normally set as XML attributes:

```
<Label Text="Hello, XAML!"  
      VerticalOptions="Center"  
      FontAttributes="Bold"  
      FontSize="18"  
      TextColor="Aqua" />
```

However, there is an alternative way to set a property in XAML:

```
<Label Text="Hello, XAML!"  
      VerticalOptions="Center"  
      FontAttributes="Bold"  
      FontSize="18">  
    <Label.TextColor>  
      Aqua  
    </Label.TextColor>  
</Label>
```

These two examples that specify the `TextColor` property are functionally equivalent, and enable the introduction of some basic terminology:

- `Label` is an *object element*. It is a .NET MAUI object expressed as an XML element.
- `Text`, `VerticalOptions`, `FontAttributes` and `FontSize` are *property attributes*. They are .NET MAUI properties expressed as XML attributes.
- In the second example, `TextColor` has become a *property element*. It is a .NET MAUI property expressed as an XML element.

NOTE

In a property element, the value of the property is always defined as the content between the property-element start and end tags.

Property-element syntax can also be used on more than one property of an object:

```

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

```

While property-element syntax might seem unnecessary, it's 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, the `Grid` layout has properties named `RowDefinitions` and `ColumnDefinitions`, which are of type `RowDefinitionCollection` and `ColumnDefinitionCollection` respectively. These types are collections of `RowDefinition` and `ColumnDefinition` objects, and you typically use property element syntax to set them:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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>

```

Attached properties

In the previous example you saw that the `Grid` requires property elements for the `RowDefinitions` and `ColumnDefinitions` collections to define the rows and columns. This suggests that there must be a technique for indicating 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 `Grid.Row` and `Grid.Column` attributes, which have default values of 0. You can also indicate if a child spans more than one row or column with the `Grid.RowSpan` and `Grid.ColumnSpan` attributes, which have default values of 1.

The following example demonstrates placing children within a `Grid`:

```

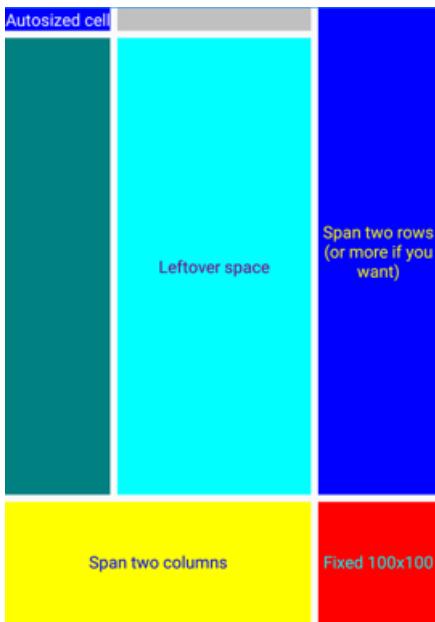
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.GridDemoPage"
    Title="Grid Demo Page">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="100" />
        </Grid.ColumnDefinitions>

        <Label Text="Autosized cell"
            TextColor="White"
            BackgroundColor="Blue" />
        <BoxView Color="Silver"
            Grid.Column="1" />
        <BoxView Color="Teal"
            Grid.Row="1" />
        <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.Column="2" Grid.RowSpan="2"
            TextColor="Yellow"
            BackgroundColor="Blue"
            HorizontalTextAlignment="Center"
            VerticalTextAlignment="Center" />
        <Label Text="Span two columns"
            Grid.Row="2" 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>

```

This XAML results in the following layout:



The `Grid.Row`, `Grid.Column`, `Grid.RowSpan`, and `Grid.ColumnSpan` attributes appear to be properties of the `Grid` class, but this class doesn't define anything named `Row`, `Column`, `RowSpan`, or `ColumnSpan`. Instead, the `Grid` class defines four bindable properties named `RowProperty`, `ColumnProperty`, `RowSpanProperty`, and `ColumnSpanProperty`, that are special types of bindable properties known as *attached properties*. They are defined by the `Grid` class but set on children of the `Grid`.

NOTE

When you wish to use these attached properties in code, the `Grid` class provides static methods named `GetRow`, `SetRow`, `GetColumn`, `SetColumn`, `GetRowSpan`, `SetRowSpan`, `GetColumnSpan`, and `SetColumnSpan`.

Attached properties are recognizable in XAML 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.

Content properties

In the previous example, the `Grid` object was set to the `Content` property of the `ContentPage`. However, the `Content` property wasn't referenced in the XAML but can be:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.XamlPlusCodePage"
    Title="XAML + Code Page">
    <ContentPage.Content>
        <Grid>
            ...
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

The `Content` property isn't required in XAML because elements defined for use in .NET MAUI XAML are allowed to have one property specified as the `ContentProperty` attribute on the class:

```
[ContentProperty("Content")]
public class ContentPage : TemplatedPage
{
    ...
}
```

Any property specified as the `ContentProperty` of a class means that the property-element tags for the property are not required. Therefore, the example above specifies that any XAML content that appears between the start and end `ContentPage` tags is assigned to the `content` property.

Many classes also have `ContentProperty` attribute definitions. For example, the content property of `Label` is `Text`.

Platform differences

.NET MAUI apps can customize UI appearance on a per-platform basis. This can be achieved in XAML using the `OnPlatform` and `On` classes:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="10, 20, 20, 10" />
        </OnPlatform>
    </ContentPage.Padding>
    ...
</ContentPage>
```

`OnPlatform` is a generic class and so you need to specify the generic type argument, in this case, `Thickness`, which is the type of `Padding` property. This is achieved with the `x:TypeArguments` XAML attribute. The `OnPlatform` class has a property named `Platforms`, that is an `IList` of `On` objects. Each `On` object can set the `Platform` and `Value` property to define the `Thickness` value for a specific platform.

In addition, 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="...">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS, Android" Value="10, 20, 20, 10" />
        </OnPlatform>
    </ContentPage.Padding>
    ...
</ContentPage>
```

This is the standard way to set a platform-dependent `Padding` property in XAML.

NOTE

If the `Value` property of an `On` object can't be represented by a single string, you can define property elements for it.

For more information, see [OnPlatform Markup Extension](#).

Next steps

.NET MAUI XAML markup extensions enable 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 app.

[XAML markup extensions](#)

XAML markup extensions

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) XAML markup extensions enable 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 app, but they find their greatest utility in data bindings.

Typically, 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.

XAML markup extensions are so named because they are backed by code in classes that implement `IMarkupExtension`. It's also possible to write your own custom markup extensions.

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

IMPORTANT

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.

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://schemas.microsoft.com/dotnet/2021/maui"
    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="Center" 
            BorderWidth="3" 
            Rotation="-15" 
            TextColor="Red" 
            FontSize="24" />
        <Button Text="Do that!" 
            HorizontalOptions="Center" 
            VerticalOptions="Center" 
            BorderWidth="3" 
            Rotation="-15" 
            TextColor="Red" 
            FontSize="24" />
        <Button Text="Do the other thing!" 
            HorizontalOptions="Center" 
            VerticalOptions="Center" 
            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 at the top of the page, and add resources within these tags. 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:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SharedResourcesPage"
    Title="Shared Resources Page">
    <ContentPage.Resources>
        <LayoutOptions x:Key="horzOptions" 
            Alignment="Center" />
        <LayoutOptions x:Key="vertOptions" 
            Alignment="Center" />
    </ContentPage.Resources>
    ...
</ContentPage>

```

In this example, the two resources 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.

NOTE

Optional `ResourceDictionary` tags can be included as the child of the `Resources` tags.

The resources can then be consumed by the `Button` objects, by using the `StaticResource` XAML markup extension to set their `HorizontalOptions` and `VerticalOptions` properties:

```
<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 .NET MAUI also supports.

`DynamicResource` is for dictionary keys associated with values that might change at runtime, while `StaticResource` accesses elements from the dictionary just once when the elements on the page are constructed. 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.

It's necessary to store doubles in the dictionary for the `BorderWidth`, `Rotation`, and `FontSize` properties. XAML conveniently defines tags for common data types like `x:Double` and `x:Int32`:

```
<ContentPage.Resources>  
    <LayoutOptions x:Key="horzOptions"  
                  Alignment="Center" />  
    <LayoutOptions x:Key="vertOptions"  
                  Alignment="Center" />  
    <x:Double x:Key="borderWidth">3</x:Double>  
    <x:Double x:Key="rotationAngle">-15</x:Double>  
    <x:Double x:Key="fontSize">24</x:Double>  
</ContentPage.Resources>
```

These additional three resources can be referenced in the same way as the `LayoutOptions` values:

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

For resources of type `Color`, you can use the same string representations that you use when directly assigning attributes of these types. Type converters included in .NET MAUI are invoked when the resource is created. It's also possible to use the `OnPlatform` class within the resource dictionary to define different values for the platforms. The following example uses this class for setting different text colors:

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

The `OnPlatform` resource gets 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`, and `Android` attributes are converted to `color` values when the object is initialized.

The following example shows the three buttons accessing six shared values:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SharedResourcesPage"
    Title="Shared Resources Page">
    <ContentPage.Resources>
        <LayoutOptions x:Key="horzOptions"
            Alignment="Center" />
        <LayoutOptions x:Key="vertOptions"
            Alignment="Center" />
        <x:Double x:Key="borderWidth">3</x:Double>
        <x:Double x:Key="rotationAngle">-15</x:Double>
        <x:Double x:Key="fontSize">24</x:Double>
        <OnPlatform x:Key="textColor"
            x:TypeArguments="Color">
            <On Platform="iOS" Value="Red" />
            <On Platform="Android" Value="Aqua" />
            <On Platform="WinUI" Value="#80FF80" />
        </OnPlatform>
    </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 following screenshot verifies the consistent styling:



Although it's common to define the `Resources` collection at the top of the page, you can have `Resources` collections on other elements on the page. For example, the following example shows resources added to a `StackLayout`:

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

One of the most common types of objects stored in resource dictionaries is the .NET MAUI `Style`, which defines a collection of property settings. For more information about styles, see [Style apps using XAML](#).

NOTE

The purpose of a resource dictionary is to share objects. Therefore, it doesn't make sense to put controls such as a `Label` or `Button` in a resource dictionary. Visual elements can't be shared because the same instance can't appear twice on a page.

x:Static Markup Extension

In addition to the `StaticResource` markup extension, there's also an `x:Static` markup extension. However, while `StaticResource` returns an object from a resource dictionary, `x:Static` accesses a public static field, a public static property, a public constant field, or an enumeration member.

NOTE

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.

The following example demonstrates 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 Colors.Aqua}" />
```

The main use of the `x:static` markup extension is in referencing 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 app:

```
namespace XamlSamples  
{  
    static class AppConstants  
    {  
        public static readonly Color BackgroundColor = Colors.Aqua;  
        public static readonly Color ForegroundColor = Colors.Brown;  
    }  
}
```

To reference the static fields of this class in a XAML file, you need to use an XML namespace declaration to indicate where this file is located. Each additional XML namespace declaration defines a new prefix. To access classes local to the root app namespace, such as `AppConstants`, you could 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. 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"
```

NOTE

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.

The static fields can then be consumed after declaring the XML namespace:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="5,25,5,0">
    <StackLayout>
        <Label Text="Hello, XAML!">
            TextColor="{x:Static local:AppConstants.BackgroundColor}"
            BackgroundColor="{x:Static local:AppConstants.ForegroundColor}"
            FontAttributes="Bold"
            FontSize="30"
            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>

```

In this example, the `BoxView` dimensions are set to `Math.PI` and `Math.E`, but scaled by a factor of 100:



Other markup extensions

Several markup extensions are intrinsic to XAML and supported in .NET MAUI XAML. 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.

For more information about XAML markup extensions, see [Consume XAML markup extensions](#).

Next steps

.NET MAUI data bindings allow properties of two objects to be linked so that a change in one causes a change in

the other.

[Data binding basics](#)

Data binding basics

9/20/2022 • 10 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) 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.

Data bindings

Data bindings connect properties of two objects, called the *source* and the *target*. In code, two steps are required:

1. The `BindingContext` property of the target object must be set to the source object,
2. 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`. A property of `Label`, such as `Text`, is associated with the bindable property `TextProperty`.

In XAML, 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.

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.

The following example 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlSamples.SliderBindingsPage"
    Title="Slider Bindings Page">
    <StackLayout>
        <Label Text="ROTATION"
            BindingContext="{x:Reference slider}"
            Rotation="{Binding Path=Value}"
            FontAttributes="Bold"
            FontSize="18"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <Slider x:Name="slider"
            Maximum="360"
            VerticalOptions="Center" />
        <Label BindingContext="{x:Reference slider}"
            Text="{Binding Value, StringFormat='The angle is {0:F0} degrees'}"
            FontAttributes="Bold"
            FontSize="18"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
    </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.

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 second `Binding` markup extension sets the `StringFormat` property. In .NET MAUI, 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`.

IMPORTANT

Formatting strings must be placed in single quotation marks.

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 example demonstrates one common use of the `OneWayToSource` and `TwoWay` binding modes:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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" />

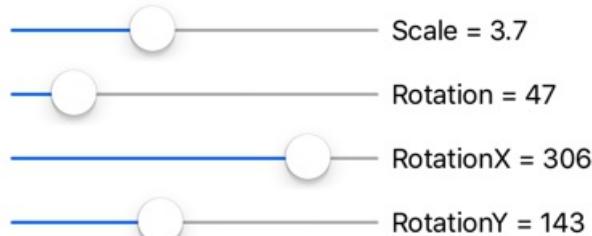
    <!-- Slider and identifying Label for RotationY -->
    <Slider x:Name="rotationYSlider"
        BindingContext="{x:Reference label}"
        Grid.Row="4" Grid.Column="0"
        Maximum="360"
        Value="{Binding RotationY, Mode=OneWayToSource}" />
    <Label BindingContext="{x:Reference rotationYSlider}"
        Text="{Binding Value, StringFormat='RotationY = {0:F0}'}"
        Grid.Row="4" Grid.Column="1"
        VerticalTextAlignment="Center" />
</Grid>

```

```
</ContentPage>
```

In this example, 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, 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`.

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

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

`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. Custom cells can

be created for items using the `ViewCell` class.

`ListView` can display a list of every named color that's available in .NET MAUI, with the help of the `NamedColor` class:

```

using System.Reflection;
using System.Text;

namespace XamlSamples
{
    public class NamedColor
    {
        public string Name { get; private set; }
        public string FriendlyName { get; private set; }
        public Color Color { get; private set; }

        // Expose the Color fields as properties
        public float Red => Color.Red;
        public float Green => Color.Green;
        public float Blue => Color.Blue;

        public static IEnumerable<NamedColor> All { get; private set; }

        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.
                    NamedColor namedColor = new NamedColor
                    {
                        Name = name,
                        FriendlyName = stringBuilder.ToString(),
                        Color = (Color)fieldInfo.GetValue(null)
                    };

                    // Add it to the collection.
                    all.Add(namedColor);
                }
            }
            all.TrimExcess();
            All = all;
        }
    }
}

```

Each `NamedColor` object has `Name` and `FriendlyName` properties of type `string`, a `Color` property of type `Color`, and `Red`, `Green`, and `Blue` properties. In addition, the `NamedColor` static constructor creates an `IEnumerable<NamedColor>` collection that contains `NamedColor` objects corresponding to the fields of type `Color`.

in the `Colors` class, and assigns it to its public static `All` property.

Setting the static `NamedColor.All` property to the `ItemsSource` of a `ListView` can be achieved using the `x:Static` markup extension:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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 result establishes that the items are of type `XamlSamples.NamedColor`:

To define a template for the items, the `ItemTemplate` should be set to a `DataTemplate` that references a `ViewCell`. The `ViewCell` should define a layout of one or more views to display each item:

```
<ListView ItemsSource="{x:Static local:NamedColor.All}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <Label Text="{Binding FriendlyName}" />
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

NOTE

The binding source for cells, and children of cells, is the `ListView.ItemsSource` collection.

In this example, 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 XAML displays the `FriendlyName` property of each `NamedColor` object:

```
...  
Pale Goldenrod  
Pale Green  
Pale Turquoise  
Pale Violet Red  
Papaya Whip  
Peach Puff  
Peru  
Pink  
Plum  
Powder Blue  
Purple  
Red  
Rosy Brown  
Royal Blue  
Saddle Brown  
Salmon  
Sandy Brown  
Sea Green  
Sea Shell  
Sienna  
Silver  
Sky Blue  
Slate Blue  
Slate Gray  
Snow...
```

The item template can be expanded to display more information and the actual color:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples"
    x:Class="XamlSamples.ListViewDemoPage"
    Title="ListView Demo Page">
<ContentPage.Resources>
    <x:Double x:Key="boxSize">50</x:Double>
    <x:Int32 x:Key="rowHeight">60</x:Int32>
    <local:FloatToIntConverter x:Key="intConverter" />
</ContentPage.Resources>

<ListView ItemsSource="{x:Static local:NamedColor.All}"
    RowHeight="{StaticResource rowHeight}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <StackLayout Padding="5, 5, 0, 5"
                    Orientation="Horizontal"
                    Spacing="15">
                    <BoxView WidthRequest="{StaticResource boxSize}"
                        HeightRequest="{StaticResource boxSize}"
                        Color="{Binding Color}" />
                    <StackLayout Padding="5, 0, 0, 0"
                        VerticalOptions="Center">
                        <Label Text="{Binding FriendlyName}"
                            FontAttributes="Bold"
                            FontSize="14" />
                    <StackLayout Orientation="Horizontal"
                        Spacing="0">
                        <Label Text="{Binding Red,
                            Converter={StaticResource intConverter},
                            ConverterParameter=255,
                            StringFormat='R={0:X2}'}" />
                        <Label Text="{Binding Green,
                            Converter={StaticResource intConverter},
                            ConverterParameter=255,
                            StringFormat=', G={0:X2}'}" />
                        <Label Text="{Binding Blue,
                            Converter={StaticResource intConverter},
                            ConverterParameter=255,
                            StringFormat=', B={0:X2}'}" />
                    </StackLayout>
                </StackLayout>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
</ContentPage>

```

Binding value converters

The previous XAML example displays the individual `Red`, `Green`, and `Blue` properties of each `NamedColor`. These properties are of type `float` 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 `float` values need to be multiplied by 255.

This issue can be 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.Globalization;

namespace XamlSamples
{
    public class FloatToIntConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
        {
            float multiplier;

            if (!float.TryParse(parameter as string, out multiplier))
                multiplier = 1;

            return (int)Math.Round(multiplier * (float)value);
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
        {
            float divider;

            if (!float.TryParse(parameter as string, out divider))
                divider = 1;

            return ((float)(int)value) / divider;
        }
    }
}

```

NOTE

The `ConvertBack` method does not play a role in this example 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 `float` value.

The converter is instantiated in the page's resource dictionary so it can be shared among multiple bindings:

```
<local:FloatToIntConverter x:Key="intConverter" />
```

Three data bindings reference this single instance:

```
<Label Text="{Binding Red,
           Converter={StaticResource intConverter},
           ConverterParameter=255,
           StringFormat='R={0:X2}'}" />
```

The item template displays the color, its friendly name, and its RGB values:

R=00, G=80, B=80

Thistle

R=D8, G=BF, B=D8

Tomato

R=FF, G=63, B=47

Transparent

R=FF, G=FF, B=FF

Turquoise

R=40, G=E0, B=D0

Violet

R=EE, G=82, B=EE

Wheat

R=F5, G=DE, B=B3

White

R=FF, G=FF, B=FF

White Smoke

R=F5, G=F5, B=F5

The `ListView` can handle changes that 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, 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.

Next steps

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 app architectural pattern begins to emerge as a useful paradigm.

[Data binding and MVVM](#)

Data binding and MVVM

9/20/2022 • 9 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The Model-View-ViewModel (MVVM) 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 XAML. The `BindingContext` for the view is usually an instance of the.viewmodel.

Simple MVVM

In [XAML markup extensions](#) you saw how to define a new XML namespace declaration to allow a XAML file to reference classes in other assemblies. The following example uses the `x:Static` markup extension to obtain the current date and time from the static `DateTime.Now` property in the `System` namespace:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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">

    <VerticalStackLayout BindingContext="{x:Static sys:DateTime.Now}"
        Spacing="25" Padding="30,0"
        VerticalOptions="Center" HorizontalOptions="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}'}" />

    </VerticalStackLayout>

</ContentPage>
```

In this example, the retrieved `DateTime` value is set as the `BindingContext` on a `StackLayout`. 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 the same `BindingContext`, and they can contain bindings to properties of that object:

The year is 2017

The month is October

The day is 25

The time is 11:58:23 AM

However, the problem is that the date and time are set once when the page is constructed and initialized, and never change.

A XAML page can display a clock that always shows the current time, but it requires additional code. The MVVM

pattern is a natural choice for .NET MAUI apps when data binding from properties between visual objects and the underlying data. 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. In MVVM, a model is ignorant of the.viewmodel, and a.viewmodel is ignorant of the view. However, often you tailor the types exposed by the.viewmodel to the types associated with the UI.

NOTE

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.

The following example shows a.viewmodel for a clock, with a single property named `DateTime` that's updated every second:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace XamlSamples;

class ClockViewModel: INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private DateTime _dateTime;
    private Timer _timer;

    public DateTime DateTime
    {
        get => _dateTime;
        set
        {
            if (_dateTime != value)
            {
                _dateTime = value;
                OnPropertyChanged(); // reports this property
            }
        }
    }

    public ClockViewModel()
    {
        this.DateTime = DateTime.Now;

        // Update the DateTime property every second.
        _timer = new Timer(new TimerCallback((s) => this.DateTime = DateTime.Now),
                           null, TimeSpan.Zero, TimeSpan.FromSeconds(1));
    }

    ~ClockViewModel() =>
    _timer.Dispose();

    public void OnPropertyChanged([CallerMemberName] string name = "") =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

Viewmodels typically implement the `INotifyPropertyChanged` interface, which provides the ability for a class to raise the `PropertyChanged` event whenever one of its properties changes. The data binding mechanism in .NET MAUI 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. In the previous code example, the `OnPropertyChanged` method handles raising the event while automatically determining the property source name: `DateTime`.

The following example shows XAML that consumes `ClockViewModel`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples"
    x:Class="XamlSamples.ClockPage"
    Title="Clock Page">
    <ContentPage.BindingContext>
        <local:ClockViewModel />
    </ContentPage.BindingContext>

    <Label Text="{Binding DateTime, StringFormat='{0:T}'}"
        FontSize="18"
        HorizontalOptions="Center"
        VerticalOptions="Center" />
</ContentPage>
```

In this example, `ClockViewModel` is set to the `BindingContext` of the `ContentPage` using property element tags. Alternatively, the code-behind file could instantiate the.viewmodel.

The `Binding` markup extension on the `Text` property of the `Label` formats the `DateTime` property. The following screenshot shows the result:

11:59:24 AM

In addition, it's 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.

The following example shows the `HslViewModel` that converts a `Color` value into `Hue`, `Saturation`, and `Luminosity` values, and back again:

```

using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace XamlSamples;

class HslViewModel: INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private float _hue, _saturation, _luminosity;
    private Color _color;

    public float Hue
    {
        get => _hue;
        set
        {
            if (_hue != value)
                Color = Color.FromHsla(value, _saturation, _luminosity);
        }
    }

    public float Saturation
    {
        get => _saturation;
        set
        {
            if (_saturation != value)
                Color = Color.FromHsla(_hue, value, _luminosity);
        }
    }

    public float Luminosity
    {
        get => _luminosity;
        set
        {
            if (_luminosity != value)
                Color = Color.FromHsla(_hue, _saturation, value);
        }
    }

    public Color Color
    {
        get => _color;
        set
        {
            if (_color != value)
            {
                _color = value;
                _hue = _color.GetHue();
                _saturation = _color.GetSaturation();
                _luminosity = _color.GetLuminosity();

                OnPropertyChanged("Hue");
                OnPropertyChanged("Saturation");
                OnPropertyChanged("Luminosity");
                OnPropertyChanged(); // reports this property
            }
        }
    }

    public void OnPropertyChanged([CallerMemberName] string name = "") =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}

```

In this example, changes to the `Hue`, `Saturation`, and `Luminosity` properties cause the `Color` property to change, and changes to the `Color` property causes the other three properties to change. This might seem like an infinite loop, except that the.viewmodel doesn't invoke the `PropertyChanged` event unless the property has changed.

The following XAML example 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples"
    x:Class="XamlSamples.HslColorScrollPage"
    Title="HSL Color Scroll Page">
<ContentPage.BindingContext>
    <local:HslViewModel Color="Aqua" />
</ContentPage.BindingContext>

<VerticalStackLayout Padding="10, 0, 10, 30">
    <BoxView Color="{Binding Color}"
        HeightRequest="100"
        WidthRequest="100"
        HorizontalOptions="Center" />
    <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}"
        HorizontalOptions="Center" />
    <Slider Value="{Binding Hue}"
        Margin="20,0,20,0" />
    <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}"
        HorizontalOptions="Center" />
    <Slider Value="{Binding Saturation}"
        Margin="20,0,20,0" />
    <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}"
        HorizontalOptions="Center" />
    <Slider Value="{Binding Luminosity}"
        Margin="20,0,20,0" />
</VerticalStackLayout>
</ContentPage>
```

The binding on each `Label` is the default `OneWay`. It only needs to display the value. However, the default binding on each `Slider` is `TwoWay`. This allows the `Slider` to be initialized from the.viewmodel. When the.viewmodel is instantiated it's `Color` property is set to `Aqua`. A change in a `Slider` sets a new value for the property in the.viewmodel, which then calculates a new color:



Hue = 0.83



Saturation = 0.91



Luminosity = 0.70



Commanding

Sometimes an app has needs that go beyond 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 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`

NOTE

Many other controls also define `Command` and `CommandParameter` properties.

The `ICommand` interface is defined in the `System.Windows.Input` namespace, and consists of 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 raised. If `CanExecute` returns `false`, the `Button` disables itself and doesn't generate `Execute` calls.

You can use the `Command` or `Command<T>` class included in .NET MAUI to implement the `ICommand` interface. These two classes define several constructors plus a `ChangeCanExecute` method that the.viewmodel can call to force the `Command` object to raise the `CanExecuteChanged` event.

The following example shows a.viewmodel for a simple keypad that is intended for entering telephone numbers:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;

namespace XamlSamples;

class KeypadViewModel: INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private string _inputString = "";
    private string _displayText = "";
    private char[] _specialChars = { '*', '#' };

    public ICommand AddCharCommand { get; private set; }
    public ICommand DeleteCharCommand { get; private set; }

    public string InputString
    {
        get => _inputString;
        private set
        {
            if (_inputString != value)
            {
                _inputString = value;
                OnPropertyChanged();
                DisplayText = FormatText(_inputString);

                // Perhaps the delete button must be enabled/disabled.
                ((Command)DeleteCharCommand).ChangeCanExecute();
            }
        }
    }

    public string DisplayText
    {
        get => _displayText;
        private set
        {
            if (_displayText != value)
            {
                _displayText = value;
                OnPropertyChanged();
            }
        }
    }
}
```

```

public KeypadViewModel()
{
    // Command to add the key to the input string
    AddCharCommand = new Command<string>((key) => InputString += key);

    // Command to delete a character from the input string when allowed
    DeleteCharCommand =
        new Command(
            // Command will strip a character from the input string
            () => InputString = InputString.Substring(0, InputString.Length - 1),

            // CanExecute is processed here to return true when there's something to delete
            () => InputString.Length > 0
        );
}

string FormatText(string str)
{
    bool hasNonNumbers = str.IndexOfAny(_specialChars) != -1;
    string formatted = str;

    // Format the string based on the type of data and the length
    if (hasNonNumbers || str.Length < 4 || str.Length > 10)
    {
        // Special characters exist, or the string is too small or large for special formatting
        // Do nothing
    }

    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;
}

public void OnPropertyChanged([CallerMemberName] string name = "") =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}

```

In this example, the `Execute` and `CanExecute` methods for the commands are defined as lambda functions in the constructor. The viewmodel assumes that the `AddCharCommand` property is bound to the `Command` property of several buttons (or anything other controls that have 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's 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 example shows the XAML that consumes the `KeypadViewModel`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:XamlSamples"
    x:Class="XamlSamples.KeypadPage"
    Title="Keypad Page">
<ContentPage.BindingContext>
    <local:KeypadViewModel />
</ContentPage.BindingContext>

<Grid HorizontalOptions="Center" VerticalOptions="Center">
    <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>

    <Label Text="{Binding DisplayText}"
        Margin="0,0,10,0" FontSize="20" LineBreakMode="HeadTruncation"
        VerticalTextAlignment="Center" HorizontalTextAlignment="End"
        Grid.ColumnSpan="2" />

    <Button Text="ꇦ" Command="{Binding DeleteCharCommand}" Grid.Column="2"/>

    <Button Text="1" Command="{Binding AddCharCommand}" CommandParameter="1" Grid.Row="1" />
    <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" />
    <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" />
    <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" />
    <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>

```

In this example, the `Command` property of the first `Button` that is bound to the `DeleteCharCommand`. The other buttons are bound to the `AddCharCommand` with a `CommandParameter` that's the same as the character that appears on the `Button`:

(800) 642-7676

↔



XAML compilation

9/20/2022 • 2 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) XAML is 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 you 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 enabled by default in .NET MAUI apps. For apps built using debug configuration, XAML compilation provides compile-time validation of XAML, but does not convert the XAML to IL in the assembly. Instead, XAML files are included as embedded resources in the app package, and evaluated at runtime. For apps built using release configuration, XAML compilation provides compile-time validation of XAML, and converts the XAML to IL that's written to the assembly. However, XAML compilation behavior can be overridden in both configurations with the `XamlCompilationAttribute` class.

Enable compilation

XAML compilation can be enabled by passing `XamlCompilationOptions.Compile` to the `XamlCompilationAttribute`:

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

In this example, XAML compilation is enabled for all of the XAML contained within the assembly, with XAML errors being reported at compile-time rather than runtime.

TIP

While the `XamlCompilationAttribute` can be placed anywhere, a good place to put it is in `MauiProgram.cs`.

XAML compilation can also be enabled at the type level:

```
[XamlCompilation (XamlCompilationOptions.Compile)]
public partial class MyPage : ContentPage
{
    ...
}
```

In this example, XAML compilation is enabled only for the `MyPage` class.

NOTE

Compiled bindings can be enabled to improve data binding performance in .NET MAUI applications. For more information, see [Compiled Bindings](#).

Disable compilation

XAML compilation can be disabled by passing `XamlCompilationOptions.Skip` to the `XamlCompilationAttribute`:

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

In this example, XAML compilation is disabled within the assembly, with XAML errors being reported at runtime rather than compile-time.

XAML compilation can also be disabled at the type level:

```
[XamlCompilation (XamlCompilationOptions.Skip)]
public partial class MyPage : ContentPage
{
    ...
}
```

In this example, XAML compilation is disabled only for the `MyPage` class.

WARNING

Disabling XAML compilation is not recommended because XAML is then parsed and interpreted at runtime, which will reduce app performance.

Field modifiers

9/20/2022 • 2 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `x:FieldModifier` attribute specifies the access level for generated fields for named XAML elements.

Valid values of the `x:FieldModifier` 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` – identical to `Internal`.

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 .NET MAUI.

The following conditions must be met for an `x:FieldModifier` attribute to be processed:

- The top-level XAML element must be a valid `x:Class`.
- The current XAML element has an `x:Name` specified.

The following XAML shows examples of setting the attribute:

```
<Label x:Name="privateLabel" />
<Label x:Name="internalLabel" x:FieldModifier="NotPublic" />
<Label x:Name="publicLabel" x:FieldModifier="Public" />
```

IMPORTANT

The `x:FieldModifier` attribute cannot be used to specify the access level of a .NET MAUI XAML class.

Generics

9/20/2022 • 3 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) 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.

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 .NET MAUI namespaces. 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](#).

IMPORTANT

Defining generic classes in .NET MAUI XAML, with the `x>TypeArguments` directive, is unsupported.

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 Common Language Runtime (CLR) type reference for a generic type, and has a similar function to the `typeof` operator in C#. For more information, see [xType 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://schemas.microsoft.com/dotnet/2021/maui"
    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://schemas.microsoft.com/dotnet/2021/maui"
    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://schemas.microsoft.com/dotnet/2021/maui"
    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://schemas.microsoft.com/dotnet/2021/maui"
    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 [Pass constructor arguments](#).

Consume XAML markup extensions

9/20/2022 • 14 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) XAML markup extensions help enhance the power and flexibility of XAML by allowing element attributes to be set from a variety of sources.

For example, you typically set the `Color` property of `BoxView` like this:

```
<BoxView Color="Blue" />
```

However, 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.

A markup extension is a different way to express an attribute of an element. .NET MAUI XAML markup extensions are usually identifiable by an attribute value that is enclosed in curly braces:

```
<BoxView Color="{StaticResource themeColor}" />
```

Any attribute value in curly braces is *always* a XAML markup extension. However, XAML markup extensions can also be referenced without the use of curly braces.

NOTE

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.

In addition to the markup extensions discussed in this article, the following markup extensions are included in .NET MAUI and discussed in other articles:

- `StaticResource` - reference objects from a resource dictionary. For more information, see [Resource dictionaries**](#).
- `DynamicResource` - respond to changes in objects in a resource dictionary. For more information, see [Dynamic styles**](#).
- `Binding` - establish a link between properties of two objects. For more information, see [Data binding**](#).
- `TemplateBinding` - performs data binding from a control template. For more information, see [Control templates](#).
- `RelativeSource` - sets the binding source relative to the position of the binding target. For more information, see [Relative bindings](#).

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 way to use `x:Static` is to first define a class with some constants or static variables, such as this `AppConstants` class:

```
static class AppConstants
{
    public static double NormalFontSize = 18;
}
```

The following XAML demonstrates the most verbose approach to instantiating the `StaticExtension` class between `Label.FontSize` property-element tags:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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 syntax can be simplified even further by 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}" />
```

In this example, 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 root tag of the XAML example also contains an XML namespace declaration for the .NET `System` namespace. 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="π × E sized text"  
      FontSize="{x:Static sys:Math.PI}"  
      Scale="{x:Static sys:Math.E}"  
      HorizontalOptions="Center" />
```

The following screenshot shows the XAML output:

The screenshot shows five labels stacked vertically. The first label has a large font size, followed by four smaller labels below it. The text in all labels is identical: "Label No. 1", "Label No. 2", "Label No. 3", "Label No. 4", and "Label No. 5". Below the labels, there is a small note: "π × E sized text".

```
Label No. 1  
Label No. 2  
Label No. 3  
Label No. 4  
Label No. 5  
π × E sized text
```

x:Reference markup extension

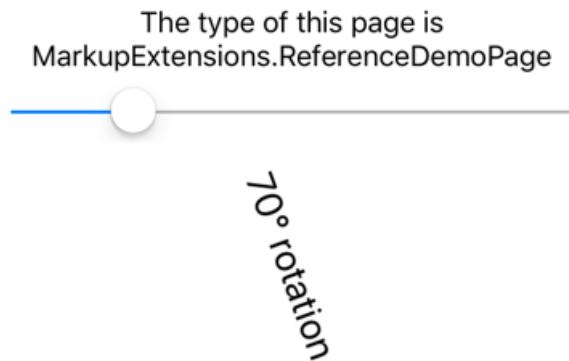
The `x:Reference` markup extension is supported by the `ReferenceExtension` class. The class has a single property named `Name` of type `string` that you set to the name of an element on the page that has been given a name with `x:Name`. This `Name` property is the content property of `ReferenceExtension`, so `Name=` is not required when `x:Reference` appears in curly braces. The `x:Reference` markup extension is used exclusively with data bindings. For more information about data bindings, see [Data binding](#).

The following XAML example 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://schemas.microsoft.com/dotnet/2021/maui"  
             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="Center"  
              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="Center" />  
    </StackLayout>  
</ContentPage>
```

In this example, 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 and the `Source` and `StringFormat` properties are separated by commas.

The following screenshot shows the XAML output:



x>Type markup extension

The `x>Type` markup extension is the XAML equivalent of the C# `typeof` keyword. It's supported by the `TypeExtension` class, which defines a property named `TypeName` of type `string` that should be set to a class or structure name. The `x>Type` markup extension returns the `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.

The `x>Type` markup extension is commonly used with the `x:Array` markup extension. For more information, see [x:Array markup extension](#).

The following XAML example demonstrates using the `x>Type` markup extension to instantiate .NET MAUI objects and add them to a `StackLayout`. The XAML consists of three `Button` elements with their `Command` properties set to a `Binding` and the `CommandParameter` properties set to types of three .NET MAUI views:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="Center"
            Command="{Binding CreateCommand}"
            CommandParameter="{x>Type Slider}" />
        <Button Text="Create a Stepper"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            Command="{Binding CreateCommand}"
            CommandParameter="{x>Type Stepper}" />
        <Button Text="Create a Switch"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            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 ICommand CreateCommand { get; private set; }

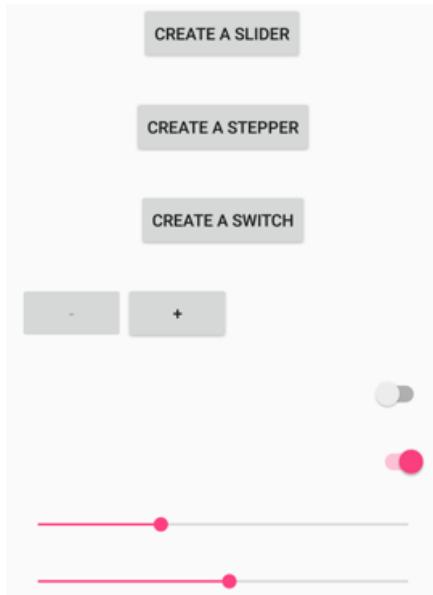
    public TypeDemoPage()
    {
        InitializeComponent();

        CreateCommand = new Command<Type>((Type viewType) =>
    {
        View view = (View)Activator.CreateInstance(viewType);
        view.VerticalOptions = LayoutOptions.Center;
        stackLayout.Add(view);
    });
    }

    BindingContext = this;
}
}

```

When a `Button` is pressed a new instance of the `CommandParameter` argument is created and added to the `StackLayout`. The three `Button` objects 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. This property should be set to an `x:Type` markup extension.
- `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.

The following XAML example shows how to use `x:Array` to add items to a `ListView` by setting the `ItemsSource` property to an array:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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>

```

In this example, the `ViewCell` creates a simple `BoxView` for each color entry:



NOTE

When defining arrays of common types like strings or numbers, use the XAML language primitives tags listed in [Pass arguments](#).

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 following XAML example shows how to use the `x:Null` markup extension:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MarkupExtensions.NullDemoPage"
    Title="x:Null Demo">
    <ContentPage.Resources>
        <Style TargetType="Label">
            <Setter Property="FontSize" Value="48" />
            <Setter Property="FontFamily" Value="OpenSansRegular" />
        </Style>
    </ContentPage.Resources>

    <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>
```

In this example, an implicit `Style` is defined for `Label` that includes a `Setter` that sets the `FontFamily` property to a specific font. However, the third `Label` avoids using the font defined in the implicit style by setting its `FontFamily` to `x:Null`:

Text 1

Text 2

Text 3

Text 4

Text 5

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.
- `iOS` of type `object`, that you set to a value to be applied on iOS.
- `MacCatalyst` of type `object`, that you set to a value to be applied on Mac Catalyst.
- `Tizen` of type `object`, that you set to a value to be applied on the Tizen platform.

- `WinUI` of type `object`, that you set to a value to be applied on WinUI.
- `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}"
         WidthRequest="{OnPlatform 250, iOS=200, Android=300}"
         HeightRequest="{OnPlatform 250, iOS=200, Android=300}"
         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 and Android. 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.

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 .NET MAUI. 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 following XAML example 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.

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 apps, see [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 following XAML example shows how to use the `FontImage` markup extension:

```
<Image BackgroundColor="#D1D1D1"
       Source="{FontImage &#xf30c;, FontFamily=Ionicons, 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`:



While the unicode character for the icon is `\uf30c`, it has to be escaped in XAML and so becomes ``.

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.

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 following XAML example shows how to use the `AppThemeBinding` markup extension:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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:

This text is green in light mode, and red in dark mode.
This text is black by default, blue in light mode, and teal in dark mode.

Create XAML markup extensions

9/20/2022 • 3 minutes to read • [Edit Online](#)

{ } [Browse the sample](#)

At the developer level, a .NET Multi-platform App UI (.NET MAUI) XAML markup extension is a class that implements the `IMarkupExtension` OR `IMarkupExtension<T>` interface. 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 .NET MAUI markup extensions:

- `TypeExtension` derives from `IMarkupExtension<Type>`
- `ArrayExtension` derives from `IMarkupExtension<Array>`
- `DynamicResourceExtension` derives from `IMarkupExtension<DynamicResource>`
- `BindingExtension` derives from `IMarkupExtension<BindingBase>`

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.

Often, XAML markup extensions define properties that contribute to the return value, and the `ProvideValue` method has a single argument of type `IServiceProvider`. For more information about service providers, see [Service providers](#).

Create a markup extension

The following XAML markup extension demonstrates how to create your own markup extension. It 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 float H { get; set; }
    public float S { get; set; }
    public float L { get; set; }
    public float A { get; set; } = 1.0f;

    public Color ProvideValue(IServiceProvider serviceProvider)
    {
        return Color.FromHsla(H, S, L);
    }

    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 a `Color` and another that returns an `object`, but the second method can call the first method.

Consume a markup extension

The following XAML demonstrates a variety of approaches that can be used to invoke the `HslColorExtension` to specify the color for a `BoxView`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:MarkupExtensions"
    x:Class="MarkupExtensions.HslColorDemoPage"
    Title="HSL Color Demo">

    <ContentPage.Resources>
        <Style TargetType="BoxView">
            <Setter Property="WidthRequest" Value="80" />
            <Setter Property="HeightRequest" Value="80" />
            <Setter Property="HorizontalOptions" Value="Center" />
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>
    </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>

```

In this example, 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:



Service providers

By using the `IServiceProvider` argument to `ProvideValue`, XAML markup extensions can get access to data about the XAML file in which they're being used. For example, the `IProvideValueTarget` service enables you to retrieve data about the object the markup extension is applied to:

```
IProvideValueTarget provideValueTarget = serviceProvider.GetService(typeof(IProvideValueTarget)) as IProvideValueTarget;
```

The `IProvideValueTarget` interface defines two properties, `TargetObject` and `TargetProperty`. When this information is obtained in the `HslColorExtension` class, `TargetObject` is the `BoxView` and `TargetProperty` is the `Color` property of `BoxView`. This is the property on which the XAML markup extension has been set.

XAML namespaces

9/20/2022 • 3 minutes to read • [Edit Online](#)

XAML uses the `xmlns` XML attribute for namespace declarations. There are two XAML namespace declarations that are always within the root element of a XAML file. The first defines the default namespace:

```
xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
```

The default namespace specifies that elements defined within the XAML file with no prefix refer to .NET Multi-platform App UI (.NET MAUI) classes, such as `ContentPage`, `Label`, and `Button`.

The second namespace declaration uses the `x` prefix:

```
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 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` constructs supported by .NET MAUI:

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 <code>ResourceDictionary</code> . The key's value is used to retrieve the XAML resource, and is typically used as the argument for the <code>StaticResource</code> markup extension.
<code>x:Name</code>	Specifies a runtime object name for the XAML element. Setting <code>x:Name</code> is similar to declaring a variable in code.
<code>x>TypeArguments</code>	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 [Pass arguments](#). For more information about the `x>TypeArguments` attribute, see [Generics](#).

NOTE

In addition to the constructs listed above, .NET MAUI also includes markup extensions that can be consumed through the `x` namespace prefix. For more information, see [Consume 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.

Declare 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 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 app code that's referencing the types.

NOTE

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:MyMauiApp">
  ...
</ContentPage>
```

Alternatively, this can be written as:

```
<ContentPage ... xmlns:local="using:MyMauiApp">
  ...
</ContentPage>
```

The `local` prefix is a convention used to indicate that the types within the namespace are local to the app.

Alternatively, if the types are in a different assembly, the assembly name should also be defined in the namespace declaration:

```
<ContentPage ... xmlns:controls="clr-namespace:Controls;assembly=MyControlLibrary" ...>
  ...
</ContentPage>
```

The namespace prefix is then specified when declaring an instance of a type from an imported namespace:

```
<controls:Expander IsExpanded="True">  
    ...  
</controls:Expander>
```

For information about defining a custom namespace schema, see [Custom namespace schemas](#).

Custom namespace schemas

9/20/2022 • 3 minutes to read • [Edit Online](#)

Types in a .NET Multi-platform App UI (.NET MAUI) library can be referenced in XAML by declaring an XML 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 XML 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 XML namespace declaration to reference all the types in an assembly, even if they are in different namespaces. It also enables a single XML namespace declaration to reference types in multiple assemblies.

For more information about XAML namespaces, see [XAML namespaces](#).

Define a custom namespace schema

The sample application contains a library that exposes some simple controls, such as `circleButton`:

```
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 an 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 a class library:

```
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.

Consume 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.

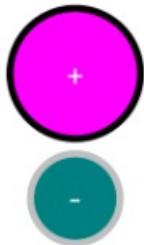
To consume the `CircleButton` control, a XAML namespace is declared, with the namespace declaration specifying the custom namespace schema URL:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="http://mycompany.com/schemas/controls"
    x:Class="CustomNamespaceSchemaDemo.MainPage">
    <StackLayout>
        ...
        <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, .NET MAUI will search referenced assemblies for `XmlNsDefinitionAttribute` instances. If the `xmns` attribute for an element in a XAML file matches the `XmlNamespace` property value in a `XmlNsDefinitionAttribute`, .NET MAUI will attempt to use the `XmlNsDefinitionAttribute.ClrNamespace` property value for resolution of the type. If type resolution fails, .NET MAUI will continue to attempt type resolution based on any additional matching `XmlNsDefinitionAttribute` instances.

The result is that two `CircleButton` instances are displayed:



Custom namespace prefixes

9/20/2022 • 2 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `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://schemas.microsoft.com/dotnet/2021/maui", "maui")]
```

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.

Pass arguments

9/20/2022 • 3 minutes to read • [Edit Online](#)

It's often necessary to instantiate objects with constructors that require arguments, or by calling a static creation method. This can be achieved in .NET Multi-platform App UI (.NET MAUI) 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 [Pass 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 [Call 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 [Specify a generic type argument](#).

Arguments can be passed to constructors and factory methods using the following .NET MAUI XAML language primitives:

- `x:Array`, which corresponds to `Array`.
- `x:Boolean`, which corresponds to `Boolean`.
- `x[Byte]`, which corresponds to `Byte`.
- `x:Char`, which corresponds to `Char`.
- `x:DateTime`, which corresponds to `DateTime`.
- `x:Decimal`, which corresponds to `Decimal`.
- `x:Double`, which corresponds to `Double`.
- `x:Int16`, which corresponds to `Int16`.
- `x:Int32`, which corresponds to `Int32`.
- `x:Int64`, which corresponds to `Int64`.
- `x:Object`, which corresponds to the `Object`.
- `x:Single`, which corresponds to `Single`.
- `x:String`, which corresponds to `String`.
- `x:TimeSpan`, which corresponds to `TimeSpan`.

With the exception of `x:DateTime`, the other language primitives are in the XAML 2009 specification.

NOTE

The `x:Single` language primitive can be used to pass `float` arguments.

Pass 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.

The following example demonstrates using the `x:Arguments` attribute with three different `Color` constructors:

```

<BoxView HeightRequest="150"
         WidthRequest="150"
         HorizontalOptions="Center">
    <BoxView.Color>
        <Color>
            <x:Arguments>
                <x:Single>0.9</x:Single>
            </x:Arguments>
        </Color>
    </BoxView.Color>
</BoxView>
<BoxView HeightRequest="150"
         WidthRequest="150"
         HorizontalOptions="Center">
    <BoxView.Color>
        <Color>
            <x:Arguments>
                <x:Single>0.25</x:Single>
                <x:Single>0.5</x:Single>
                <x:Single>0.75</x:Single>
            </x:Arguments>
        </Color>
    </BoxView.Color>
</BoxView>
<BoxView HeightRequest="150"
         WidthRequest="150"
         HorizontalOptions="Center">
    <BoxView.Color>
        <Color>
            <x:Arguments>
                <x:Single>0.8</x:Single>
                <x:Single>0.5</x:Single>
                <x:Single>0.2</x:Single>
                <x:Single>0.5</x:Single>
            </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 `float` value from 0 (black) to 1 (white). The `Color` constructor with three parameters requires `float` red, green, and blue values ranging from 0 to 1. The `Color` constructor with four parameters adds a `float` alpha channel as the fourth parameter.

Call factory methods

Factory methods can be called in .NET MAUI 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 example demonstrates calling three of them:

```

<BoxView HeightRequest="150"
         WidthRequest="150"
         HorizontalOptions="Center">
    <BoxView.Color>
        <Color x:FactoryMethod="FromRgba">
            <x:Arguments>
                <x:Byte>192</x:Byte>
                <x:Byte>75</x:Byte>
                <x:Byte>150</x:Byte>
                <x:Byte>128</x:Byte>
            </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 `byte` arguments, which represent the red, green, blue, and alpha values, ranging from 0 to 255 respectively. The `FromHsla` factory method requires four `float` arguments, which represent the hue, saturation, luminosity, and alpha values, ranging from 0 to 1 respectively. The `FromHex` factory method requires a `string` argument that represents the hexadecimal (A)RGB color.

Specify 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 example:

```

<StackLayout>
    <StackLayout.Margin>
        <OnPlatform x>TypeArguments="Thickness">
            <On Platform="iOS" Value="0,20,0,0" />
            <On Platform="Android" Value="5, 10" />
        </OnPlatform>
    </StackLayout.Margin>
</StackLayout>

```

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 XAML](#).

Load XAML at runtime

9/20/2022 • 2 minutes to read • [Edit Online](#)

When a .NET Multi-platform App UI (.NET MAUI) XAML class is constructed, a `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, a source generator generates new C# source that contains the definition of the `InitializeComponent` method and adds it to the compilation object. The following example shows the generated `InitializeComponent` method for the `MainPage` class:

```
private void InitializeComponent()
{
    global::Microsoft.Maui.Controls.Xaml.Extensions.LoadFromXaml(this, typeof(MainPage));
    ...
}
```

The `InitializeComponent` method calls the `LoadFromXaml` method to extract the XAML compiled binary (or its file) from the app package. After extraction, it initializes all of the objects defined in the XAML, 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.

Load XAML at runtime

The `Extensions` class, in the `Microsoft.Maui.Controls.Xaml` namespace, includes `LoadFromXaml` extension methods that can be used to load and parse XAML at runtime. The `LoadFromXaml` methods are `public`, and therefore can be called from .NET MAUI applications to load, and parse XAML at runtime. This enables scenarios such as an app downloading XAML from a web service, creating the required view from the XAML, and displaying it in the app.

WARNING

Loading XAML at runtime has a significant performance cost, and generally should be avoided.

The following code example shows a simple usage:

```
string navigationButtonXAML = "<Button Text=\"Navigate\" />";
Button navigationButton = new Button().LoadFromXaml(navigationButtonXAML);
...
stackLayout.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 it's operating on.

The `LoadFromXaml` method can be used to inflate any XAML, with the following example inflating a `ContentPage` and then navigating to it:

```
// See the sample for the full XAML string
string pageXAML = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\r\n<ContentPage
xmlns=\"http://schemas.microsoft.com/dotnet/2021/maui\"\nxmlns:x=\"http://schemas.microsoft.com/winfx/2009/x
aml\"\nx:Class=\"LoadRuntimeXAML.CatalogItemsPage\"\nTitle=\"Catalog Items\">\n</ContentPage>";

ContentPage page = new ContentPage().LoadFromXaml(pageXAML);
await Navigation.PushAsync(page);
```

Access elements

Loading XAML at runtime with the `LoadFromXaml` method does not allow 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://schemas.microsoft.com/dotnet/2021/maui\"\nxmlns:x=\"http://schemas.microsoft.com/winfx/2009/x
aml\"\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.

XAML Hot Reload for .NET MAUI

9/20/2022 • 3 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) XAML Hot Reload is a Visual Studio feature that enables you to view the result of XAML changes in your running app, without having to rebuild your project. Without XAML Hot Reload, you have to build and deploy your app every time you want to view the result of a XAML change.

When your .NET MAUI app is running in debug configuration, with the debugger attached, XAML Hot Reload parses your XAML edits and sends those changes to the running app. It preserves your UI state, since it doesn't recreate the UI for the full page, and updates changed properties on controls affected by edits. In addition, your navigate state and data will be maintained, enabling you to quickly iterate on your UI without losing your location in the app. Therefore, you'll spend less time rebuilding and deploying your apps to validate UI changes.

By default, you don't need to save your XAML file to see the results of your edits. Instead, 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 in the Hot Reload IDE settings available by selecting **Debug > Options > XAML Hot Reload** from the Visual Studio menu bar. Only updating on file save can sometimes be useful if you make bigger XAML updates and don't wish them to be displayed until they are complete.

NOTE

If you're writing a native UWP or WPF app, without using .NET MAUI, see [What is XAML Hot Reload for WPF and UWP apps?](#).

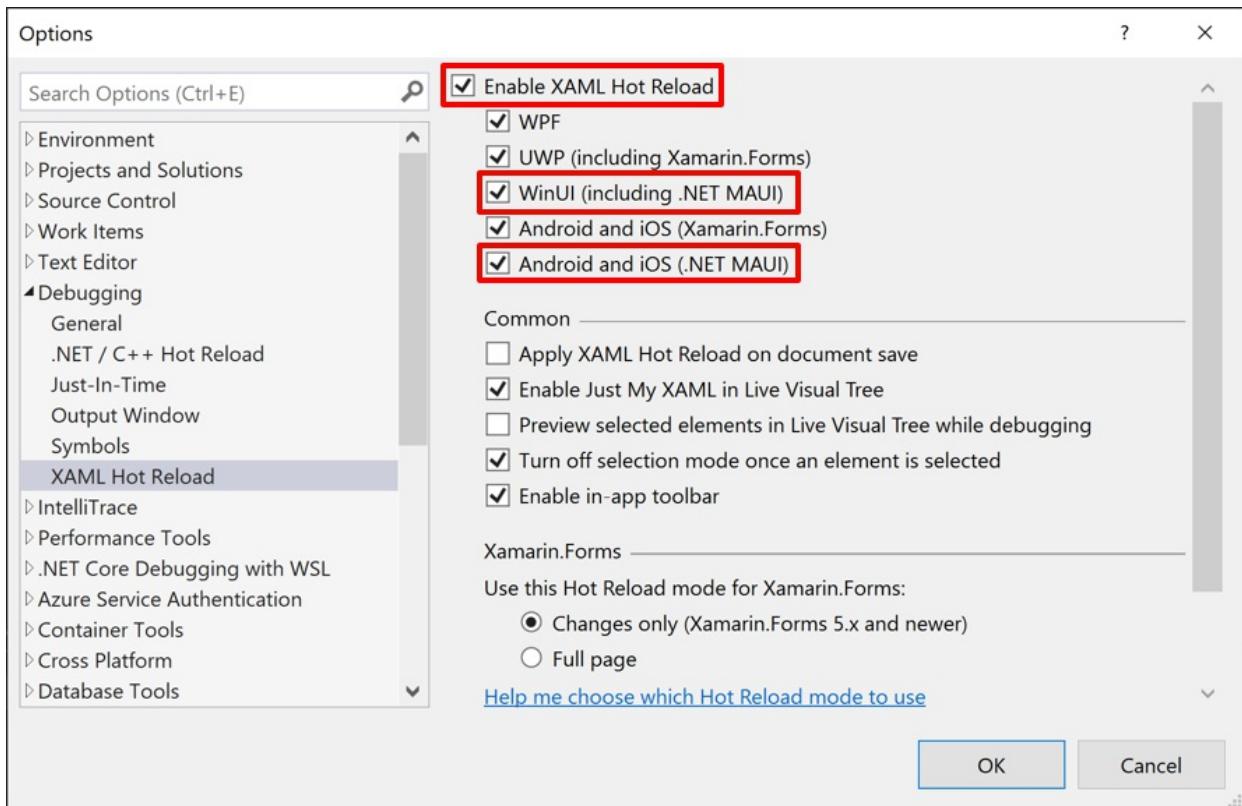
On Windows, XAML Hot Reload is available on Android, iOS, and WinUI on emulators, simulators, and physical devices.

IMPORTANT

XAML Hot Reload doesn't reload C# code, including event handlers.

Enable XAML Hot Reload

XAML Hot Reload is enabled by default in Visual Studio 2022. If it's been previously disabled, it can be enabled by selecting **Debug > Options > XAML Hot Reload** from the Visual Studio menu bar. Next, in the **Options** dialog box, ensure that the **Enable XAML Hot Reload, WinUI (including .NET MAUI)**, and **Android and iOS (.NET MAUI)** options are checked:



Then, on iOS in your build settings, check that the Linker is set to "Don't Link".

Troubleshooting

The XAML Hot Reload output displays status messages that can help with troubleshooting. These can be displayed by selecting **View > Output** from the Visual Studio menu bar, and then selecting **Xamarin Hot Reload** in the **Show output from:** drop-down.

If XAML Hot Reload fails to initialize you should ensure that you're using the latest version of .NET MAUI, the latest version of the IDE, and that your iOS linker settings are set to **Don't Link** in the project's build settings.

If nothing happens when saving your XAML file, ensure that XAML Hot Reload is enabled in the IDE. For more information, see [Enable XAML Hot Reload](#).

If you make a change that the XAML Hot Reload parser sees as invalid, it will show the error underlined in the editor and include it in the **Error List** window. Hot Reload errors have an error code starting with "XHR" (for XAML Hot Reload). If there are any such errors on the page, XAML Hot Reload won't apply changes to your running app until the errors have been fixed.

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.

Disabling XAML compilation with `[XamlCompilation(XamlCompilationOptions.Skip)]` isn't supported and can cause issues with the Live Visual Tree. For more information about Live Visual Tree, see [Inspect the visual tree of a .NET MAUI app](#).

Build accessible apps with semantic properties

9/20/2022 • 15 minutes to read • [Edit Online](#)

Semantics for accessibility is concerned with building experiences that make your apps inclusive for people who use technology in a wide range of environments and approach your UI with a range of needs and experiences. In many situations, legal requirements for accessibility may provide an impetus for developers to address accessibility issues. Regardless, it's advisable to build inclusive and accessible apps so that your apps reach the largest possible audience.

The [Web Content Accessibility Guidelines \(WCAG\)](#) are the global accessibility standard and legal benchmark for web and mobile. These guidelines describe the various ways in which apps can be made more perceivable, operable, understandable, and robust, for all.

Many user accessibility needs are met by assistive technology products installed by the user or by tools and settings provided by the operating system. This includes functionality such as screen readers, screen magnification, and high-contrast settings.

Screen readers typically provide auditory descriptions of controls that are displayed on the screen. These descriptions help users navigate through the app and provide references to controls, such as images, that have no input or text. Screen readers are often controlled through gestures on the touchscreen, trackpad, or keyboard. For information about enabling screen readers, see [Enable screen readers](#).

Operating systems have their own screen readers with their own unique behavior and configuration. For example, most screen readers read the text associated with a control when it receives focus, enabling users to orient themselves as they navigate through the app. However, some screen readers can also read the entire app 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.

Most screen readers will automatically read any text associated with a control that receives accessibility focus. This means that controls, such as `Label` or `Button`, that have a `Text` property set will be accessible for the user. However, `Image`, `ImageButton`, `ActivityIndicator`, and others might not be in the accessibility tree because no text is associated with them.

.NET Multi-platform App UI (.NET MAUI) supports two approaches to providing access to the accessibility experience of the underlying platform. *Semantic properties* are the .NET MAUI approach to providing accessibility values in apps, and are the recommended approach. *Automation properties* are the Xamarin.Forms approach to providing accessibility values in apps, and have been superseded by semantic properties. In both cases, the default accessibility order of controls is the same order in which they're listed in XAML or added to the layout. However, different layouts might have additional factors that influence accessibility order. For example, the accessibility order of `StackLayout` is also based on its orientation, and the accessibility order of `Grid` is based on its row and column arrangement. For more information about content ordering, see [Meaningful Content Ordering](#) on the Xamarin blog.

NOTE

When a `WebView` displays a website that's accessible, it will also be accessible in a .NET MAUI app. Conversely, when a `WebView` displays a website that's not accessible, it won't be accessible in a .NET MAUI app.

Semantic properties

Semantic properties are used to define information about which controls should receive accessibility focus and

which text should be read aloud to the user. Semantic properties are attached properties that can be added to any element to set the underlying platform accessibility APIs.

IMPORTANT

Semantic properties don't try to force equivalent behavior on each platform. Instead, they rely on the accessibility experience provided by each platform.

The `SemanticProperties` class defines the following attached properties:

- `Description`, of type `string`, which represents a description that will be read aloud by the screen reader. For more information, see [Description](#).
- `Hint`, of type `string`, which is similar to `Description`, but provides additional context such as the purpose of a control. For more information, see [Hint](#).
- `HeadingLevel`, of type `SemanticHeadingLevel`, which enables an element to be marked as a heading to organize the UI and make it easier to navigate. For more information, see [Heading levels](#).

These attached properties set platform accessibility values so that a screen reader can speak about the element. For more information about attached properties, see [Attached properties](#).

Description

The `SemanticProperties.Description` attached property represents a short, descriptive `string` that a screen reader uses to announce an element. This property should be set for elements that have a meaning that's important for understanding the content or interacting with the user interface. Setting this property can be accomplished in XAML:

```
<Image Source="dotnet_bot.png"
      SemanticProperties.Description="Cute dot net bot waving hi to you!" />
```

Alternatively, it can be set in C#:

```
Image image = new Image { Source = "dotnet_bot.png" };
SemanticProperties.SetDescription(image, "Cute dot net bot waving hi to you!");
```

In addition, the `SetValue` method can also be used to set the `SemanticProperties.Description` attached property:

```
image.SetValue(SemanticProperties.DescriptionProperty, "Cute dot net bot waving hi to you!");
```

WARNING

Avoid setting the `SemanticProperties.Description` attached property on a `Label`. This will stop the `Text` property being spoken by the screen reader. This is because the visual text should ideally match the text read aloud by the screen reader.

The accessibility information for an element can also be defined on another 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 SemanticProperties.Description="{Binding Source={x:Reference label} Path=Text}" />
```

Alternatively, it can be set in C# as follows:

```
Label label = new Label
{
    Text = "Enter your name: "
};
Entry entry = new Entry();
SemanticProperties.SetDescription(entry, label.Text);
```

Hint

The `SemanticProperties.Hint` attached property represents a `string` that provides additional context to the `SemanticProperties.Description` attached property, such as the purpose of a control. Setting this property can be accomplished in XAML:

```
<Image Source="like.png"
      SemanticProperties.Description="Like"
      SemanticProperties.Hint="Like this post." />
```

Alternatively, it can be set in C#:

```
Image image = new Image { Source = "like.png" };
SemanticProperties.SetDescription(image, "Like");
SemanticProperties.SetHint(image, "Like this post.");
```

In addition, the `SetValue` method can also be used to set the `SemanticProperties.Hint` attached property:

```
image.SetValue(SemanticProperties.HintProperty, "Like this post.");
```

On Android, this property behaves slightly differently depending on the control it's attached to. For example, for controls without text values, such as `Switch` and `CheckBox`, the controls will display the hint with the control. However, for controls with text values, the hint is not displayed and is read after the text value.

WARNING

The `SemanticProperties.Hint` property conflicts with the `Entry.Placeholder` property on Android, which both map to the same platform property. Therefore, setting a different `SemanticProperties.Hint` value to the `Entry.Placeholder` value isn't recommended.

Heading levels

The `SemanticProperties.HeadingLevel` attached property enables an element to be marked as a heading to organize the UI and make it easier to navigate. Some screen readers enable users to quickly jump between headings.

Headings have a level from 1 to 9, and are represented by the `SemanticHeadingLevel` enumeration, which defines `None`, and `Level1` through `Level9` members.

IMPORTANT

While Windows offers 9 levels of headings, Android and iOS only offer a single heading. Therefore, when `SemanticProperties.HeadingLevel` is set on Windows it maps to the correct heading level. However, when set on Android and iOS it maps to a single heading level.

The following example demonstrates setting this attached property:

```
<Label Text="Get started with .NET MAUI"
      SemanticProperties.HeadingLevel="Level1" />
<Label Text="Paragraphs of text go here." />
<Label Text="Installation"
      SemanticProperties.HeadingLevel="Level2" />
<Label Text="Paragraphs of text go here." />
<Label Text="Build your first app"
      SemanticProperties.HeadingLevel="Level3" />
<Label Text="Paragraphs of text go here." />
<Label Text="Publish your app"
      SemanticProperties.HeadingLevel="Level4" />
<Label Text="Paragraphs of text go here." />
```

Alternatively, it can be set in C#:

```
Label label1 = new Label { Text = "Get started with .NET MAUI" };
Label label2 = new Label { Text = "Paragraphs of text go here." };
Label label3 = new Label { Text = "Installation" };
Label label4 = new Label { Text = "Paragraphs of text go here." };
Label label5 = new Label { Text = "Build your first app" };
Label label6 = new Label { Text = "Paragraphs of text go here." };
Label label7 = new Label { Text = "Publish your app" };
Label label8 = new Label { Text = "Paragraphs of text go here." };
SemanticProperties.SetHeadingLevel(label1, SemanticHeadingLevel.Level1);
SemanticProperties.SetHeadingLevel(label3, SemanticHeadingLevel.Level1);
SemanticProperties.SetHeadingLevel(label5, SemanticHeadingLevel.Level1);
SemanticProperties.SetHeadingLevel(label7, SemanticHeadingLevel.Level1);
```

In addition, the `SetValue` method can also be used to set the `SemanticProperties.HeadingLevel` attached property:

```
label1.SetValue(SemanticProperties.HeadingLevelProperty, SemanticHeadingLevel.Level1);
```

Semantic focus

Controls have a `SetSemanticFocus` extension method, defined in the `Microsoft.Maui` namespace, which forces screen reader focus to a specified element. For example, given a `Label` named `label`, screen reader focus can be forced to the element with the following code:

```
label.SetSemanticFocus();
```

Semantic screen reader

.NET Maui provides the `ISemanticScreenReader` interface, with which you can instruct a screen reader to announce text to the user. The interface is exposed through the `SemanticScreenReader.Default` property, and is available in the `Microsoft.Maui.Accessability` namespace.

To instruct a screen reader to announce text, use the `Announce` method, passing a `string` argument that represents the text. The following example demonstrates using this method:

```
SemanticScreenReader.Default.Announce("This is the announcement text.");
```

Limitations

The default platform screen reader must be enabled for text to be read aloud.

Automation properties

Automation properties are attached properties that can be added to any element to indicate how the element is reported to the underlying platform's accessibility framework.

The `AutomationProperties` class defines the following attached properties:

- `ExcludedWithChildren`, of type `bool?`, determines if an element and its children should be excluded from the accessibility tree. For more information, see [ExcludedWithChildren](#).
- `IsInAccessibleTree`, of type `bool?`, indicates whether the element is available in the accessibility tree. For more information, see [IsInAccessibleTree](#).
- `Name`, of type `string`, represents a short description of the element that serves as a speakable identifier for that element. For more information, see [Name](#).
- `HelpText`, of type `string`, represents a longer description of the element, which can be thought of as tooltip text that's associated with the element. For more information, see [HelpText](#).
- `LabeledBy`, of type `VisualElement`, which enables another element to define accessibility information for the current element. For more information, see [LabeledBy](#).

These attached properties set platform accessibility values so that a screen reader can speak about the element. For more information about attached properties, see [Attached properties](#).

Different screen readers read different accessibility values. Therefore, when using automation properties it's recommended that thorough accessibility testing is carried out on each platform to ensure an optimal experience.

IMPORTANT

Automation properties are the Xamarin.Forms approach to providing accessibility values in apps, and have been superseded by semantic properties. For more information about semantic properties, see [Semantic properties](#).

ExcludedWithChildren

The `AutomationProperties.ExcludedWithChildren` attached property, of type `bool?`, determines if an element and its children should be excluded from the accessibility tree. This enables scenarios such as displaying an `AbsoluteLayout` over another layout such as a `StackLayout`, with the `StackLayout` being excluded from the accessibility tree when it's not visible. It can be used from XAML as follows:

```
<StackLayout AutomationProperties.ExcludedWithChildren="true">
...
</StackLayout>
```

Alternatively, it can be set in C# as follows:

```
StackLayout stackLayout = new StackLayout();
...
AutomationProperties.SetExcludedWithChildren(stackLayout, true);
```

When this attached property is set, .NET MAUI sets the `AutomationProperties.IsInAccessibleTree` attached property to `false` on the specified element and its children.

IsInAccessibleTree

WARNING

This attached property should typically remain unset. The majority of controls should be present in the accessibility tree, and the `AutomationProperties.ExcludedWithChildren` attached property can be set in scenarios where an element and its children need removing from the accessibility tree.

The `AutomationProperties.IsInAccessibleTree` attached property, of type `bool?`, determines if the element is visible to screen readers. It must be set to `true` to use the other automation properties. This can be accomplished in XAML as follows:

```
<Entry AutomationProperties.IsInAccessibleTree="true" />
```

Alternatively, it can be set in C# as follows:

```
Entry entry = new Entry();
AutomationProperties.SetIsInAccessibleTree(entry, true);
```

Name

IMPORTANT

The `AutomationProperties.Name` attached property has been superseded by the `SemanticProperties.Description` attached property.

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:

```
ActivityIndicator activityIndicator = new ActivityIndicator();
AutomationProperties.SetIsInAccessibleTree(activityIndicator, true);
AutomationPropertiesSetName(activityIndicator, "Progress indicator");
```

HelpText

IMPORTANT

The `AutomationProperties.HelpText` attached property has been superseded by the `SemanticProperties.Hint` attached property.

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:

```
Button button = new Button { Text = "Toggle ActivityIndicator" };  
AutomationProperties.SetIsInAccessibleTree(button, true);  
AutomationProperties.SetHelpText(button, "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.

LabeledBy

IMPORTANT

The `AutomationProperties.LabeledBy` attached property has been superseded by bindings. For more information, see [SemanticProperties: Description](#).

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:

```
Label label = new Label { Text = "Enter your name: " };  
Entry entry = new Entry();  
AutomationProperties.SetIsInAccessibleTree(entry, true);  
AutomationProperties.SetLabeledBy(entry, label);
```

IMPORTANT

The `AutomationProperties.LabeledByProperty` is not supported on iOS.

Testing accessibility

.NET MAUI apps 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:

- [Test your app's accessibility](#) on Android.
- [Verifying app accessibility on iOS.](#)
- [Testing for accessibility on OS X](#)
- [Accessibility testing](#) on Windows.

The following tools can assist with your accessibility testing:

- [Accessibility Insights](#) for Android and Windows apps.

- [Accessibility Scanner](#) for Android apps.
- [Accessibility Inspector](#) for iOS and macOS apps.
- [Android Studio Layout Inspector](#) for Android apps.
- [Xcode View Debugger](#) for iOS and macOS apps.

However, none of these tools can perfectly emulate the screen reader user experience, and the best way to test and troubleshoot your apps for accessibility will always be manually on physical devices with screen readers.

Enabling screen readers

Each platform has a different default screen reader to narrate accessibility values:

- Android has TalkBack. For information on enabling TalkBack, see [Enable TalkBack](#).
- iOS and macOS have VoiceOver. For information on enabling VoiceOver, see [Enable VoiceOver](#).
- Windows has Narrator. For information on enabling Narrator, see [Enable Narrator](#).

Enable TalkBack

TalkBack is the primary screen reader used on Android. How it's enabled depends on the device manufacturer, Android version, and TalkBack version. However, TalkBack can typically be enabled on your Android device via the device settings:

1. Open the **Settings** app.
2. Select **Accessibility > TalkBack**.
3. Turn **Use TalkBack** on.
4. Select **OK**.

NOTE

While these steps apply to most devices, you might experience some differences.

A TalkBack tutorial opens automatically the first time you enable TalkBack.

For alternative methods of enabling TalkBack, see [Turn Talkback on or off](#).

Enable VoiceOver

VoiceOver is the primary screen reader used on iOS and macOS. On iOS, VoiceOver can be enabled as follows:

1. Open the **Settings** app.
2. Select **Accessibility > VoiceOver**.
3. Turn **VoiceOver** on.

A VoiceOver tutorial can be opened by selecting **VoiceOver Practice**, once VoiceOver is enabled.

For alternative methods of enabling VoiceOver, see [Turn on and practice VoiceOver on iPhone](#) and [Turn on and practice VoiceOver on iPad](#).

On macOS, VoiceOver can be enabled as follows:

1. Open the **System Preferences**.
2. Select **Accessibility > VoiceOver**.
3. Select **Enable VoiceOver**.
4. Select **Use VoiceOver**.

A VoiceOver tutorial can be opened by selecting **Open VoiceOver Training....**

For alternative methods of enabling VoiceOver, see [Turn VoiceOver on or off on Mac](#).

Enable Narrator

Narrator is the primary screen reader used on Windows. Narrator can be enabled by pressing the **Windows logo key + Ctrl + Enter** together. These keys can be pressed again to stop Narrator.

For more information about Narrator, see [Complete guide to Narrator](#).

Accessibility checklist

Follow these tips to ensure that your .NET MAUI apps are accessible to the widest audience possible:

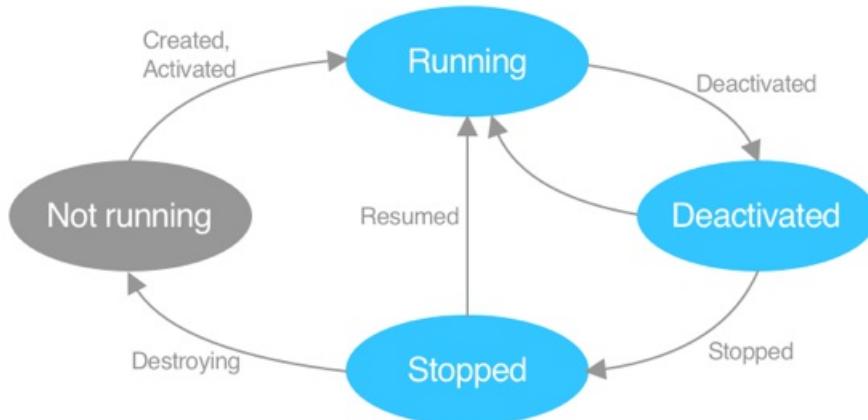
- Ensure your app is perceivable, operable, understandable, and robust for all by following the Web Content Accessibility Guidelines (WCAG). WCAG is the global accessibility standard and legal benchmark for web and mobile. For more information, see [Web Content Accessibility Guidelines \(WCAG\) Overview](#).
- Make sure the user interface is self-describing. Test that all the elements of your user interface are screen reader accessible. Add descriptive text and hints when necessary.
- Ensure that images and icons have alternate text descriptions.
- Support large fonts and high contrast. Avoid hardcoding control dimensions, and instead prefer layouts that resize to accommodate larger font sizes. Test color schemes in high-contrast mode to ensure they are readable.
- Design the visual tree with navigation in mind. Use appropriate layout controls so that navigating between controls using alternate input methods follows the same logical flow as using touch. In addition, exclude unnecessary elements from screen readers (for example, decorative images or labels for fields that are already accessible).
- Don't rely on audio or color cues alone. Avoid situations where the sole indication of progress, completion, or some other state is a sound or color change. Either design the user interface to include clear visual cues, with sound and color for reinforcement only, or add specific accessibility indicators. When choosing colors, try to avoid a palette that is hard to distinguish for users with color blindness.
- Provide captions for video content and a readable script for audio content. It's also helpful to provide controls that adjust the speed of audio or video content, and ensure that volume and transport controls are easy to find and use.
- Localize your accessibility descriptions when the app supports multiple languages.
- Test the accessibility features of your app on each platform it targets. For more information, see [Testing accessibility](#).

App lifecycle

9/20/2022 • 14 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) apps generally have four execution states: *not running*, *running*, *deactivated*, and *stopped*. .NET MAUI raises cross-platform lifecycle events on the `Window` class when an app transitions from the not running state to the running state, the running state to the deactivated state, the deactivated state to the stopped state, the stopped state to the running state, and the stopped state to the not running state.

The following diagram shows an overview of the .NET MAUI app lifecycle:



In the diagram, the gray oval indicates that the app isn't loaded into memory. The light blue ovals indicate that the app is in memory. Text on arcs indicates events that are raised by .NET MAUI, that provide notifications to the running app.

The execution state of an app depends on the app's history. For example, when an app is installed for the first time, or a device is started, the app can be considered to be *not running*. When the app is started, the `Created` and `Activated` events are raised and the app is *running*. If a different app window gains focus, the `Deactivated` event is raised and the app is *deactivated*. If the user switches to a different app or returns to the device's Home screen, so that the app window is no longer visible, the `Deactivated` and `Stopped` events are raised and the app is *stopped*. If the user returns to the app, the `Resuming` event is raised and app is *running*. Alternatively, an app might be terminated by a user while it's running. In this situation the app is *deactivated* then *stopped*, the `Destroying` event is raised, and the app is *not running*. Similarly, a device might terminate an app while it's *stopped*, due to resource restrictions, and the `Destroying` event is raised and the app is *not running*.

In addition, .NET MAUI enables apps to be notified when platform lifecycle events are raised. For more information, see [Platform lifecycle events](#).

Cross-platform lifecycle events

The `Window` class defines the following cross-platform lifecycle events:

EVENT	DESCRIPTION	ACTION TO TAKE
-------	-------------	----------------

EVENT	DESCRIPTION	ACTION TO TAKE
<code>Created</code>	This event is raised after the native window has been created. At this point the cross-platform window will have a native window handler, but the window might not be visible yet.	
<code>Activated</code>	This event is raised when the window has been activated, and is, or will become, the focused window.	
<code>Deactivated</code>	This event is raised when the window is no longer the focused window. However, the window might still be visible.	
<code>Stopped</code>	This event is raised when the window is no longer visible. There's no guarantee that an app will resume from this state, because it may be terminated by the operating system.	Disconnect from any long running processes, or cancel any pending requests that might consume device resources.
<code>Resumed</code>	This event is raised when an app resumes after being stopped. This event won't be raised the first time your app launches, and can only be raised if the <code>Stopped</code> event has previously been raised.	Subscribe to any required events, and refresh any content that's on the visible page.
<code>Destroying</code>	This event is raised when the native window is being destroyed and deallocated. The same cross-platform window might be used against a new native window when the app is reopened.	Remove any event subscriptions that you've attached to the native window.

These cross-platform events map to different platform events, and the following table shows this mapping:

EVENT	ANDROID	IOS	WINDOWS
<code>Created</code>	<code>OnPostCreate</code>	<code>FinishedLaunching</code>	<code>Created</code>
<code>Activated</code>	<code>OnResume</code>	<code>OnActivated</code>	<code>Activated</code> (<code>CodeActivated</code> and <code>PointerActivated</code>)
<code>Deactivated</code>	<code>OnPause</code>	<code>OnResignActivation</code>	<code>Activated</code> (<code>Deactivated</code>)
<code>Stopped</code>	<code>OnStop</code>	<code>DidEnterBackground</code>	<code>VisibilityChanged</code>
<code>Resumed</code>	<code>OnRestart</code>	<code>WillEnterForeground</code>	<code>Resumed</code>
<code>Destroying</code>	<code>OnDestroy</code>	<code>WillTerminate</code>	<code>Closed</code>

In addition to these events, the `Window` class also has the following overridable methods:

- `OnCreated`, which is invoked when the `Created` event is raised.
- `OnActivated`, which is invoked when the `Activated` event is raised.
- `OnDeactivated`, which is invoked when the `Deactivated` event is raised.
- `OnStopped`, which is invoked when the `Stopped` event is raised.
- `OnResumed`, which is invoked when the `Resumed` event is raised.
- `OnDestroying`, which is invoked when the `Destroying` event is raised.

To subscribe to the `Window` lifecycle events, override the `CreateWindow` method in your `App` class to create a `Window` instance on which you can subscribe to events:

```
namespace MyMauiApp
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();

            MainPage = new MainPage();
        }

        protected override Window CreateWindow(IActivationState activationState)
        {
            Window window = base.CreateWindow(activationState);

            window.Created += (s, e) =>
            {
                // Custom logic
            };

            return window;
        }
    }
}
```

Alternatively, to consume the lifecycle overrides, create a class that derives from the `Window` class

```
namespace MyMauiApp
{
    public class MyWindow : Window
    {
        public MyWindow() : base()
        {

        }

        public MyWindow(Page page) : base(page)
        {

        }

        protected override void OnCreated()
        {
            // Register services
        }
    }
}
```

The `Window`-derived class can then be consumed by overriding the `CreateWindow` method in your `App` class to return a `MyWindow` instance.

WARNING

An `InvalidOperationException` will be thrown if the `App.MainPage` property is set and the `CreateWindow` method creates a `Window` object using the override that accepts a `Page` argument.

Platform lifecycle events

.NET MAUI defines delegates that are invoked in response to platform lifecycle events being raised. Handlers can be specified for these delegates, using named methods or anonymous functions, which are executed when the delegate is invoked. This mechanism enables apps to be notified when common platform lifecycle events are raised.

IMPORTANT

The `ConfigureLifecycleEvents` method is in the `Microsoft.Maui.LifecycleEvents` namespace.

Android

The following table lists the .NET MAUI delegates that are invoked in response to Android lifecycle events being raised:

DELEGATE	ARGUMENTS	DESCRIPTION	COMMENTS
<code>OnActivityResult</code>	<code>Android.App.Activity</code> , <code>int</code> , <code>Android.App.Result</code> , <code>Android.Content.Intent?</code>	Invoked when an activity you launched exits.	
<code>OnApplicationConfigurationChanged</code>	<code>Android.App.Application</code> , <code>Android.Content.Res.Configuration</code>	Invoked when the device configuration changes while your component is running.	
<code>OnApplicationCreate</code>	<code>Android.App.Application</code>	Invoked when the app has started, before an activity, service, or receiver objects (excluding content providers) have been created.	
<code>OnApplicationCreating</code>	<code>Android.App.Application</code>	Invoked when the app is starting, before an activity, service, or receiver objects (excluding content providers) have been created.	
<code>OnApplicationLowMemory</code>	<code>Android.App.Application</code>	Invoked when the system is running low on memory, and actively running processes should trim their memory usage.	

DELEGATE	ARGUMENTS	DESCRIPTION	COMMENTS
<code>OnApplicationTrimMemory</code>	<code>Android.App.Application</code> , <code>Android.Content.TrimMemory</code>	Invoked when the operating system has determined that it's a good time for a process to trim unneeded memory from its process.	
<code>OnBackPressed</code>	<code>Android.App.Activity</code>	Invoked when the activity has detected a press of the back key.	
<code>OnConfigurationChanged</code>	<code>Android.App.Activity</code> , <code>Android.Content.Res.Configuration</code>	Invoked when the device configuration changes while your activity is running.	
<code>OnCreate</code>	<code>Android.App.Activity</code> , <code>Android.OS.Bundle?</code>	Raised when the activity is created.	
<code>OnDestroy</code>	<code>Android.App.Activity</code>	Invoked when the activity is finishing, or because the system is temporarily destroying the activity instance to save space.	Always call the super class's implementation.
<code>OnNewIntent</code>	<code>Android.App.Activity</code> , <code>Android.Content.Intent?</code>	Invoked when the activity is relaunched while at the top of the activity stack instead of a new instance of the activity being started.	
<code>OnPause</code>	<code>Android.App.Activity</code>	Invoked when an activity is going into the background, but has not yet been killed.	Always call the super class's implementation.
<code>OnPostCreate</code>	<code>Android.App.Activity</code> , <code>Android.OS.Bundle?</code>	Invoked when activity startup is complete, after <code>OnStart</code> and <code>OnRestoreInstanceState</code> have been called.	Always call the super class's implementation. This is a system-only event that generally shouldn't be used by apps.
<code>OnPostResume</code>	<code>Android.App.Activity</code>	Invoked when activity resume is complete, after <code>OnResume</code> has been called.	Always call the super class's implementation. This is a system-only event that generally shouldn't be used by apps.
<code>OnRequestPermissionsResult</code>	<code>Android.App.Activity</code> , <code>int</code> , <code>string[]</code> , <code>Android.Content.PM.Permission[]</code>	Invoked as a callback for the result from requesting permissions.	
<code>OnRestart</code>	<code>Android.App.Activity</code>	Invoked after <code>onStop</code> when the current activity is being redisplayed to the user (the user has navigated back to it).	Always call the super class's implementation.

DELEGATE	ARGUMENTS	DESCRIPTION	COMMENTS
<code>OnRestoreInstanceState</code>	<code>Android.App.Activity</code> , <code>Android.OS.Bundle</code>	Invoked after <code>OnStart</code> when the activity is being reinitialized from a previously saved state.	
<code>OnResume</code>	<code>Android.App.Activity</code>	Invoked after <code>OnRestoreInstanceState</code> , <code>OnRestart</code> , or <code>OnPause</code> , to indicate that the activity is active and is ready to receive input.	
<code>OnSaveInstanceState</code>	<code>Android.App.Activity</code> , <code>Android.OS.Bundle</code>	Invoked to retrieve per-instance state from an activity being killed so that the state can be restored in <code>OnCreate</code> or <code>OnRestoreInstanceState</code> .	
<code>OnStart</code>	<code>Android.App.Activity</code>	Invoked after <code>OnCreate</code> or <code>OnRestart</code> when the activity has been stopped, but is now being displayed to the user.	Always call the super class's implementation.
<code>OnStop</code>	<code>Android.App.Activity</code>	Invoked when the activity is no longer visible to the user.	Always call the super class's implementation.

IMPORTANT

Each delegate has a corresponding identically named extension method, that can be called to register a handler for the delegate.

To respond to an Android lifecycle delegate being invoked, call the `ConfigureLifecycleEvents` method on the `MauiApplicationBuilder` object in the `CreateMauiapp` method of your `MauiProgram` class. Then, on the `ILifecycleBuilder` object, call the `AddAndroid` method and specify the `Action` that registers handlers for the required delegates:

```

using Microsoft.Maui.LifecycleEvents;

namespace PlatformLifecycleDemo
{
    public static class MauiProgram
    {
        public static MauiApp CreateMauiApp()
        {
            var builder = MauiApp.CreateBuilder();
            builder
                .UseMauiApp<App>()
                .ConfigureLifecycleEvents(events =>
            {
#if ANDROID
                events.AddAndroid(android => android
                    .OnActivityResult((activity, requestCode, resultCode, data) =>
LogEvent("OnActivityResult", requestCode.ToString()))
                    .OnStart((activity) => LogEvent("OnStart"))
                    .OnCreate((activity, bundle) => LogEvent("OnCreate"))
                    .OnBackPressed((activity) => LogEvent("OnBackPressed"))
                    .OnStop((activity) => LogEvent("OnStop")));
#endif
                static void LogEvent(string eventName, string type = null)
                {
                    System.Diagnostics.Debug.WriteLine($"Lifecycle event: {eventName}{{(type == null ? string.Empty : $" ({type}))}}");
                }
            });

            return builder.Build();
        }
    }
}

```

For more information about the Android app lifecycle, see [Understand the Activity Lifecycle](#) on developer.android.com.

iOS

The following table lists the .NET MAUI delegates that are invoked in response to iOS lifecycle events being raised:

DELEGATE	ARGUMENTS	DESCRIPTION
<code>ContinueUserActivity</code>	<code>UIKit.UIApplication</code> , <code>Foundation.NSUserActivity</code> , <code>UIKit.UIApplicationRestorationHandler</code>	Invoked when the app receives data associated with a user activity, such as transferring an activity from a different device using Handoff.
<code>DidEnterBackground</code>	<code>UIKit.UIApplication</code>	Invoked when the app has entered the background.
<code>FinishedLaunching</code>	<code>UIKit.UIApplication</code> , <code>Foundation.NSDictionary</code>	Invoked when the app has launched.
<code>OnActivated</code>	<code>UIKit.UIApplication</code>	Invoked when the app is launched and every time the app returns to the foreground.

DELEGATE	ARGUMENTS	DESCRIPTION
OnResignActivation	UIKit.UIApplication	Invoked when the app is about to enter the background, be suspended, or when the user receives an interruption such as a phone call or text.
OpenUrl	UIKit.UIApplication , Foundation.NSDictionary	Invoked when the app should open a specified URL.
PerformActionForShortcutItem	UIKit.UIApplication , UIKit.UIApplicationShortcutItem , UIKit.UIOperationHandler	Invoked when a Home screen quick action is initiated.
WillEnterForeground	UIKit.UIApplication	Invoked if the app will be returning from a backgrounded state.
WillFinishLaunching	UIKit.UIApplication , Foundation.NSDictionary	Invoked when app launching has begun, but state restoration has not yet occurred.
WillTerminate	UIKit.UIApplication	Invoked if the app is being terminated due to memory constraints, or directly by the user.

IMPORTANT

Each delegate has a corresponding identically named extension method, that can be called to register a handler for the delegate.

To respond to an iOS lifecycle delegate being invoked, call the `ConfigureLifecycleEvents` method on the `MauiAppBuilder` object in the `CreateMauiapp` method of your `MauiProgram` class. Then, on the `ILifecycleBuilder` object, call the `AddiOS` method and specify the `Action` that registers handlers for the required delegates:

```

using Microsoft.Maui.LifecycleEvents;

namespace PlatformLifecycleDemo
{
    public static class MauiProgram
    {
        public static MauiApp CreateMauiApp()
        {
            var builder = MauiApp.CreateBuilder();
            builder
                .UseMauiApp<App>()
                .ConfigureLifecycleEvents(events =>
            {

#if IOS
                events.AddiOS(ios => ios
                    .OnActivated((app) => LogEvent("OnActivated"))
                    .OnResignActivation((app) => LogEvent("OnResignActivation"))
                    .DidEnterBackground((app) => LogEvent("DidEnterBackground"))
                    .WillTerminate((app) => LogEvent("WillTerminate")));
#endif
                static void LogEvent(string eventName, string type = null)
                {
                    System.Diagnostics.Debug.WriteLine($"Lifecycle event: {eventName}{{(type == null ? string.Empty : $" ({type}))}}");
                }
            });

            return builder.Build();
        }
    }
}

```

For more information about the iOS app lifecycle, see [Managing Your App's Life Cycle](#) on developer.apple.com.

Windows

The following table lists the .NET MAUI delegates that are invoked in response to Windows lifecycle events being raised:

DELEGATE	ARGUMENTS	DESCRIPTION
<code>OnActivated</code>	<code>Microsoft.UI.Xaml.Window</code> , <code>Microsoft.UI.Xaml.WindowActivatedEventArgs</code>	Invoked when the platform Activated event is raised, if the app isn't resuming.
<code>OnClosed</code>	<code>Microsoft.UI.Xaml.Window</code> , <code>Microsoft.UI.Xaml.WindowEventArgs</code>	Invoked when the platform Closed event is raised.
<code>OnLaunched</code>	<code>Microsoft.UI.Xaml.Window</code> , <code>Microsoft.UI.Xaml.LaunchActivatedEventArgs</code>	Invoked by .NET MAUI's Application.OnLaunched override once the native window has been created and activated.
<code>OnLaunching</code>	<code>Microsoft.UI.Xaml.Window</code> , <code>Microsoft.UI.Xaml.LaunchActivatedEventArgs</code>	Invoked by .NET MAUI's Application.OnLaunched override before the native window has been created and activated.
<code>OnPlatformMessage</code>	<code>Microsoft.UI.Xaml.Window</code> , <code>WindowsPlatformMessageEventArgs</code>	Invoked when .NET MAUI receives specific native Windows messages.

DELEGATE	ARGUMENTS	DESCRIPTION
<code>OnPlatformWindowSubclassed</code>	<code>Microsoft.UI.Xaml.Window</code> , <code>WindowsPlatformWindowSubclassedEventArgs</code>	Invoked by .NET MAUI when the Win32 window is subclassed.
<code>OnResumed</code>	<code>Microsoft.UI.Xaml.Window</code>	Invoked when the platform <code>Activated</code> event is raised, if the app is resuming.
<code>OnVisibilityChanged</code>	<code>Microsoft.UI.Xaml.Window</code> , <code>Microsoft.UI.Xaml.WindowVisibilityChangedEventArgs</code>	Invoked when the platform <code>VisibilityChanged</code> event is raised.
<code>OnWindowCreated</code>	<code>Microsoft.UI.Xaml.Window</code>	Invoked when the native window is created for the cross-platform <code>Window</code> .

.NET MAUI exposes specific native Windows messages as a lifecycle event with the `OnPlatformMessage` delegate. The `WindowsPlatformMessageEventArgs` object that accompanies this delegate includes a `MessageId` property, of type `uint`. The value of this property can be examined to determine which message has been passed to your app window. For more information about windows messages, see [Windows Messages \(Get Started with Win32 and C++\)](#). For a list of window message constants, see [Window notifications](#).

IMPORTANT

Each delegate has a corresponding identically named extension method, that can be called to register a handler for the delegate.

To respond to a Windows lifecycle delegate being invoked, call the `ConfigureLifecycleEvents` method on the `MauiAppBuilder` object in the `CreateMauiApp` method of your `MauiProgram` class. Then, on the `ILifecycleBuilder` object, call the `AddWindows` method and specify the `Action` that registers handlers for the required delegates:

```

using Microsoft.Maui.LifecycleEvents;

namespace PlatformLifecycleDemo
{
    public static class MauiProgram
    {
        public static MauiApp CreateMauiApp()
        {
            var builder = MauiApp.CreateBuilder();
            builder
                .UseMauiApp<App>()
                .ConfigureLifecycleEvents(events =>
            {

#if WINDOWS
                events.AddWindows(windows => windows
                    .OnActivated((window, args) => LogEvent("OnActivated"))
                    .OnClosed((window, args) => LogEvent("OnClosed"))
                    .OnLaunched((window, args) => LogEvent("OnLaunched"))
                    .OnLaunching((window, args) => LogEvent("OnLaunching"))
                    .OnVisibilityChanged((window, args) => LogEvent("OnVisibilityChanged"))
                    .OnPlatformMessage((window, args) =>
                {
                    if (args.MessageId == Convert.ToInt32("0x02E0"))
                    {
                        // DPI has changed
                    }
                }));
#endif
                static void LogEvent(string eventName, string type = null)
                {
                    System.Diagnostics.Debug.WriteLine($"Lifecycle event: {eventName}{{(type == null ? string.Empty : $"{(type)}")}}");
                }
            });

            return builder.Build();
        }
    }
}

```

Retrieve the Window object

Platform code can retrieve the app's `Window` object from platform lifecycle events, with the `GetWindow` extension method:

```

using Microsoft.Maui.LifecycleEvents;

namespace PlatformLifecycleDemo
{
    public static class MauiProgram
    {
        public static MauiApp CreateMauiApp()
        {
            var builder = MauiApp.CreateBuilder();
            builder
                .UseMauiApp<App>()
                .ConfigureLifecycleEvents(events =>
            {
                #if WINDOWS
                    events.AddWindows(windows => windows
                        .OnClosed((window, args) =>
                    {
                        IWindow appWindow = window.GetWindow();
                    }));
                #endif
            });
            return builder.Build();
        }
    }
}

```

Custom lifecycle events

While .NET MAUI defines delegates that are invoked in response to platform lifecycle events being raised, it only exposes a common set of platform lifecycle events. However, it also includes a mechanism, typically for library authors, that enables apps to be notified when additional platform lifecycle events are raised. The process for accomplishing this is as follows:

- Register an event handler for a platform lifecycle event that isn't exposed by .NET MAUI.
- In the event handler for the platform lifecycle event, retrieve the `ILifecycleEventService` instance and call its `InvokeEvents` method, specifying the platform event name as its argument.

Then, apps that want to receive notification of the platform lifecycle event should modify the `CreateMauiApp` method of their `MauiProgram` class to call the `ConfigureLifecycleEvents` method on the `MauiAppBuilder` object. Then, on the `ILifecycleBuilder` object, call the `AddEvent` method and specify the platform event name and the `Action` that will be invoked when the platform event is raised.

Example

The WinUI 3 `Window.SizeChanged` event occurs when the native app window has first rendered, or has changed its rendering size. .NET MAUI doesn't expose this platform event as a lifecycle event. However, apps can receive notification when this platform event is raised by using the following approach:

- Register an event handler for the `Window.SizeChanged` platform lifecycle event:

```

using Microsoft.Maui.LifecycleEvents;
...

public static MauiApp CreateMauiApp()
{
    var builder = MauiApp.CreateBuilder();
    builder
        .UseMauiApp<App>()
        .ConfigureLifecycleEvents(events =>
    {
        #if WINDOWS
            events.AddWindows(windows => windows
                .OnWindowCreated(window =>
                {
                    window.SizeChanged += OnSizeChanged;
                }));
        #endif
    });
    return builder.Build();
}

```

- In the event handler for the platform lifecycle event, retrieve the `ILifecycleEventService` instance and call its `InvokeEvents` method, specifying the platform event name as its argument:

```

using Microsoft.Maui.LifecycleEvents;
...

#if WINDOWS
    static void OnSizeChanged(object sender, Microsoft.UI.Xaml.WindowSizeChangedEventArgs args)
    {
        ILifecycleEventService service =
        MauiWinUIApplication.Current.Services.GetRequiredService<ILifecycleEventService>();
        service.InvokeEvents(nameof(Microsoft.UI.Xaml.Window.SizeChanged));
    }
#endif

```

The `MauiWinUIApplication` type on Windows can be used to access the native app instance via its `Current` property. The `MauiApplication` type on Android can be used to access the native app instance. Similarly, the `MauiUIApplicationDelegate` type on iOS can be used to access the native app instance.

WARNING

Invoking an unregistered event, with the `InvokeEvents` method, doesn't throw an exception.

- In the `CreateMauiApp` method of your `MauiProgram` class, call the `ConfigureLifecycleEvents` method on the `MauiAppBuilder` object. Then, on the `ILifecycleBuilder` object, call the `AddEvent` method and specify the platform event name and the `Action` that will be invoked when the platform event is raised:

```

using Microsoft.Maui.LifecycleEvents;

namespace PlatformLifecycleDemo
{
    public static class MauiProgram
    {
        public static MauiApp CreateMauiApp()
        {
            var builder = MauiApp.CreateBuilder();
            builder
                .UseMauiApp<App>()
                .ConfigureLifecycleEvents(events =>
            {
                #if WINDOWS
                    events.AddWindows(windows => windows
                        .OnWindowCreated(window =>
                        {
                            window.SizeChanged += OnSizeChanged;
                        }));
                events.AddEvent(nameof(Microsoft.UI.Xaml.Window.SizeChanged), () =>
                    LogEvent("Window SizeChanged"));
                #endif
                static void LogEvent(string eventName, string type = null)
                {
                    System.Diagnostics.Debug.WriteLine($"Lifecycle event: {eventName}{{(type == null ? string.Empty : $" ({type}))}}");
                }
            });

            return builder.Build();
        }
    }
}

```

The overall effect is that when a user changes the app window size on Windows, the action specified in the `AddEvent` method is executed.

NOTE

The `AddEvent` method also has an overload that enables a delegate to be specified.

Behaviors

9/20/2022 • 9 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) 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.

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.

.NET MAUI supports two different types of behaviors:

- Attached behaviors are `static` classes with one or more attached properties. For more information about attached behaviors, see [Attached behaviors](#).
- .NET MAUI behaviors are 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, see [.NET MAUI Behaviors](#).

Attached behaviors

Attached behaviors are static classes with one or more attached properties. 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. For more information about attached properties, see [Attached properties](#).

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.

WARNING

Attached behaviors are defined in a `static` class, with `static` properties and methods. This makes it difficult to create attached behaviors that have state.

Create an attached behavior

An attached behavior can be implemented by creating a static class that contains an attached property that

specifies a `PropertyChanged` delegate.

The following example shows the `AttachedNumericValidationBehavior` class, which highlights the value entered by the user into an `Entry` control in red if it's not a `double`:

```
public static class AttachedNumericValidationBehavior
{
    public static readonly BindableProperty AttachBehaviorProperty =
        BindableProperty.CreateAttached("AttachBehavior", typeof(bool),
            typeof(AttachedNumericValidationBehavior), 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)
    {
        Entry 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 ? Colors.Black : Colors.Red;
    }
}
```

In this example, the `AttachedNumericValidationBehavior` 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 in the `Entry` and sets the `TextColor` property to red if the value isn't a `double`.

Consume an attached behavior

An attached behavior can be consumed by setting its attached property on the target control.

The following example shows consuming the `AttachedNumericValidationBehavior` class on an `Entry` by adding the `AttachBehavior` attached property to the `Entry`:

```
<ContentPage ...>
    <Entry Placeholder="Enter a System.Double" local:AttachedNumericValidationBehavior.AttachBehavior="true" />
</ContentPage>
```

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

```
Entry entry = new Entry { Placeholder = "Enter a System.Double" };
AttachedNumericValidationBehavior.SetAttachBehavior(entry, true);
```

The following screenshot shows 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.

Remove an attached behavior

The `AttachedNumericValidationBehavior` class can be removed from a control by setting the `AttachBehavior` attached property to `false`:

```
<Entry Placeholder="Enter a System.Double" local:AttachedNumericValidationBehavior.AttachBehavior="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 you interact with the control.

.NET MAUI behaviors

.NET MAUI behaviors are created by deriving from the `Behavior` or `Behavior<T>` class.

The process for creating a .NET MAUI 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 example:

```

public class MyBehavior : 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 called 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 called 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 control.

Create a .NET MAUI Behavior

A .NET MAUI behavior can be implemented by creating a class that derives from the `Behavior` or `Behavior<T>` class, and overriding the `OnAttachedTo` and `OnDetachingFrom` methods.

The following example shows the `NumericValidationBehavior` class, which highlights the value entered by the user into an `Entry` control in red if it's not a `double`:

```

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 ? Colors.Black : Colors.Red;
    }
}

```

In this example, the `NumericValidationBehavior` class 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 in the `Entry` and sets the `TextColor` property to red if the value isn't a `double`.

IMPORTANT

.NET MAUI does not set the `BindingContext` of a behavior, because behaviors can be shared and applied to multiple controls through styles.

Consume a .NET MAUI behavior

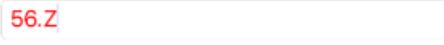
Every .NET MAUI control has a `Behaviors` collection, to which one or more behaviors can be added:

```
<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:

```
Entry entry = new Entry { Placeholder = "Enter a System.Double" };
entry.Behaviors.Add(new NumericValidationBehavior());
```

The following screenshot shows the .NET MAUI behavior responding to invalid input:



WARNING

.NET MAUI 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 .NET MAUI behavior to an incompatible control will result in an exception being thrown.

Consume a .NET MAUI behavior with a style

.NET MAUI behaviors can 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 be attached. 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 example shows the `NumericValidationStyleBehavior` class, which has an attached property that controls adding and removing the behavior:

```

public class NumericValidationStyleBehavior : Behavior<Entry>
{
    public static readonly BindableProperty AttachBehaviorProperty =
        BindableProperty.CreateAttached("AttachBehavior", typeof(bool),
        typeof(NumericValidationStyleBehavior), 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)
    {
        Entry entry = view as Entry;
        if (entry == null)
        {
            return;
        }

        bool attachBehavior = (bool)newValue;
        if (attachBehavior)
        {
            entry.Behaviors.Add(new NumericValidationStyleBehavior());
        }
        else
        {
            Behavior toRemove = entry.Behaviors.FirstOrDefault(b => b is NumericValidationStyleBehavior);
            if (toRemove != null)
            {
                entry.Behaviors.Remove(toRemove);
            }
        }
    }
    ...
}

```

In this example, the `NumericValidationStyleBehavior` 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 `NumericValidationStyleBehavior` 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:NumericValidationStyleBehavior.AttachBehavior" Value="true" />
    </Style.Setters>
</Style>

```

The `Style` can be applied to an `Entry` by setting its `Style` property to the style using the `StaticResource` markup extension:

```
<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`.

Remove a .NET MAUI behavior

The `OnDetachingFrom` method is called 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 the `Remove` or `Clear` method:

```
Behavior toRemove = entry.Behaviors.FirstOrDefault(b => b is NumericValidationStyleBehavior);
if (toRemove != null)
{
    entry.Behaviors.Remove(toRemove);
}
```

Alternatively, the control's `Behaviors` collection can be cleared:

```
entry.Behaviors.Clear();
```

NOTE

.NET MAUI 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.

Data binding

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

A .NET Multi-platform App UI (.NET MAUI) app consists of one or more pages, each of which typically contains multiple user-interface objects called *views*. One of the primary tasks of the app 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 users manipulate 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 successfully, the app 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, which results in a lot of boilerplate code.

Data binding automates this task, and renders the event handlers unnecessary. Data bindings can be implemented either in XAML or code, 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 app is simplified and clarified.

Data binding is therefore the technique of linking properties of two objects so that changes in one property are automatically reflected in the other property. 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.

NOTE

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*.

Basic bindings

9/20/2022 • 7 minutes to read • [Edit Online](#)

{ } [Browse the sample](#)

A .NET Multi-platform App UI (.NET MAUI) 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.

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.

IMPORTANT

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

Consider the following XAML example, whose intent is to rotate a `Label` by manipulating a `Slider`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="Center" />

        <Slider x:Name="slider"
            Maximum="360"
            VerticalOptions="Center" />
    </StackLayout>
</ContentPage>
```

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 this task, and so 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 for the XAML uses a simpler `SetBinding` extension method from the `BindableObjectExtensions` 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`.

IMPORTANT

The target property must be backed by a bindable property. Therefore, the target object must be an instance of a class that derives from `BindableObject`. For more information, see [Bindable properties](#).

The source property 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.

As you manipulate the `Slider`, the `Label` rotates accordingly:



Alternatively, the data binding can be specified in XAML:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="Center"
            BindingContext="{x:Reference Name=slider}"
            Rotation="{Binding Path=Value}" />

        <Slider x:Name="slider"
            Maximum="360"
            VerticalOptions="Center" />
    </StackLayout>
</ContentPage>

```

Just as in code, the data binding is set on the target object, which is the `Label`. Two XAML markup extensions are used to define the data binding:

- 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`.

For more information about XAML markup extensions, see [Consume XAML markup extensions](#).

NOTE

The source property is specified with the `Path` property of the `Binding` markup extension, which corresponds with the `Path` property of the `Binding` class.

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="Center"
    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:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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>

```

In this example, 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.

The code-behind file sets the binding with the `SetBinding` method, with the second argument being 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.

NOTE

The `VisualElement` class also defines `ScaleX` and `ScaleY` properties, which can scale the `visualElement` differently in the horizontal and vertical directions.

Alternatively, the data binding can be specified in XAML:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="Center"
            Scale="{Binding Source={x:Reference slider},
                Path=Value}" />

        <Slider x:Name="slider"
            Minimum="-2"
            Maximum="2"
            VerticalOptions="Center" />
    </StackLayout>
</ContentPage>

```

In this example, the `Binding` markup extension has two properties set, `Source` and `Path`, separated by a comma. The `Source` property is set to an embedded `x:Reference` markup extension that otherwise has the same syntax as setting the `BindingContext`.

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="Center">
    <Label.Scale>
        <Binding Source="{x:Reference slider}"
            Path="Value" />
    </Label.Scale>
</Label>

```

In this example, 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="Center">
    <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. You can also 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 often an instance of a.viewmodel.

NOTE

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

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`.

IMPORTANT

The `BindingContext` property value is inherited through the visual tree.

The following XAML example demonstrates binding context inheritance:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataBindingDemos.BindingContextInheritancePage"
    Title="BindingContext Inheritance">
    <StackLayout Padding="10">
        <StackLayout VerticalOptions="Fill"
            BindingContext="{x:Reference slider}">
            <Label Text="TEXT"
                FontSize="80"
                HorizontalOptions="Center"
                VerticalOptions="End"
                Rotation="{Binding Value}" />
            <BoxView Color="#800000FF"
                WidthRequest="180"
                HeightRequest="40"
                HorizontalOptions="Center"
                VerticalOptions="Start"
                Rotation="{Binding Value}" />
        </StackLayout>
        <Slider x:Name="slider"
            Maximum="360" />
    </StackLayout>
</ContentPage>
```

In this example, 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`:

TEXT



Binding mode

9/20/2022 • 7 minutes to read • [Edit Online](#)

 [Browse the sample](#)

Every .NET Multi-platform App UI (.NET MAUI) 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.

The following example shows a data binding defined on a `Slider`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="Center" />
        <Slider x:Name="slider"
            VerticalOptions="Center"
            Value="{Binding Source={x:Reference label},
                Path=Opacity}" />
    </StackLayout>
</ContentPage>
```

In this example, 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. Therefore, the `Slider` is initialized to the value 1 from the initial `Opacity` value of `Label`. This is shown in the following screenshot:



In addition, the `Slider` continues to work. This is because 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 but the source is also set from the target. This allows the `Slider` to be set from the initial `Opacity` value.

NOTE

Bindable properties don't signal a property change unless the property actually changes. This prevents an infinite loop.

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:

- `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 .NET MAUI 3.0)

Two-way bindings

Most bindable properties have a default binding mode of `OneWay` but some properties have a default binding mode of `TwoWay`, including the following:

- `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 properties are defined as `TwoWay` because when data bindings are used with the Model-View-ViewModel (MVVM) pattern, 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 example above, because it's 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.

One-way-to-source bindings

Read-only bindable properties have a default binding mode of `OneWayToSource`. For example, the `SelectedItem` property of `ListView` has a default binding mode of `OneWayToSource`. This is because a binding on the `SelectedItem` property should result in setting the binding source.

One-time bindings

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.

Several properties have a default binding mode of `OneTime`, including the `IsTextPredictionEnabled` property of `Entry`.

Viewmodels and property-change notifications

When using a viewmodel in a data-binding, the viewmodel is the data-binding source. The viewmodel doesn't 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 typically fires the event when one of its public properties changes value. The event does not need to be raised if the property never changes. The `INotifyPropertyChanged` interface is also implemented by

`BindableObject` and a `PropertyChanged` event is raised whenever a bindable property changes value.

In the following example, data bindings allow you to select a color using three `Slider` elements for the hue, saturation, and luminosity:

```
public class HslColorViewModel : INotifyPropertyChanged
{
    Color color;
    string name;
    float hue;
    float saturation;
    float luminosity;

    public event PropertyChangedEventHandler PropertyChanged;

    public float Hue
    {
        get
        {
            return hue;
        }
        set
        {
            if (hue != value)
            {
                Color = Color.FromHsla(value, saturation, luminosity);
            }
        }
    }

    public float Saturation
    {
        get
        {
            return saturation;
        }
        set
        {
            if (saturation != value)
            {
                Color = Color.FromHsla(hue, value, luminosity);
            }
        }
    }

    public float Luminosity
    {
        get
        {
            return luminosity;
        }
        set
        {
            if (luminosity != value)
            {
                Color = Color.FromHsla(hue, saturation, value);
            }
        }
    }

    public Color Color
    {
        get
        {
            return color;
        }
        set
        {
            if (color != value)
            {
                Color = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Color"));
            }
        }
    }
}
```

```

    {
        if (color != value)
        {
            color = value;
            hue = color.GetHue();
            saturation = color.GetSaturation();
            luminosity = color.GetLuminosity();
            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);
        }
    }
}

public string Name
{
    get
    {
        return name;
    }
    private set
    {
        if (name != value)
        {
            name = value;
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Name"));
        }
    }
}
}

```

In this example, the `HslColorViewModel` class defines `Hue`, `Saturation`, `Luminosity`, `Color`, and `Name` properties. When any one of the three color components changes value, the `Color` property is recalculated, and `PropertyChanged` events are raised for all four properties. When the `Color` property changes, the static `GetNearestColorName` method in the `NamedColor` class obtains the closest named color and sets the `Name` property.

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 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 following XAML consumes the `HslColorViewModel`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.SimpleColorSelectorPage">
    <ContentPage.BindingContext>
        <local:HslColorViewModel Color="MediumTurquoise" />
    </ContentPage.BindingContext>

    <ContentPage.Resources>
        <Style TargetType="Slider">
            <Setter Property="VerticalOptions" Value="CenterAndExpand" />
        </Style>
    </ContentPage.Resources>

    <Grid>
        <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>

```

In this example, the `HslColorViewModel` is instantiated, and `Color` property set, and set as the page's `BindingContext`. The `BoxView`, `Label`, and three `Slider` views inherit the binding context from the `ContentPage`. 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, uses a `TwoWay` binding mode. This enables each `Slider` to be set from the viewmodel, and also for the viewmodel to be set from each `Slider`.

When the example 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:



MediumTurquoise



As you manipulate the sliders, the `BoxView` and `Label` are updated accordingly.

Overriding the binding mode

The binding mode for a target property can be overridden 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 doesn't always produce the expected result. For example, in the following example setting the `Mode` property to `TwoWay` doesn't work as you might expect:

```
<Label Text="TEXT"
    FontSize="40"
    HorizontalOptions="Center"
    VerticalOptions="CenterAndExpand"
    Scale="{Binding Source={x:Reference slider},
        Path=Value,
        Mode=TwoWay}" />
```

In this example, 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.

Alternatively, you can set the binding mode to `OneWayToSource`:

```
<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.

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 a `TwoWay` binding mode 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.

String formatting

9/20/2022 • 3 minutes to read • [Edit Online](#)

{ } [Browse the sample](#)

In a .NET Multi-platform App UI (.NET MAUI) app, it's sometimes 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`.

String formatting in code is typically accomplished with 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. For more information, see [Formatting Types in .NET](#) for more information on string formatting.

String formatting can also be accomplished with data bindings by setting the `StringFormat` property of `Binding` (or the `StringFormat` property of the `Binding` markup extension) to a standard .NET formatting string with a placeholder:

```
<Slider x:Name="slider" />
<Label Text="{Binding Source={x:Reference slider},
    Path=Value,
    StringFormat='The slider value is {0:F2}'}" />
```

In XAML the formatting string is delimited by single-quote characters to help the XAML parser avoid treating the curly braces as another XAML markup extension. In this example, the formatting specification of `F2` causes the value to be displayed with two decimal places.

NOTE

Using 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.

The following example demonstrates several uses of the `StringFormat` property:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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>
        <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>
    </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>

```

In this example, 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.

NOTE

If you need to display left or right curly braces in your formatting string, use a pair of them. For example,

```
StringFormat='{{0:MMMM}}'.
```

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:



The slider value is 0.24

9:41 PM

The TimeSpan is 21:41:00

Some text

The Entry text is "Some text"

12/15/2017 11:55:17

636,489,357,176,684,200 ticks since 1/1/1

The {0:MMMM} specifier produces December

The long date is Friday, December 15, 2017

3.14159265358979

PI to 4 decimal points = 3.1416

PI in scientific notation = 3.1415927E+000

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 following example:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.BetterColorSelectorPage"
    Title="Better Color Selector">
<ContentPage.BindingContext>
    <local:HslColorViewModel Color="Sienna" />
</ContentPage.BindingContext>

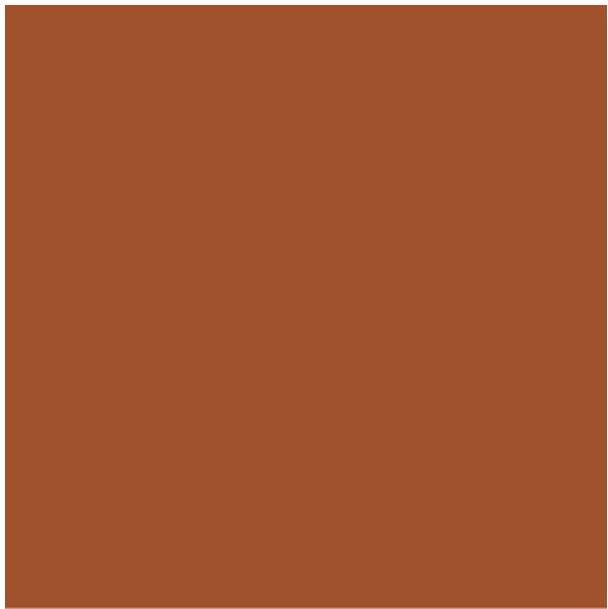
<ContentPage.Resources>
    <ResourceDictionary>
        <Style TargetType="Slider">
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>

        <Style TargetType="Label">
            <Setter Property="HorizontalTextAlignment" Value="Center" />
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>

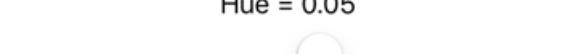
<StackLayout Margin="20">
    <BoxView Color="{Binding Color}"
        HeightRequest="100"
        WidthRequest="100"
        HorizontalOptions="Center" />
    <StackLayout Margin="10, 0">
        <Label Text="{Binding Name}" />
        <Slider Value="{Binding Hue}" />
        <Label Text="{Binding Hue, 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>

```

In this example, there are three pairs of `Slider` and `Label` elements that are bound to the same source property in the `HslColorViewModel` object. Each `Label` that accompanies a `Slider` has a `StringFormat` property to display each `Slider` value:



Sienna



Binding path

9/20/2022 • 3 minutes to read • [Edit Online](#)

 [Browse the sample](#)

In .NET Multi-platform App UI (.NET MAUI), the `Path` property of the `Binding` class (or the `Path` property of the `Binding` markup extension) can be set to a single property, to a *sub-property* (a property of a property), or to a member of a collection.

For example, suppose a page contains a `TimePicker`:

```
<TimePicker x:Name="timePicker">
```

The `Time` property of `TimePicker` is of type `TimeSpan`, and it has a `TotalSeconds` property. A data binding can be created that references the `TotalSeconds` property of that `TimeSpan` value:

```
{Binding Source={x:Reference timePicker},  
        Path=Time.TotalSeconds}
```

The `Time` and `TotalSeconds` properties are simply connected with a period.

NOTE

The items in the `Path` string always refer to properties and not to the types of these properties.

The following XAML shows multiple examples of binding to sub-properties:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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>
    <Style TargetType="Label">
        <Setter Property="FontSize" Value="18" />
        <Setter Property="HorizontalTextAlignment" Value="Center" />
        <Setter Property="VerticalOptions" Value="Center" />
    </Style>
</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

In the example above, the binding in the third `Label` 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>
```

For more information about specifying constructor arguments in XAML, see [Pass constructor arguments](#).

The last `Label` 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:

Select a time:

12:16 AM

960 total seconds

There are 7 children in this
StackLayout

The middle day of the week
is Wednesday

The middle day of the week in
French is mercredi

The second Label has
17 characters

Debug 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 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 `Microsoft.Maui.Controls.StackLayout`. Add the `Children` property to the `Path` and the type is also `Microsoft.Maui.Controls.StackLayout`. Add an index to that and the type is `Microsoft.Maui.Controls.Label`. Continue in this way.

As .NET MAUI 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.

Binding value converters

9/20/2022 • 9 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) 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. Classes that implement `IValueConverter` are called *value converters*, but they are also often referred to as *binding converters* or *binding value converters*.

Binding value converters

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. This can be achieved 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 then 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.

NOTE

If a data binding also includes a `StringFormat` setting, the value converter is invoked before the result is formatted as a string.

The following example demonstrates how to use this value converter in a data binding:

```

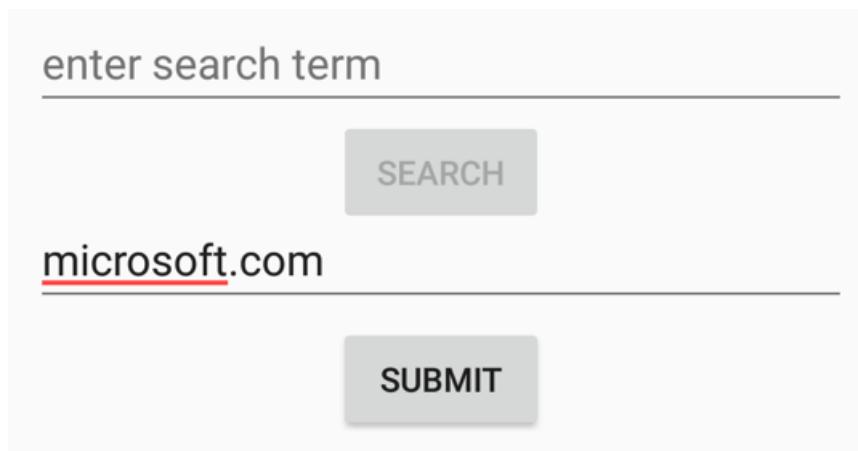
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.EnableButtonsPage"
    Title="Enable Buttons">
    <ContentPage.Resources>
        <local:IntToBoolConverter x:Key="intToBool" />
    </ContentPage.Resources>

    <StackLayout Padding="10, 0">
        <Entry x:Name="entry1"
            Text=""
            Placeholder="enter search term"
            VerticalOptions="Center" />
        <Button Text="Search"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            IsEnabled="{Binding Source={x:Reference entry1},
                Path=Text.Length,
                Converter={StaticResource intToBool}}" />
        <Entry x:Name="entry2"
            Text=""
            Placeholder="enter destination"
            VerticalOptions="Center" />
        <Button Text="Submit"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            IsEnabled="{Binding Source={x:Reference entry2},
                Path=Text.Length,
                Converter={StaticResource intToBool}}" />
    </StackLayout>
</ContentPage>

```

In this example, the `IntToBoolConverter` is instantiated in the page's resource dictionary. It's 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. If a value converter is used in multiple pages of your application, you can instantiate it in the application-level resource dictionary.

This example demonstrates a common need when a `Button` performs an operation based on text that the user types into an `Entry` view. The `Text` property of each `Entry` is initialized to an empty string, because the `Text` property is `null` by default, and the data binding will not work in that case. 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:



NOTE

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 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. In this case 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.

Binding converter properties

Value converter classes can have properties and generic parameters. The following 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 { get; set; }
    public T FalseObject { get; set; }

    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 following example demonstrates how this converter 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 example demonstrates an alternative. Here, 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.SwitchIndicatorsPage"
    Title="Switch Indicators">

    <ContentPage.Resources>
        <Style TargetType="Label">
            <Setter Property="FontSize" Value="18" />
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>

        <Style TargetType="Switch">
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>
    </ContentPage.Resources>

```

```

        </Style>
</ContentPage.Resources>

<StackLayout Padding="10, 0">
    <StackLayout Orientation="Horizontal"
                 VerticalOptions="Center">
        <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="Center">
        <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="Center">
        <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>
                    </Binding.Converter>
                </Binding>
            </Label.Style>
        </Label>
    </StackLayout>

```

```

<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 this example, in the last of the three `Switch` and `Label` pairs, the generic argument is set to a `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:

Subscribe?  Of course!

Allow popups?  No

Learn more?  *Indubitably!*

NOTE

It's also possible to use triggers to implement changes in the user-interface based on other views. For more information, see [Triggers](#).

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 different conversions.

The use of the `ConverterParameter` property can be demonstrated with a color-selection program. The following example shows the `RgbColorViewModel`, which has three properties of type `float` 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 float Red
    {
        get { return color.Red; }
        set
        {
            if (color.Red != value)
            {

```

```

        Color = new Color(value, color.Green, color.Blue);
    }
}
}

public float Green
{
    get { return color.Green; }
    set
    {
        if (color.Green != value)
        {
            Color = new Color(color.Red, value, color.Blue);
        }
    }
}

public float Blue
{
    get { return color.Blue; }
    set
    {
        if (color.Blue != value)
        {
            Color = new Color(color.Red, color.Green, value);
        }
    }
}

public Color Color
{
    get { return 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);
        }
    }
}

public string Name
{
    get { return name; }
    private set
    {
        if (name != value)
        {
            name = value;
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Name"));
        }
    }
}
}

```

The `Red`, `Green`, and `Blue` property values can 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. Multiplying by 255 and converting to an integer can be performed 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 FloatToIntConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (int)Math.Round((float)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 float)
            return (float)parameter;
        else if (parameter is int)
            return (int)parameter;
        else if (parameter is string)
            return float.Parse((string)parameter);

        return 1;
    }
}
```

In this example, the `Convert` method converts from a `float` to `int` while multiplying by the `parameter` value. The `ConvertBack` method divides the integer `value` argument by `parameter` and returns a `float` result.

The type of the `parameter` argument is likely to be different depending on whether the data binding is defined in XAML or code. 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.

However, in XAML the `ConverterParameter` is likely to be set like this:

```
<Label Text="{Binding Red,
            Converter={StaticResource doubleToInt},
            ConverterParameter=255,
            StringFormat='Red = {0:X2}'}" />
```

While 255 looks like a number, because `ConverterParameter` is of type `Object`, the XAML parser treats 255 as a string. For this reason the value converter includes a separate `GetParameter` method that handles cases for `parameter` being of type `float`, `int`, or `string`.

The following XAML example instantiates `FloatToIntConverter` in its resource dictionary:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.RgbColorSelectorPage"
    Title="RGB Color Selector">
    <ContentPage.BindingContext>
        <local:RgbColorViewModel Color="Gray" />
    </ContentPage.BindingContext>
    <ContentPage.Resources>
        <Style TargetType="Slider">
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>

        <Style TargetType="Label">
            <Setter Property="HorizontalTextAlignment" Value="Center" />
        </Style>

        <local:FloatToIntConverter x:Key="floatToInt" />
    </ContentPage.Resources>

    <StackLayout Margin="20">
        <BoxView Color="{Binding Color}"
            HeightRequest="100"
            WidthRequest="100"
            HorizontalOptions="Center" />
        <StackLayout Margin="10, 0">
            <Label Text="{Binding Name}" />
            <Slider Value="{Binding Red}" />
            <Label Text="{Binding Red,
                Converter={StaticResource floatToInt},
                ConverterParameter=255,
                StringFormat='Red = {0:X2}'}" />
            <Slider Value="{Binding Green}" />
            <Label Text="{Binding Green,
                Converter={StaticResource floatToInt},
                ConverterParameter=255,
                StringFormat='Green = {0:X2}'}" />
            <Slider Value="{Binding Blue}" />
            <Label>
                <Label.Text>
                    <Binding Path="Blue"
                        StringFormat="Blue = {0:X2}"
                        Converter="{StaticResource floatToInt}">
                        <Binding.ConverterParameter>
                            <x:Single>255</x:Single>
                        </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 `float` value can be set to `ConverterParameter` property:



MediumVioletRed

Red = BF

Green = 3C

Blue = 7B

Relative bindings

9/20/2022 • 5 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) 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 .NET MAUI markup extensions, see [Consume 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`.

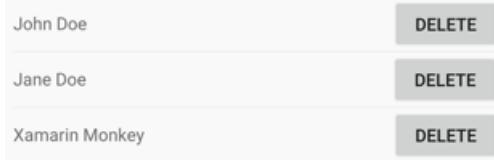
NOTE

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="End" />
                        </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:



John Doe

Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Nulla elit
dolor, convallis non interdum.



Jane Doe

Phasellus eu convallis mi. In tempus
augue eu dignissim fermentum. Morbi
ut lacus vitae eros lacinia.



Xamarin Monkey

Aliquam sagittis, odio lacinia
fermentum dictum, mi erat scelerisque
erat, quis aliquet arcu.

For more information about control templates, see [Control templates](#).

Binding fallbacks

9/20/2022 • 3 minutes to read • [Edit Online](#)

 [Browse 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. In a .NET Multi-platform App UI (.NET MAUI) app 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.

Define 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 following example demonstrates setting the `FallbackValue` property:

```
<Label Text="{Binding Population, FallbackValue='Population size unknown'}"  
... />
```

The binding on the `Label` defines a `FallbackValue` value (delimited by single-quote characters) 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.

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.

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:

Face-Palm Monkey

Location unknown

Population size unknown

Therefore, in this example 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.

Define 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 following example demonstrates 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 (delimited by single-quote characters) 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.

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.

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:



Baboon

Africa & Asia



Seated Monkey

Location unknown



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



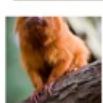
Squirrel Monkey

Central & South America



Face-Palm Monkey

Location unknown



Golden Lion Tamarin

Brazil

Therefore, in this example the `Image` and `Label` objects display their `TargetNullValue` when their source objects are `null`.

IMPORTANT

String formatting is not applied in a binding expression when the `TargetNullValue` property is set.

Multi-bindings

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) 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 app. 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. .NET MAUI 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 .NET MAUI not to perform any action. For example, to instruct .NET MAUI 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 typically 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://schemas.microsoft.com/dotnet/2021/maui"
    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>
```

NOTE

If the format string starts with the { character, the XAML parser will confuse it for a markup extension. To avoid this ambiguity, prefix the format string with an empty set of curly braces.

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 .NET MAUI, see [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 [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:

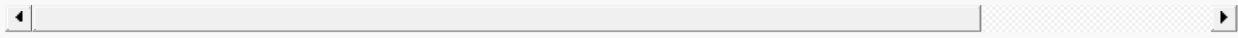
```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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">
    <ContentPage.Resources>
        <local:AllTrueMultiConverter x:Key="AllTrueConverter" />

        <ControlTemplate x:Key="CardViewExpanderControlTemplate">
            <local:Expander BindingContext="{Binding Source={RelativeSource TemplatedParent}}"
                IsExpanded="{Binding IsExpanded, Source={RelativeSource TemplatedParent}}"
                BackgroundColor="{Binding CardColor}"
                RowDefinitions="Auto,Auto"
                Padding="8">
                <local:Expander.IsVisible>
                    <MultiBinding Converter="{StaticResource AllTrueConverter}">
                        <Binding Path="IsExpanded" />
                        <Binding Path=".IsEnabled" />
                    </MultiBinding>
                </local:Expander.IsVisible>
                <Grid>
                    <!-- XAML that defines Expander header goes here -->
                </Grid>
                <Grid>
                    <!-- XAML that defines Expander content goes here -->
                </Grid>
            </local: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>

```

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 [Relative bindings](#). For more information about control templates, see [Control templates](#).

Commanding

9/20/2022 • 13 minutes to read • [Edit Online](#)

 [Browse the sample](#)

In a .NET Multi-platform App UI (.NET MAUI) app that uses the Model-View-ViewModel (MVVM) pattern, data bindings are defined between properties in the viewmodel, which is typically a class that derives from `IPropertyChanged`, and properties in the view, which is typically the XAML file. Sometimes an app 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 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 the `CommandParameter` property to arbitrary data to distinguish between multiple buttons if they are all bound to the same `ICommand` property in the viewmodel.

Many other views also define `Command` and `CommandParameter` properties. All these commands can be handled within a viewmodel using an approach that doesn't depend on the user-interface object in the view.

ICommands

The `ICommand` interface is defined 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 should contain properties of type `ICommand`:

```
public ICommand MyCommand { private set; get; }
```

The viewmodel must also reference a class that implements the `ICommand` interface. 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.

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 raised from within the viewmodel. When that event is raised, the `Button` calls `CanExecute` again. The `Button` enables itself if `CanExecute` returns `true` and disables itself if `CanExecute` returns `false`.

WARNING

Do not use the `IsEnabled` property of `Button` if you're using the command interface.

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 use the `Command` or `Command<T>` class included in .NET MAUI to implement the `ICommand` interface. These classes allow you to specify the bodies of the `Execute` and `CanExecute` methods in class constructors.

TIP

Use `Command<T>` when you use the `CommandParameter` property to distinguish between multiple views bound to the same `ICommand` property, and the `Command` class when that isn't a requirement.

Basic commanding

The following examples demonstrate basic commands implemented in a viewmodel.

The `PersonViewModel` class defines three properties named `Name`, `Age`, and `Skills` that define a person:

```

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` class shown below creates new objects of type `PersonViewModel` and allows the user to fill in the data. For that purpose, the class defines `IsEditing`, of type `bool`, and `PersonEdit`, of type `PersonViewModel`, properties. 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));
    }
}

```

In this example, changes to the three `ICommand` properties and the `Persons` property do not result in `PropertyChanged` events being raised. These properties are all set when the class is first created and do not change.

The following example shows the XAML that consumes the `PersonCollectionViewModel`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.PersonEntryPage"
    Title="Person Entry">
    <ContentPage.BindingContext>
        <local:PersonCollectionViewModel />
    </ContentPage.BindingContext>
    <Grid Margin="10">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <!-- New Button -->
    </Grid>

```

```

<Button Text="New"
        Grid.Row="0"
        Command="{Binding NewCommand}"
        HorizontalOptions="Start" />

<!-- Entry Form -->
<Grid Grid.Row="1"
      IsEnabled="{Binding IsEditing}"
      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="Center" />
    <Button Text="Cancel"
           Grid.Column="1"
           Command="{Binding CancelCommand}"
           VerticalOptions="Center" />
  </Grid>

<!-- List of Persons -->
<ListView Grid.Row="3"
          ItemsSource="{Binding Persons}" />
</Grid>
</ContentPage>

```

In this example, the page's `BindingContext` property is 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 `ListView` displays the collection of persons already entered:

The following screenshot shows the **Submit** button enabled after an age has been set:

NEW

Name: Barbara Zighetti

Age: - + 33 years old

Skills: C#, Android, Xamarin.Forms

SUBMIT **CANCEL**

Amy Strande, age 30

Andreas Hanefeld Dziegief, age 28

Arvind B. Rao, age 24

When 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.

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. This action and function can be defined as lambda functions in the `Command` constructor:

```

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`. A 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`:

```

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 its initial state.

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.

NOTE

It isn't necessary to define the `execute` and `canExecute` methods as lambda functions. You can write them as private methods in the viewmodel and reference them in the `Command` constructors. However, this approach can result in a lot of methods that are referenced only once in the.viewmodel.

Using Command parameters

It's 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 can 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 specifying a `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 following example demonstrates a keyboard for entering decimal numbers:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.DecimalKeypadPage"
    Title="Decimal Keyboard">
    <ContentPage.BindingContext>
        <local:DecimalKeypadViewModel />
    </ContentPage.BindingContext>
    <ContentPage.Resources>
        <Style TargetType="Button">
            <Setter Property="FontSize" Value="32" />
            <Setter Property="BorderWidth" Value="1" />
            <Setter Property="BorderColor" Value="Black" />
        </Style>
    </ContentPage.Resources>

    <Grid WidthRequest="240"
        HeightRequest="480"
        ColumnDefinitions="80, 80, 80"
        RowDefinitions="Auto, Auto, Auto, Auto, Auto"
        ColumnSpacing="2"
        RowSpacing="2"
        HorizontalOptions="Center"
        VerticalOptions="Center">
        <Label Text="{Binding Entry}"
            Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3"
            Margin="0,0,10,0"
            FontSize="32"
            LineBreakMode="HeadTruncation"
            VerticalTextAlignment="Center"
            HorizontalTextAlignment="End" />
        <Button Text="CLEAR"
            Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2"
            Command="{Binding ClearCommand}" />
        <Button Text="⇦" 
            Grid.Row="1" Grid.Column="2"
            Command="{Binding BackspaceCommand}" />
        <Button Text="7"
            Grid.Row="2" Grid.Column="0"
            Command="{Binding DigitCommand}"
            CommandParameter="7" />
        <Button Text="8"
```

```

        Grid.Row="2" Grid.Column="1"
        Command="{Binding DigitCommand}"
        CommandParameter="8" />
    <Button Text="9"
        Grid.Row="2" Grid.Column="2"
        Command="{Binding DigitCommand}"
        CommandParameter="9" />
    <Button Text="4"
        Grid.Row="3" Grid.Column="0"
        Command="{Binding DigitCommand}"
        CommandParameter="4" />
    <Button Text="5"
        Grid.Row="3" Grid.Column="1"
        Command="{Binding DigitCommand}"
        CommandParameter="5" />
    <Button Text="6"
        Grid.Row="3" Grid.Column="2"
        Command="{Binding DigitCommand}"
        CommandParameter="6" />
    <Button Text="1"
        Grid.Row="4" Grid.Column="0"
        Command="{Binding DigitCommand}"
        CommandParameter="1" />
    <Button Text="2"
        Grid.Row="4" Grid.Column="1"
        Command="{Binding DigitCommand}"
        CommandParameter="2" />
    <Button Text="3"
        Grid.Row="4" Grid.Column="2"
        Command="{Binding DigitCommand}"
        CommandParameter="3" />
    <Button Text="0"
        Grid.Row="5" Grid.Column="0" Grid.ColumnSpan="2"
        Command="{Binding DigitCommand}"
        CommandParameter="0" />
    <Button Text=" ."
        Grid.Row="5" Grid.Column="2"
        Command="{Binding DigitCommand}"
        CommandParameter="." />
</Grid>
</ContentPage>
```

In this example, the page's `BindingContext` 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 commands in the viewmodel: `ClearCommand`, `BackspaceCommand`, and `DigitCommand`. 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:

3.14159

CLEAR



7

8

9

4

5

6

1

2

3

0

.

The `DecimalKeypadViewModel` defines an `Entry` property of type `string` 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 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 **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";
                }
                RefreshCanExecutes();
            },
            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` is set to an instance of the `Command<T>` class. When using the commanding interface with XAML, the `CommandParameter` properties are usually strings, which is type of the generic argument. The `execute` and `canExecute` functions then have arguments of type `string`:

```

public class DecimalKeypadViewModel : INotifyPropertyChanged
{
    ...
    public DecimalKeypadViewModel()
    {
        ...
        DigitCommand = new Command<string>(
            execute: (string arg) =>
            {
                Entry += arg;
                if (Entry.StartsWith("0") && !Entry.StartsWith("0."))
                {
                    Entry = Entry.Substring(1);
                }
                RefreshCanExecutes();
            },
            canExecute: (string arg) =>
            {
                return !(arg == "." && Entry.Contains("."));
            });
        ...
    }
}

```

The `execute` method appends the string argument to the `Entry` property. However, if the result begins with a zero (but not a zero and a decimal point) then that initial zero must be removed using the `Substring` function. The `canExecute` method returns `false` only if the argument is the decimal point (indicating that the decimal point is being pressed) and `Entry` already contains a decimal point. All the `execute` methods call `RefreshCanExecutes`, which then calls `ChangeCanExecute` for both `DigitCommand` and `ClearCommand`. This ensures that the decimal point and backspace buttons are enabled or disabled based on the current sequence of entered

digits.

Compiled bindings

9/20/2022 • 7 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) data bindings have two main issues:

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 .NET MAUI 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. Ensure that XAML compilation is enabled. 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 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.

In addition, compiled bindings are currently unsupported on multi-bindings.

Use compiled bindings

The following example demonstrates using compiled bindings between .NET MAUI views and.viewmodel properties:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.CompiledColorSelectorPage"
    x:DataType="local:HslColorViewModel"
    Title="Compiled Color Selector">
    <ContentPage.BindingContext>
        <local:HslColorViewModel Color="Sienna" />
    </ContentPage.BindingContext>
    ...
    <StackLayout>
        <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>

```

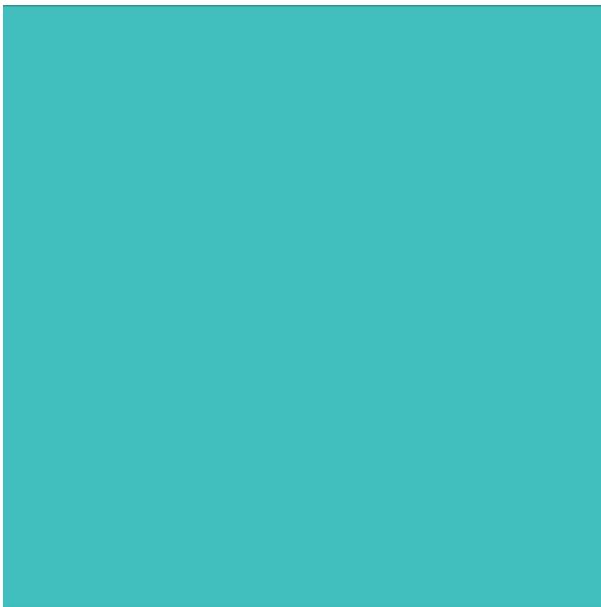
The `ContentPage` instantiates the `HslColorViewModel` and initializes the `Color` property within property element tags for the `BindingContext` property. The `ContentPage` also defines the `x:DataType` attribute as the viewmodel type, indicating that any binding expressions in the `ContentPage` view 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 `ContentPage`. 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 example 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. As the sliders are manipulated, the `BoxView` and `Label` elements are updated accordingly:



MediumTurquoise



Hue = 0.50



Saturation = 0.50



Luminosity = 0.50

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 following example demonstrates using compiled bindings in a `DataTemplate`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataBindingDemos"
    x:Class="DataBindingDemos.CompiledColorListPage"
    Title="Compiled Color List">

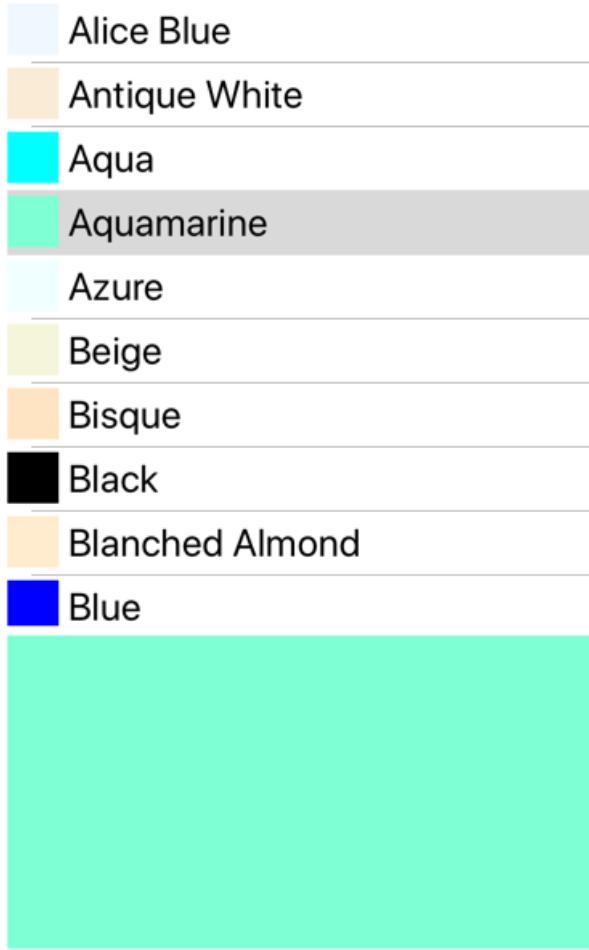
    <Grid>
        ...
        <ListView x:Name="colorListView"
            ItemsSource="{x:Static local:NamedColor.All}"
            ... >
            <ListView.ItemTemplate>
                <DataTemplate x:DataType="local:NamedColor">
                    <ViewCell>
                        <StackLayout 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 `Colors` class, 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.

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 [x:Type Markup Extension](#).

When the example 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 following example 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:

- 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.

Recognize a drag and drop gesture

9/20/2022 • 10 minutes to read • [Edit Online](#)

A .NET Multi-platform App UI (.NET MAUI) drag and drop gesture recognizer 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.

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.

IMPORTANT

On iOS a minimum platform of iOS 11 is required.

The process for enabling drag and drop in an app 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. .NET MAUI 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. .NET MAUI 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](#).

Enable drag

In .NET MAUI, 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 .NET MAUI 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

A drag gesture is initiated with a long-press followed by a drag.

Build a data package

.NET MAUI 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>	
<code>Entry</code>	<code>Text</code>	
<code>Label</code>	<code>Text</code>	
<code>RadioButton</code>	<code>.IsChecked</code>	<code>bool</code> converted to a <code>string</code> .
<code>Switch</code>	<code>IsToggled</code>	<code>bool</code> converted to a <code>string</code> .
<code>TimePicker</code>	<code>Time</code>	<code>TimeSpan</code> converted to a <code>string</code> .

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 raised 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 raised 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 .NET MAUI, 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`, that 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 .NET MAUI 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 .NET MAUI automatically copies dragged images, and text, to compatible drop targets.

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 raised 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 raised when a drag source is released over a drop target. When this occurs, .NET MAUI 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.

Recognize a pan gesture

9/20/2022 • 3 minutes to read • [Edit Online](#)

A .NET Multi-platform App UI (.NET MAUI) pan gesture recognizer detects the movement of fingers around the screen and can be used to apply that movement to content. A typical 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.

In .NET MAUI, pan gesture recognition is provided by the `PanGestureRecognizer` class. This class defines the `TouchPoints` property, of type `int`, which represents the number of touch points in the gesture. The default value of this property is 1. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

The `PanGestureRecognizer` class also defines a `PanUpdated` event that's raised when the detected pan gesture changes. The `PanUpdatedEventArgs` object that accompanies this event defines the following properties:

- `GestureId`, of type `int`, which represents the id of the gesture that raised the event.
- `StatusType`, of type `GestureStatus`, which indicates if the event has been raised for a newly started gesture, a running gesture, a completed gesture, or a canceled gesture.
- `TotalX`, of type `double`, which indicates the total change in the X direction since the beginning of the gesture.
- `TotalY`, of type `double`, which indicates the total change in the Y direction since the beginning of the gesture.

Create a PanGestureRecognizer

To make a `View` recognize a pan gesture, create a `PanGestureRecognizer` object, handle the `PanUpdated` event, and add the new gesture recognizer to the `GestureRecognizers` collection on the view. The following code example shows a `PanGestureRecognizer` attached to an `Image`:

```
<Image Source="monkey.jpg">
    <Image.GestureRecognizers>
        <PanGestureRecognizer PanUpdated="OnPanUpdated" />
    </Image.GestureRecognizers>
</Image>
```

The code for the `OnPanUpdated` event handler should be added to the code-behind file:

```
void OnPanUpdated(object sender, PanUpdatedEventArgs e)
{
    // Handle the pan
}
```

The equivalent C# code is:

```

PanGestureRecognizer panGesture = new PanGestureRecognizer();
panGesture.PanUpdated += (s, e) =>
{
    // Handle the pan
};
image.GestureRecognizers.Add(panGesture);

```

Create a pan container

Freeform panning is typically suited to navigating within images and maps. The `PanContainer` class, which is shown in the following example, is a generalized helper class that performs freeform panning:

```

public class PanContainer : ContentView
{
    double x, y;

    public PanContainer()
    {
        // Set PanGestureRecognizer.TouchPoints to control the
        // number of touch points needed to pan
        PanGestureRecognizer panGesture = new PanGestureRecognizer();
        panGesture.PanUpdated += OnPanUpdated;
        GestureRecognizers.Add(panGesture);
    }

    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 -
DeviceDisplay.MainDisplayInfo.Width));
                Content.TranslationY = Math.Max(Math.Min(0, y + e.TotalY), -Math.Abs(Content.Height -
DeviceDisplay.MainDisplayInfo.Height));
                break;

            case GestureStatus.Completed:
                // Store the translation applied during the pan
                x = Content.TranslationX;
                y = Content.TranslationY;
                break;
        }
    }
}

```

In this example, the `OnPanUpdated` method updates the viewable content of the wrapped view, 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 `DeviceDisplay.MainDisplayInfo.Width` and `DeviceDisplay.MainDisplayInfo.Height` properties provide the screen width and screen height values of the device. 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.

The `PanContainer` class can be wrapped around a `View` so that a recognized pan gesture will pan the wrapped view. The following XAML example shows the `PanContainer` wrapping an `Image`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:PanGesture"
    x:Class="PanGesture.MainPage">
    <AbsoluteLayout>
        <local:PanContainer>
            <Image Source="monkey.jpg" WidthRequest="1024" HeightRequest="768" />
        </local:PanContainer>
    </AbsoluteLayout>
</ContentPage>
```

In this example, when the `Image` receives a pan gesture, the displayed image will be panned.

Recognize a pinch gesture

9/20/2022 • 3 minutes to read • [Edit Online](#)

A .NET Multi-platform App UI (.NET MAUI) pinch gesture recognizer is used for performing interactive zoom. 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.

In .NET MAUI, pinch gesture recognition is provided by the `PinchGestureRecognizer` class, which defines a `PinchUpdated` event that's raised when the detected pinch gesture changes. The `PinchGestureEventArgs` object that accompanies the `PinchUpdated` event defines the following properties:

- `Scale`, of type `double`, which indicates the relative size of the pinch gesture since the last update was received.
- `ScaleOrigin`, of type `Point`, which indicates the updated origin of the pinch's gesture.
- `Status`, of type `GestureStatus`, which indicates if the event has been raised for a newly started gesture, a running gesture, a completed gesture, or a canceled gesture.

Create a PinchGestureRecognizer

To make a `View` recognize a pinch gesture, create a `PinchGestureRecognizer` object, handle the `PinchUpdated` event, and add the new gesture recognizer to the `GestureRecognizers` collection on the view. The following code example shows a `PinchGestureRecognizer` attached to an `Image`:

```
<Image Source="waterfront.jpg">
    <Image.GestureRecognizers>
        <PinchGestureRecognizer PinchUpdated="OnPinchUpdated" />
    </Image.GestureRecognizers>
</Image>
```

The code for the `OnPinchUpdated` event handler should be added to the code-behind file:

```
void OnPinchUpdated(object sender, PinchGestureEventArgs e)
{
    // Handle the pinch
}
```

The equivalent C# code is:

```
PinchGestureRecognizer pinchGesture = new PinchGestureRecognizer();
pinchGesture.PinchUpdated += (s, e) =>
{
    // Handle the pinch
};
image.GestureRecognizers.Add(pinchGesture);
```

Create a pinch container

The `PinchToZoomContainer` class, which is shown in the following example, is a generalized helper class that can be used to interactively zoom a `View`:

```

public class PinchToZoomContainer : ContentView
{
    double currentScale = 1;
    double startScale = 1;
    double xOffset = 0;
    double yOffset = 0;

    public PinchToZoomContainer()
    {
        PinchGestureRecognizer pinchGesture = new PinchGestureRecognizer();
        pinchGesture.PinchUpdated += OnPinchUpdated;
        GestureRecognizers.Add(pinchGesture);
    }

    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 = Math.Clamp(targetX, -Content.Width * (currentScale - 1), 0);
            Content.TranslationY = Math.Clamp(targetY, -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;
        }
    }
}

```

In this example, the `OnPinchUpdated` method updates the zoom level of the wrapped view, based on the user's pinch gesture. This is achieved by using the values of the `Scale`, `ScaleOrigin` and `Status` properties of the

`PinchGestureUpdatedEventArgs` object to calculate the scale factor to be applied at the origin of the pinch gesture. The wrapped view is then zoomed at the origin of the pinch gesture by setting its `TranslationX`, `TranslationY`, and `scale` properties to the calculated values.

The `PinchToZoomContainer` class can be wrapped around a `View` so that a recognized pinch gesture will zoom the wrapped view. The following XAML example shows the `PinchToZoomContainer` wrapping an `Image`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:PinchGesture;assembly=PinchGesture"
    x:Class="PinchGesture.HomePage">
    <Grid>
        <local:PinchToZoomContainer>
            <Image Source="waterfront.jpg" />
        </local:PinchToZoomContainer>
    </Grid>
</ContentPage>
```

In this example, when the `Image` receives a pinch gesture, the displayed image will be zoomed-in or out.

Recognize a swipe gesture

9/20/2022 • 4 minutes to read • [Edit Online](#)

A .NET Multi-platform App UI (.NET MAUI) swipe gesture recognizer detects when a finger is moved across the screen in a horizontal or vertical direction, and is often used to initiate navigation through content.

In .NET MAUI, drag gesture recognition is provided by the `SwipeGestureRecognizer` class. This class defines the following properties:

- `Command`, of type `ICommand`, which is executed when a swipe gesture is recognized.
- `CommandParameter`, of type `object`, which is the parameter that's passed to the `Command`.
- `Direction`, of type `SwipeDirection`, which defines the direction
- `Threshold`, of type `uint`, which 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, which means that any swipes that are less than 100 device-independent units will be ignored.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `SwipeGestureRecognizer` also defines a `Swiped` event that's raised when a swipe is recognized. The `SwipedEventArgs` object that accompanies the `Swiped` event defines the following properties:

- `Direction`, of type `SwipeDirection`, indicates the direction of the swipe gesture.
- `Parameter`, of type `object`, indicates the value passed by the `CommandParameter` property, if defined.

Create a SwipeGestureRecognizer

To make a `View` recognize a swipe gesture, create a `SwipeGestureRecognizer` object, 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 example shows a `SwipeGestureRecognizer` attached to a `BoxView`:

```
<BoxView Color="Teal" ...>
    <BoxView.GestureRecognizers>
        <SwipeGestureRecognizer Direction="Left" Swiped="OnSwiped"/>
    </BoxView.GestureRecognizers>
</BoxView>
```

The equivalent C# code is:

```
BoxView boxView = new BoxView { Color = Colors.Teal, ... };
SwipeGestureRecognizer leftSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Left };
leftSwipeGesture.Swiped += OnSwiped;

boxView.GestureRecognizers.Add(leftSwipeGesture);
```

Recognize the swipe direction

The `SwipeGestureRecognizer.Direction` property can be set to a single value from the `SwipeDirection` enumeration, or multiple values. This enables the `Swiped` event to be raised 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`:

```
SwipeGestureRecognizer 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>
```

The equivalent C# code is:

```
BoxView boxView = new BoxView { Color = Colors.Teal, ... };  
SwipeGestureRecognizer leftSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Left };  
leftSwipeGesture.Swiped += OnSwiped;  
SwipeGestureRecognizer rightSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Right };  
rightSwipeGesture.Swiped += OnSwiped;  
SwipeGestureRecognizer upSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Up };  
upSwipeGesture.Swiped += OnSwiped;  
SwipeGestureRecognizer 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);
```

Respond to a swipe

A recognized swipe can be responded to by a handler for the `Swiped` event:

```

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.

Create a swipe container

The `SwipeContainer` class, which is shown in the following 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)
    {
        SwipeGestureRecognizer 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`:

```
<StackLayout>
    <local:SwipeContainer Swipe="OnSwiped" ...>
        <BoxView Color="Teal" ... />
    </local:SwipeContainer>
</StackLayout>
```

In this example, when the `BoxView` receives a swipe gesture, the `swiped` event in the `SwipeGestureRecognizer` is raised. This is handled by the `SwipeContainer` class, which raises 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.

The equivalent C# code is:

```
BoxView boxView = new BoxView { Color = Colors.Teal, ... };
SwipeContainer swipeContainer = new SwipeContainer { Content = boxView, ... };
swipeContainer.Swipe += (sender, e) =>
{
    // Handle the swipe
};

StackLayout stackLayout = new StackLayout();
stackLayout.Add(swipeContainer);
```

Recognize a tap gesture

9/20/2022 • 2 minutes to read • [Edit Online](#)

A .NET Multi-platform App UI (.NET MAUI) tap gesture recognizer is used for tap detection and is implemented with the `TapGestureRecognizer` class. This class defines the following properties:

- `Command`, of type `ICommand`, which is executed when a tap is recognized.
- `CommandParameter`, of type `object`, which is the parameter that's passed to the `Command`.
- `NumberOfTapsRequired`, of type `int`, which represents the number of taps required to recognize a tap gesture. The default value of this property is 1.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `TapGestureRecognizer` class also defines a `Tapped` event that's raised when a tap is recognized. The `TappedEventArgs` object that accompanies the `Tapped` event defines a `Parameter` property of type `object` that indicates the value passed by the `CommandParameter` property, if defined.

Create a TapGestureRecognizer

To make a `View` recognize a tap gesture, create a `TapGestureRecognizer` object, handle the `Tapped` event, and add the new gesture recognizer to the `GestureRecognizers` collection on the view. The following code example shows a `TapGestureRecognizer` attached to an `Image`:

```
<Image Source="tapped.jpg">
    <Image.GestureRecognizers>
        <TapGestureRecognizer Tapped="OnTapGestureRecognizerTapped"
            NumberOfTapsRequired="2" />
    </Image.GestureRecognizers>
</Image>
```

The code for the `OnTapGestureRecognizerTapped` event handler should be added to the code-behind file:

```
void OnTapGestureRecognizerTapped(object sender, EventArgs args)
{
    // Handle the tap
}
```

The equivalent C# code is:

```
TapGestureRecognizer tapGestureRecognizer = new TapGestureRecognizer();
tapGestureRecognizer.Tapped += (s, e) =>
{
    // Handle the tap
};
image.GestureRecognizers.Add(tapGestureRecognizer);
```

By default the `Image` will respond to single taps. When the `NumberOfTapsRequired` property is set above one, the event handler will only be executed if the taps occur within a set period of time. If the second (or subsequent) taps don't occur within that period, they're effectively ignored.

Bindable properties

9/20/2022 • 7 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) bindable properties extend Common Language Runtime (CLR) property functionality by backing a property with a `BindableProperty` type, instead of 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.

In .NET MAUI apps, 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 .NET MAUI 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.

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 UI 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 IsExpandedProperty =
    BindableProperty.Create ("IsExpanded", typeof(bool), typeof(Expander), false);
```

This creates a `BindableProperty` instance named `IsExpandedProperty`, of type `bool`. The property is owned by the `Expander` class, and has a default value of `false`.

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 `IsExpanded` bindable property:

```
public bool IsExpanded
{
    get => (bool)GetValue(IsExpandedProperty);
    set => SetValue(IsExpandedProperty, 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:DataBindingDemos" ...>
...
</ContentPage>
```

The namespace declaration is used when setting the `IsExpanded` bindable property, as demonstrated in the following XAML code example:

```
<Expander IsExpanded="true">
...
</Expander>
```

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

```
Expander expander = new Expander
{
    IsExpanded = true
};
```

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 `IsExpanded` bindable property registers the `OnIsExpandedChanged` method as a property-changed callback method:

```
public static readonly BindableProperty IsExpandedProperty =
    BindableProperty.Create(nameof(IsExpanded), typeof(bool), typeof(Expander), false, PropertyChanged:
        OnIsExpandedChanged);
...

static void OnIsExpandedChanged (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(MainPage), 0.0, validateValue: IsValidValue);
...
static bool IsValidValue(BindableObject view, object value)
{
    double result;
    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 you should handle. 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(MainPage), 0.0, coerceValue: CoerceAngle);
public static readonly BindableProperty MaximumAngleProperty =
    BindableProperty.Create("MaximumAngle", typeof(double), typeof(MainPage), 360.0, propertyChanged:
ForceCoerceValue);
...

static object CoerceAngle(BindableObject bindable, object value)
{
    MainPage page = bindable as MainPage;
    double input = (double)value;

    if (input > page.MaximumAngle)
    {
        input = page.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 example:

```

public static readonly BindableProperty DateProperty =
    BindableProperty.Create ("Date", typeof(DateTime), typeof(MyPage), default(DateTime),
BindingMode.TwoWay, defaultValueCreator: bindable => DateTime.Today);

```

The `defaultValueCreator` parameter is set to a `Func` that returns a `DateTime` that represents today's date.

Attached properties

9/20/2022 • 4 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) 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` layout enables 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(Shadow), false);
```

This creates an attached property named `HasShadowProperty`, of type `bool`. The property is owned by the `Shadow` 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 example demonstrates a XAML namespace for a custom type that contains an attached property, which is defined within the same assembly as the app code that's referencing the custom type:

```
<ContentPage ... xmlns:local="clr-namespace:ShadowDemo" ...>
...
</ContentPage>
```

The namespace declaration is then used when setting the attached property on a specific control, as demonstrated in the following XAML:

```
<Label Text="Label with shadow" local:Shadow.HasShadow="true" />
```

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

```
Label label = new Label { Text = "Label with shadow" };
Shadow.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 for `Label` controls that uses the `HasShadow` attached property:

```
<Style x:Key="ShadowStyle" TargetType="Label">
  <Style.Setters>
    <Setter Property="local:Shadow.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 with shadow" Style="{StaticResource ShadowStyle}" />
```

For more information about styles, see [Styles](#).

Advanced scenarios

When creating an attached property, there are some 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](#).

Publish and subscribe to messages

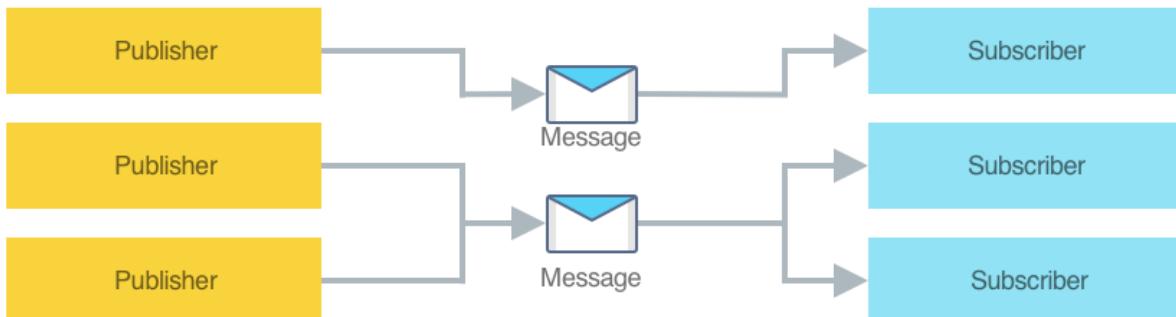
9/20/2022 • 4 minutes to read • [Edit Online](#)

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 .NET Multi-platform App UI (.NET MAUI) `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`.

Resource dictionaries

9/20/2022 • 8 minutes to read • [Edit Online](#)

A .NET Multi-platform App UI (.NET MAUI) `ResourceDictionary` is a repository for resources that are used by a .NET MAUI app. Typical resources that are stored in a `ResourceDictionary` include styles, control templates, data templates, converters, and colors.

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.

TIP

In Visual Studio, a XAML-based `ResourceDictionary` file that's backed by a code-behind file can be added to your project by the [.NET MAUI ResourceDictionary \(XAML\)](#) item template.

Create resources

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 .NET MAUI app can contain only a single class that derives from `Application`, but often makes use of many classes that derive from `visualElement`, including pages, layouts, and views. 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 app.

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://schemas.microsoft.com/dotnet/2021/maui"
    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="NavigationPage">
            <Setter Property="BarBackgroundColor"
                Value="{StaticResource NavigationBarColor}" />
            <Setter Property="BarTextColor"
                Value="{StaticResource NavigationBarTextColor}" />
        </Style>

        <Style TargetType="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.

IMPORTANT

Inserting resources directly between the `Resources` property-element tags automatically creates a `ResourceDictionary` object. However, it's also valid to place all resources between optional `ResourceDictionary` tags.

Consume resources

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` XAML 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 app. For more information about markup extensions, see [XAML markup extensions](#).

The following XAML example shows how to consume resources, and also define an additional resource in a `StackLayout`:

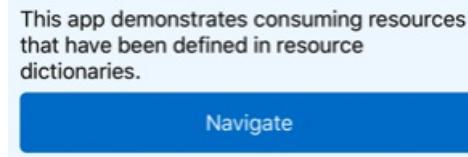
```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ResourceDictionaryDemo.MainPage"
    Title="Main page">
    <StackLayout Margin="{StaticResource PageMargin}"
        Spacing="6">
        <StackLayout.Resources>
            <!-- Implicit style -->
            <Style TargetType="Button">
                <Setter Property="FontSize" Value="14" />
                <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 screenshot:



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 app startup instead of when required by a page.

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 still isn't 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 layout or view level `AppBackgroundColor` resource.

Stand-alone resource dictionaries

A `ResourceDictionary` can also be created as a stand-alone XAML file that isn't backed by a code-behind file. To create a stand-alone `ResourceDictionary`, add a new `ResourceDictionary` file to the project with the **.NET MAUI ResourceDictionary (XAML)** item template and delete its code-behind file. Then, in the XAML file remove the `x:Class` attribute from the `ResourceDictionary` tag near the start of the file. In addition, add `<?xml-compile="true" ?>` after the XML header to ensure that the XAML will be compiled.

NOTE

A stand-alone `ResourceDictionary` must have a build action of **MauiXaml**.

The following XAML example shows a stand-alone `ResourceDictionary` named `MyResourceDictionary.xaml`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-compile="true" ?>
<ResourceDictionary xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
                     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml">
    <DataTemplate x:Key="PersonDataTemplate">
        <ViewCell>
            <Grid RowSpacing="6"
                  ColumnSpacing="6">
                <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.

Merge resource dictionaries

Resource dictionaries can be combined by merging 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 `ResourceDictionary.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 `MauiXaml`, 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:DefaultTheme />
                <!-- 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, .NET MAUI 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.

TIP

Searching resource dictionaries can be a computationally intensive task if an app 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.

.NET MAUI Shell overview

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) Shell reduces the complexity of app development by providing the fundamental features that most apps require, including:

- A single place to describe the visual hierarchy of an app.
- A common navigation user experience.
- A URI-based navigation scheme that permits navigation to any page in the app.
- An integrated search handler.

App visual hierarchy

In a .NET MAUI Shell app, the visual hierarchy of the app 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 app requires a flyout. A `TabBar` represents the bottom tab bar, and should be used when the navigation pattern for the app 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 app's visual hierarchy. Shell will take these objects and produce the navigation user interface for the content.

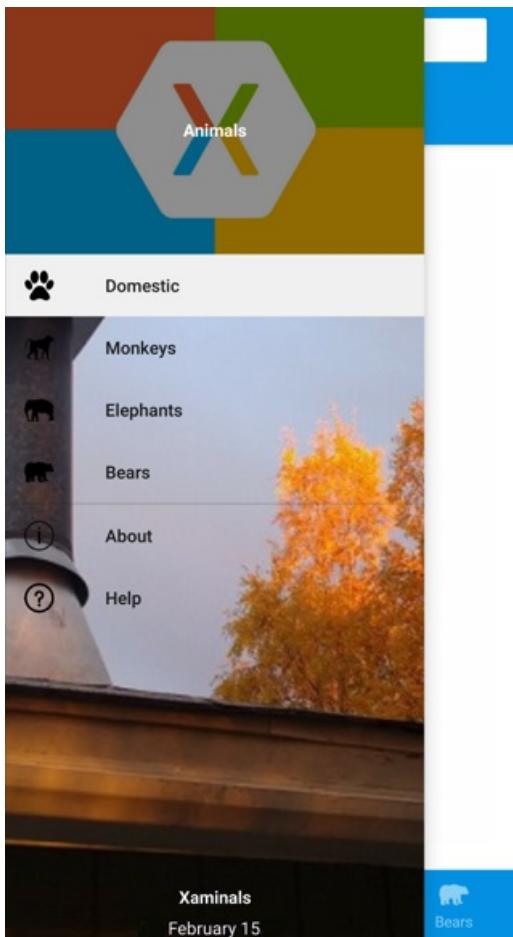
NOTE

Pages are created on demand in Shell apps, in response to navigation.

For more information, see [Create a .NET MAUI Shell app](#).

Navigation user experience

The navigation experience provided by .NET MAUI Shell is based on flyouts and tabs. The top level of navigation in a Shell app is either a flyout or a bottom tab bar, depending on the navigation requirements of the app. The following example shows an app where the top level of navigation is a flyout:



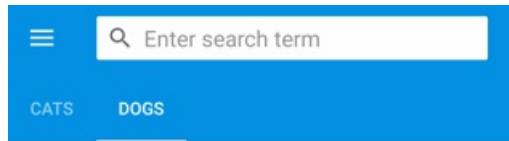
In this example, some flyout items are duplicated as tab bar items. However, there are also items that can only be accessed from the flyout. Selecting a flyout item results in the bottom tab that represents the item being selected and displayed:

A screenshot of a mobile application interface. At the top is a search bar with the placeholder 'Enter search term' and a magnifying glass icon. To the left of the search bar is a menu icon (three horizontal lines). Below the search bar are two tabs: 'CATS' and 'DOGS'. The 'CATS' tab is selected, indicated by a blue background. The main content area displays a list of cat breeds, each with a small thumbnail image, the breed name, and its origin. The breeds listed are: Abyssinian (Ethopia), Arabian Mau (Arabian Peninsula), Bengal (Asia), Burmese (Thailand), Cyprus (Cyprus), German Rex (Germany), and Highlander (United States). At the bottom of the screen is a horizontal tab bar with four items: 'Domestic' (paw print icon), 'Monkeys', 'Elephants', and 'Bears'. The 'Domestic' tab is currently selected, matching the blue color of the 'CATS' tab.

NOTE

When the flyout isn't open the bottom tab bar can be considered to be the top level of navigation in the app.

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:



Afghan Hound

Afghanistan



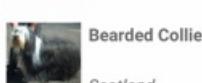
Alpine Dachsbracke

Austria



American Bulldog

United States



Bearded Collie

Scotland



Boston Terrier

United States



Canadian Eskimo

Canada



Eurohound

Scandinavia



Within each tab, additional `ContentPage` objects that are known as detail pages, can be navigated to:

Irish Terrier

Ireland



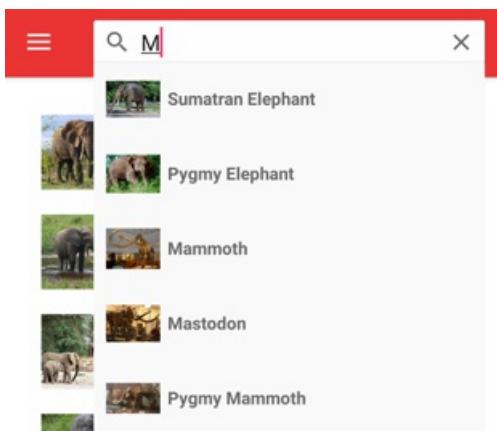
The Irish Terrier is a dog breed from Ireland, one of many breeds of terrier. The Irish Terrier is considered one of the oldest terrier breeds. The Dublin dog show in 1873 was the first to provide a separate class for Irish Terriers. By the 1880s, Irish Terriers were the fourth most popular breed in Ireland and Britain.



Shell uses a URI-based navigation experience that uses routes to navigate to any page in the app, 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 [.NET MAUI Shell navigation](#).

Search

.NET MAUI 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 [.NET MAUI Shell search](#).

Create a .NET MAUI Shell app

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

A .NET Multi-platform App UI (.NET MAUI) Shell app can be created with the **.NET MAUI App** project template, and then by describing the visual hierarchy of the app in the `AppShell` class.

Describe the visual hierarchy of the app

The visual hierarchy of a .NET MAUI Shell app is described in the subclassed `Shell` class, which the project template names `AppShell`. 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 app requires a flyout. A `TabBar` represents the bottom tab bar, and should be used when the navigation pattern for the app 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 app's visual hierarchy. Shell will take these objects and produce the navigation user interface for the content.

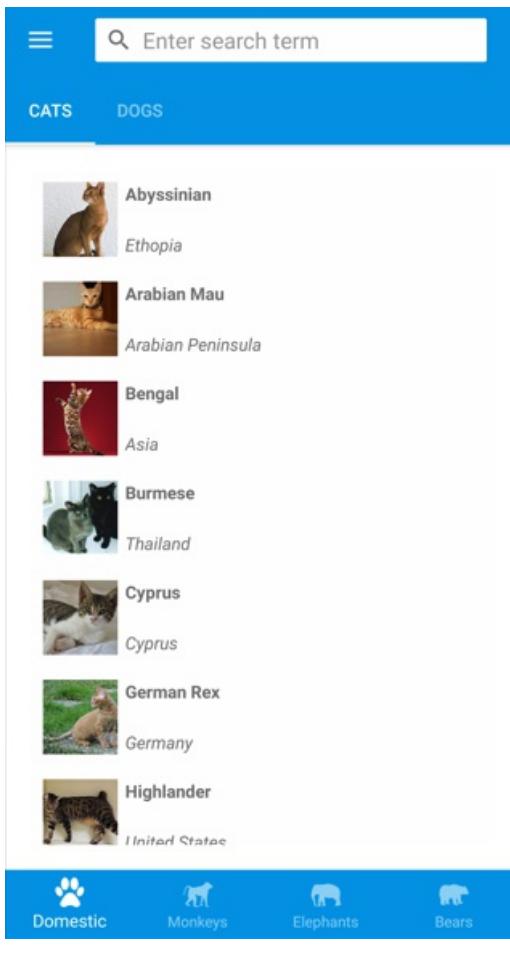
The following XAML shows an example of a subclassed `Shell` class:

```

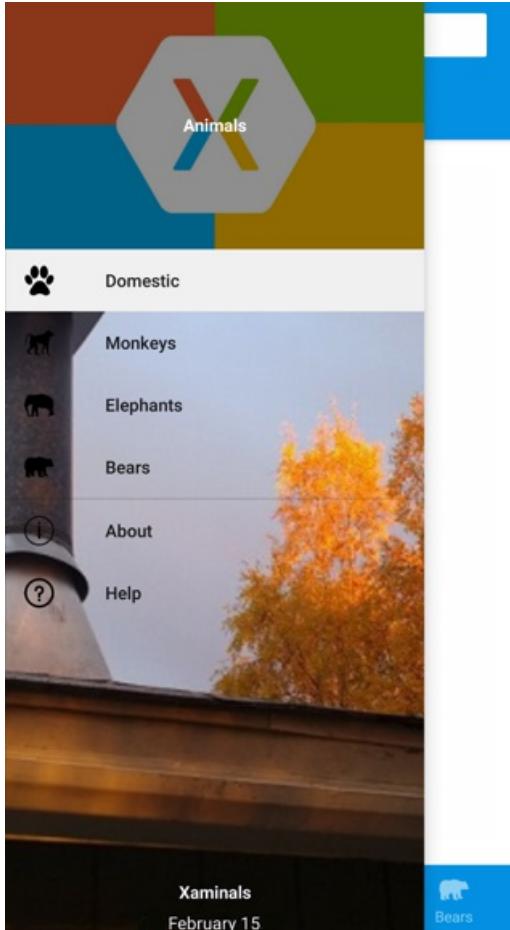
<Shell xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Xaminals.Views"
       x:Class="Xaminals.AppShell">
    ...
    <FlyoutItem FlyoutDisplayOptions="AsMultipleItems">
        <Tab Title="Domestic"
             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

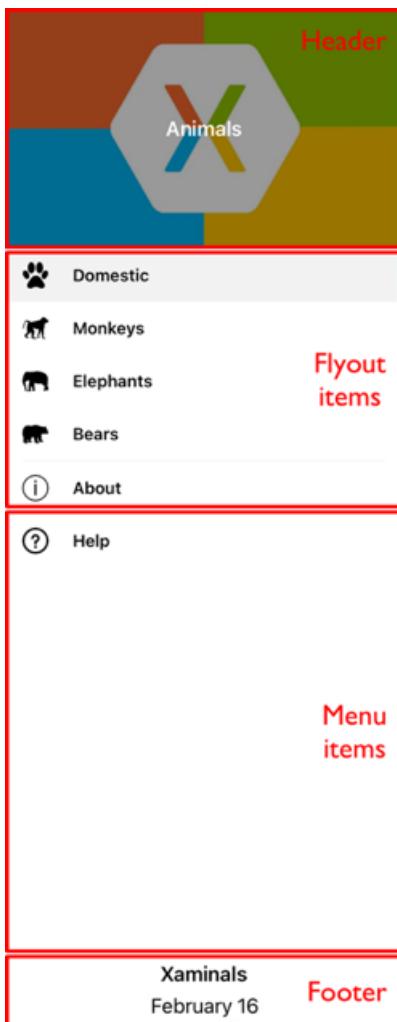
In a Shell app, pages are created on demand in response to navigation. This is accomplished by using the `DataTemplate` markup extension to set the `ContentTemplate` property of each `ShellContent` object to a `ContentPage` object.

.NET MAUI Shell flyout

9/20/2022 • 15 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The navigation experience provided by .NET Multi-platform App UI (.NET MAUI) Shell is based on flyouts and tabs. A flyout is the optional root menu for a Shell app, 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://schemas.microsoft.com/dotnet/2021/maui"
       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 app, 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://schemas.microsoft.com/dotnet/2021/maui"
       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 direct 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://schemas.microsoft.com/dotnet/2021/maui"
       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>

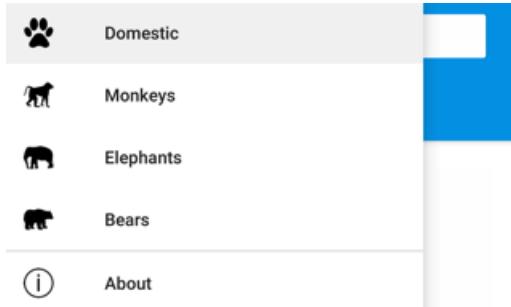
    <ShellContent Title="About"
                  Icon="info.png"
                  ContentTemplate="{DataTemplate views:AboutPage}" />
</Shell>
```

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="{OnPlatform 44, Android=50}"
        ColumnSpacing="{OnPlatform WinUI=0}"
        RowSpacing="{OnPlatform WinUI=0}">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroupList>
                <VisualStateGroup x:Name="CommonStates">
                    <VisualState x:Name="Normal">
                        <Setter Property="BackgroundColor"
                            Value="Transparent" />
                    </VisualState>
                    <VisualState x:Name="Selected">
                        <VisualState.Setters>
                            <Setter Property="BackgroundColor"
                                Value="{AppThemeBinding Light=Black, Dark=White}" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateGroupList>
        </VisualStateManager.VisualStateGroups>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="{OnPlatform Android=54, iOS=50, WinUI=Auto}" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Image x:Name="FlyoutItemImage"
            Source="{Binding FlyoutIcon}"
            VerticalOptions="Center"
            HorizontalOptions="{OnPlatform Default=Center, WinUI=Start}"
            HeightRequest="{OnPlatform Android=24, iOS=22, WinUI=16}"
            WidthRequest="{OnPlatform Android=24, iOS=22, WinUI=16}">
            <Image.Margin>
                <OnPlatform x:TypeArguments="Thickness">
                    <OnPlatform.Platforms>
                        <On Platform="WinUI"
                            Value="12,0,12,0" />
                    </OnPlatform.Platforms>
                </OnPlatform>
            </Image.Margin>
        </Image>
        <Label x:Name="FlyoutItemLabel"
            Grid.Column="1"
            Text="{Binding Title}"
            FontSize="{OnPlatform Android=14, iOS=14}"
            FontAttributes="{OnPlatform iOS=Bold}"
            HorizontalOptions="{OnPlatform WinUI=Start}"
            HorizontalTextAlignment="{OnPlatform WinUI=Start}"
            VerticalTextAlignment="Center">
            <Label.TextColor>
                <OnPlatform x:TypeArguments="Color">
                    <OnPlatform.Platforms>
                        <On Platform="Android"
                            Value="{AppThemeBinding Light=Black, Dark=White}" />
                    </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>

```

```
        </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 ...
    x:Name="shell">
...
<Shell.FlyoutContent>
    <CollectionView BindingContext="{x:Reference shell}"
        IsGrouped="True"
        ItemsSource="{Binding FlyoutItems}">
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <Label Text="{Binding Title}"
                    TextColor="White"
                    FontSize="18" />
            </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 ...>
    x:Name="shell">
    ...
    <Shell.FlyoutContentTemplate>
        <DataTemplate>
            <CollectionView BindingContext="{x:Reference shell}"
                IsGrouped="True"
                ItemsSource="{Binding FlyoutItems}">
                <CollectionView.ItemTemplate>
                    <DataTemplate>
                        <Label Text="{Binding Title}"
                            TextColor="White"
                            FontSize="18" />
                    </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.

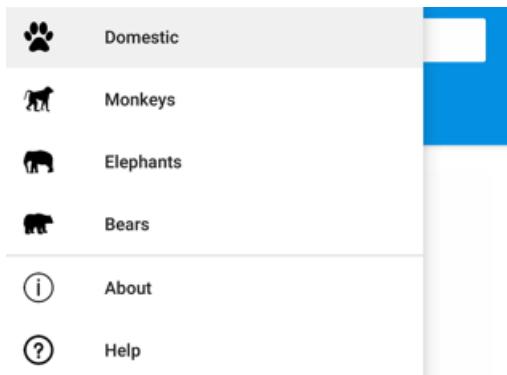
`MenuItem` objects can be added to the flyout as shown in the following example:

```

<Shell ...>
    ...
    <MenuItem Text="Help"
        IconImageSource="help.png"
        Command="{Binding HelpCommand}"
        CommandParameter="https://docs.microsoft.com/dotnet/maui/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

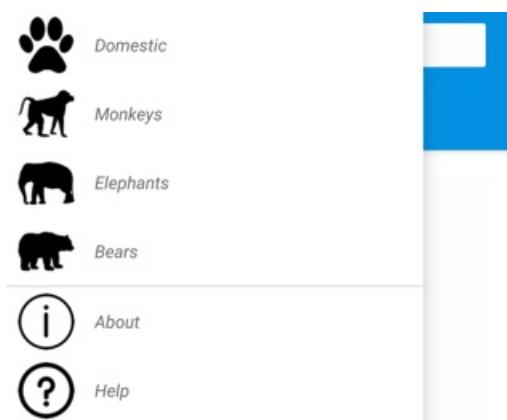
The `BindingContext` of each `MenuItem` is inherited from the subclassed `Shell` object.

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 `DataTemplate` 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 [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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Xaminals.Controls.FlyoutHeader"
    HeightRequest="200">
    <Grid BackgroundColor="Black">
        <Image Aspect="AspectFill"
            Source="store.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="store.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:

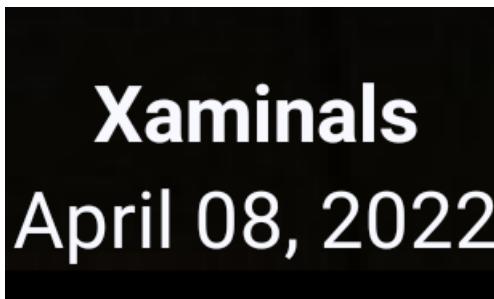
```
<ContentView xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:sys="clr-namespace:System;assembly=netstandard"
    x:Class="Xaminals.Controls.FlyoutFooter">
    <StackLayout>
        <Label Text="Xaminals"
            TextColor="GhostWhite"
            FontAttributes="Bold"
            HorizontalOptions="Center" />
        <Label Text="{Binding Source={x:Static sys:DateTime.Now}, StringFormat='{0:MMMM dd, yyyy}'}"
            TextColor="GhostWhite"
            HorizontalOptions="Center" />
    </StackLayout>
</ContentView>
```

IMPORTANT

The previous XAML example defined a new XAML namespace named `sys`:

`xmlns:sys="clr-namespace:System;assembly=netstandard"`. This XAML namespace maps `sys` to the .NET `System` namespace. The mapping allows you to use the .NET types defined in that namespace, such as `DateTime`, in the XAML. For more information, see [XAML Namespaces](#).

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 apps 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 [.NET MAUI 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 [.NET MAUI 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.

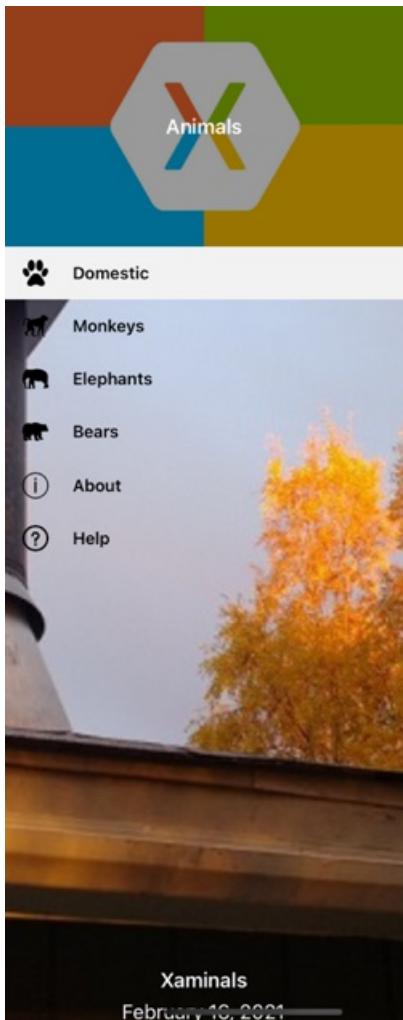
The aspect ratio of the background image can be configured by setting the `FlyoutBackgroundImageAspect` bindable property, of type `Aspect`, to one of the `Aspect` enumeration members:

- `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 [.NET MAUI 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 selection

When a Shell app 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 app 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;
```

.NET MAUI Shell tabs

9/20/2022 • 5 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The navigation experience provided by .NET Multi-platform App UI (.NET MAUI) Shell is based on flyouts and tabs. The top level of navigation in a Shell app is either a flyout or a bottom tab bar, depending on the navigation requirements of the app. When the navigation experience for an app 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 are navigable by top tabs. Within a tab, you can navigate to other `ContentPage` objects that are known as detail pages.

IMPORTANT

The `TabBar` type disables the flyout.

Single page

A single page Shell app 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://schemas.microsoft.com/dotnet/2021/maui"
      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>
```

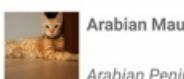
This example results in the following single page app:

Cats



Abyssinian

Ethopia



Arabian Mau

Arabian Peninsula



Bengal

Asia



Burmese

Thailand



Cyprus

Cyprus



German Rex

Germany



Highlander

United States



Manx

Isle of Man

Shell has implicit-conversion operators that enable the Shell visual hierarchy to be simplified, without introducing more 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://schemas.microsoft.com/dotnet/2021/maui"
      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 app, 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

If there are multiple `Tab` objects in a single `TabBar` object, `Tab` objects are rendered as bottom tabs:

```

<Shell xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       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:

Dogs

	Afghan Hound
	Afghanistan
	Alpine Dachsbracke
	Austria
	American Bulldog
	United States
	Bearded Collie
	Scotland
	Boston Terrier
	United States
	Canadian Eskimo
	Canada
	Eurohound
	Scandinavia
	Irish Terrier

Cats Dogs

When there are more than five tabs on a `TabBar`, a **More** tab will appear, which can be used to access the other 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://schemas.microsoft.com/dotnet/2021/maui"
       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 app, 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://schemas.microsoft.com/dotnet/2021/maui"
       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 code results in the layout shown in the following screenshot:



Abyssinian

Ethopia



Arabian Mau

Arabian Peninsula



Bengal

Asia



Burmese

Thailand



Cyprus

Cyprus



German Rex

Germany



Highlander

United States



Domestic



Monkeys

In addition, Shell's implicit conversion operators can be used to remove the second `Tab` object from the previous example:

```
<Shell xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       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 [.NET MAUI Shell specific properties](#).

Tab selection

When a Shell app 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.SelectedItem = dogsItem;
```

TabBar and Tab visibility

The tab bar and tabs are visible in Shell apps 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 `isVisible` 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.

.NET MAUI Shell pages

9/20/2022 • 6 minutes to read • [Edit Online](#)

 [Browse 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 .NET Multi-platform App UI (.NET MAUI) Shell apps. 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 .NET MAUI Shell apps, 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://schemas.microsoft.com/dotnet/2021/maui"
      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:

Cats



Abyssinian

Ethopia



Arabian Mau

Arabian Peninsula



Bengal

Asia



Burmese

Thailand



Cyprus

Cyprus



German Rex

Germany



Highlander

United States



Manx



Cats



Dogs



Monkeys

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 [.NET MAUI Shell navigation](#).

Load pages at app startup

In a Shell app, each `ContentPage` object is typically created on demand, in response to navigation. However, it's also possible to create `ContentPage` objects at app startup.

WARNING

`ContentPage` objects that are created at app startup can lead to a poor startup experience.

`ContentPage` objects can be created at app startup by setting the `shellContent.Content` properties to `ContentPage` objects:

```

<Shell xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
       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 app 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 app:

- `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 [.NET MAUI 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://schemas.microsoft.com/dotnet/2021/maui"
       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 app, 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 [Style apps using XAML](#).

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 `true` on Android, and `false` on other platforms.

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>
```

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="logo.png"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
    </Shell.TitleView>
    ...
</ContentPage>
```

IMPORTANT

If the navigation bar has been made invisible, with the `NavBarIsVisible` 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.

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 app and it won't be possible to navigate to it.

.NET MAUI Shell navigation

9/20/2022 • 15 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) Shell includes a URI-based navigation experience that uses routes to navigate to any page in the app, 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 app by using the `Navigation` property. For more information, see [Perform modeless navigation](#).

Routes

Navigation is performed in a Shell app 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 app. 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 app 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 app 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 app. 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 app 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:

- ▷ 🔐 `currentItem` Title = null, Route = "animals"
- ▷ 🔐 `currentState` Location = {/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 app.

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
Current	ShellNavigationState	The URI of the current page.
Previous	ShellNavigationState	The URI of the previous page.
Source	ShellNavigationSource	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

Primitive data can be passed as string-based 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:

```

async void OnCollectionViewSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    string elephantName = (e.CurrentSelection.FirstOrDefault() as Animal).Name;
    await Shell.Current.GoToAsync($"elephantdetails?name={elephantName}");
}

```

This example retrieves the currently selected elephant in the `CollectionView`, and navigates to the `elephantdetails` route, passing `elephantName` as a query parameter.

Object-based navigation data can be passed with a `GoToAsync` overload that specifies an `IDictionary<string, object>` argument:

```

async void OnCollectionViewSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    Animal animal = e.CurrentSelection.FirstOrDefault() as Animal;
    var navigationParameter = new Dictionary<string, object>
    {
        { "Bear", animal }
    };
    await Shell.Current.GoToAsync($"beardetails", navigationParameter);
}

```

This example retrieves the currently selected bear in the `CollectionView`, as an `Animal`. The `Animal` object is added to a `Dictionary` with the key `Bear`. Then, navigation to the `beardetails` route is performed, with the

`Dictionary` being passed as a navigation 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 string-based query parameter and object-based navigation parameter:

```
[QueryProperty(nameof(Bear), "Bear")]
public partial class BearDetailPage : ContentPage
{
    Animal bear;
    public Animal Bear
    {
        get => bear;
        set
        {
            bear = value;
            OnPropertyChanged();
        }
    }

    public BearDetailPage()
    {
        InitializeComponent();
        BindingContext = this;
    }
}
```

In this example the first argument for the `QueryPropertyAttribute` specifies the name of the property that will receive the data, with the second argument specifying the parameter id. Therefore, the `QueryPropertyAttribute` in the above example specifies that the `Bear` property will receive the data passed in the `Bear` navigation parameter in the `GoToAsync` method call.

NOTE

String-based 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, object>`, that contains any data passed during navigation. Each key in the dictionary is a query parameter id, with its value corresponding to the object that represents the data. 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, object> query)
    {
        Monkey = query["Monkey"] as Animal;
        OnPropertyChanged("Monkey");
    }
    ...
}

```

In this example, the `ApplyQueryAttributes` method retrieves the object that corresponds to the `Monkey` key in the `query` dictionary, which was passed as an argument to the `GoToAsync` method call.

IMPORTANT

String-based query parameter values that are received via the `IQueryAttributable` interface aren't automatically URL decoded.

Pass and process multiple items of data

Multiple string-based 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 string-based 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, object> query)
    {
        string name = HttpUtility.UrlDecode(query["name"].ToString());
        string location = HttpUtility.UrlDecode(query["location"].ToString());
        ...
    }
    ...
}
```

In this example, the `ApplyQueryAttributes` method retrieves the value of the `name` and `location` query parameters from the URI in the `GoToAsync` method call.

NOTE

String-based query parameters and object-based navigation parameters can be simultaneously passed when performing route-based navigation.

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`.
- `IsVisible`, of type `boolean`, indicates whether the back button is visible. 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 `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:



.NET MAUI Shell search

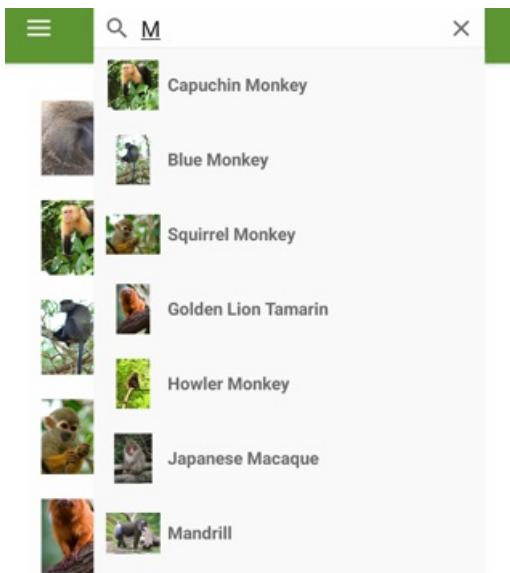
9/20/2022 • 5 minutes to read • [Edit Online](#)

{ } [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) 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 app 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 app.
        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 [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:

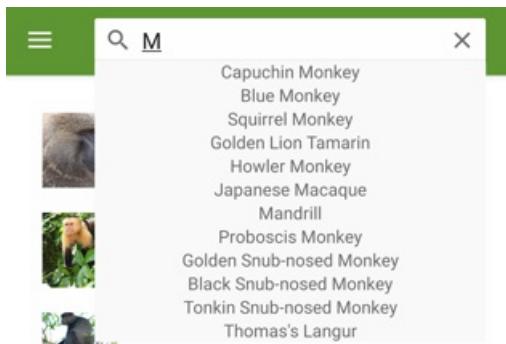
```

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:



Howler Monkey

South America



Howler monkeys are among the largest of the New World monkeys. Fifteen species are currently recognised. Previously classified in the family Cebidae, they are now placed in the family Atelidae.



Domestic



Monkeys



Elephants



Bears

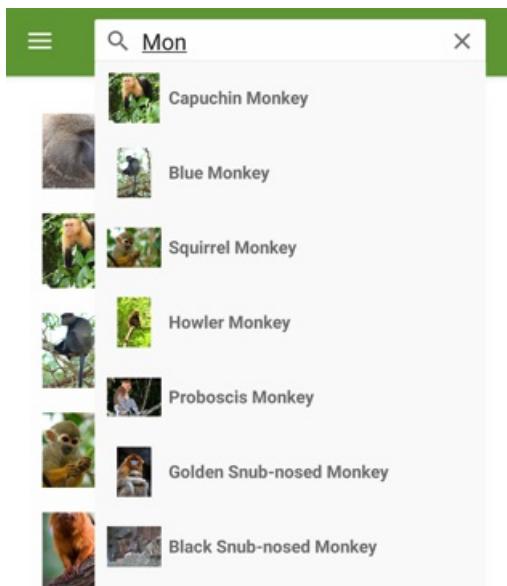
Define search results item appearance

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 ...>
    xmlns: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 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 screenshot shows the result of templating each item in the suggestions area:



For more information about data templates, see [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>
        ...
    </Shell.SearchHandler>
</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 `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>
```

.NET MAUI Shell lifecycle

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

Shell apps respect the .NET Multi-platform App UI (.NET MAUI) 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 app, 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.

Modeless navigation

In a Shell app, 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 modeless navigation, see [Perform modeless navigation](#).

Modal navigation

In a Shell app, 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 [Perform modal navigation](#).

Target multiple platforms from .NET MAUI single project

9/20/2022 • 8 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) single project takes the platform-specific development experiences you typically encounter while developing apps and abstracts them into a single shared project that can target Android, iOS, macOS, and Windows.

.NET MAUI single project provides a simplified and consistent cross-platform development experience, regardless of the platforms being targeted. .NET MAUI single project provides the following features:

- A single shared project that can target Android, iOS, macOS, Tizen, and Windows.
- A simplified debug target selection for running your .NET MAUI apps.
- Shared resource files within the single project.
- A single app manifest that specifies the app title, ID, and version.
- Access to platform-specific APIs and tools when required.
- A single cross-platform app entry point.

.NET MAUI single project is enabled using multi-targeting and the use of SDK-style projects in .NET 6.

Resource files

Resource management for cross-platform app development has traditionally been problematic, because each platform has its own approach to managing resources. For example, each platform has differing image requirements that typically involves creating multiple versions of each image at different resolutions. Therefore, a single image typically has to be duplicated multiple times at different resolutions, with the resulting images having to use different filename and folder conventions on each platform.

.NET MAUI single project enables resource files to be stored in a single location while being consumed on each platform. This includes fonts, images, the app icon, the splash screen, raw assets, and CSS files for styling .NET MAUI apps. Each image resource file is used as a source image, from which images of the required resolutions are generated for each platform at build time.

NOTE

iOS Asset Catalogs are currently unsupported in .NET MAUI single projects.

Resource files should typically be placed in the *Resources* folder of your .NET MAUI app project, or child folders of the *Resources* folder, and must have their build action set correctly. The following table shows the build actions for each resource file type:

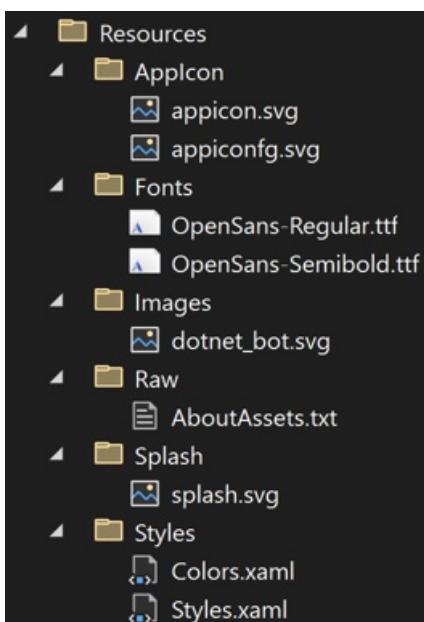
RESOURCE	BUILD ACTION
App icon	MauiIcon
Fonts	MauiFont
Images	MauiImage

RESOURCE	BUILD ACTION
Splash screen	MauiSplashScreen
Raw assets	MauiAsset
CSS files	MauiCss

NOTE

XAML files are also stored in your .NET MAUI app project, and are automatically assigned the **MauiXaml** build action when created by project and item templates. However, only XAML resource dictionaries will typically be placed in the *Resources* folder of the app project.

When a resource file is added to a .NET MAUI app project, a corresponding entry for the resource is created in the project (.csproj) file, with the exception of CSS files. The following screenshot shows a typical *Resources* folder containing child-folders for each resource type:



The build action for a resource file will be correctly set, if the resource has been added to the correct *Resources* child folder.

Child folders of the *Resources* folder can be designated for each resource type by editing the project file for your app:

```

<ItemGroup>
  <!-- Images -->
  <MauiImage Include="Resources\Images\*" />

  <!-- Fonts -->
  <MauiFont Include="Resources\Fonts\*" />

  <!-- Raw assets -->
  <MauiAsset Include="Resources\Raw\*" />
</ItemGroup>
  
```

The wildcard character (`*`) indicates that all the files within the folder will be treated as being of the specified resource type. In addition, it's possible to include all files from child folders:

```
<ItemGroup>
    <!-- Images -->
    <MauiImage Include="Resources\Images\**\*" />
</ItemGroup>
```

In this example, the double wildcard character ('**') specifies that the *Images* folder can contain child folders. Therefore, `<MauiImage Include="Resources\Images***" />` specifies that any files in the *Resources\Images* folder, or any child folders of the *Images* folder, will be used as source images from which images of the required resolution are generated for each platform.

Platform-specific resources will override their shared resource counterparts. For example, if you have an Android-specific image located at *Platforms\Android\Resources\drawable-xhdpi\logo.png*, and you also provide a shared *Resources\Images\logo.svg* image, the Scalable Vector Graphics (SVG) file will be used to generate the required Android images, except for the XHDPI image that already exists as a platform-specific image.

App icons

An app icon can be added to your app project by dragging an image into the *Resources\Icon* folder of the project, where its build action will automatically be set to **MauiIcon**. This creates a corresponding entry in your project file:

```
<MauiIcon Include="Resources\Icon\appicon.svg" />
```

At build time, the app icon can be resized to the correct sizes for the target platform and device. The resized app icons are then added to your app package. App icons are resized to multiple resolutions because they have multiple uses, including being used to represent the app on the device, and in the app store.

For more information, see [Add an app icon to a .NET MAUI app project](#).

Images

An image can be added to your app project by dragging it into the *Resources\Images* folder of the project, where its build action will automatically be set to **MauiImage**. This creates a corresponding entry in your project file:

```
<MauiImage Include="Resources\Images\logo.svg" />
```

At build time, images can be resized to the correct resolutions for the target platform and device. The resulting images are then added to your app package.

For more information, see [Add images to a .NET MAUI app project](#).

Fonts

A true type format (TTF) or open type font (OTF) font can be added to your app project by dragging it into the *Resources\Fonts* folder of your project, where its build action will automatically be set to **MauiFont**. This creates a corresponding entry per font in your project file:

```
<MauiFont Include="Resources\Fonts\OpenSans-Regular.ttf" />
```

At build time, the fonts are copied to your app package.

For more information, see [Fonts](#).

Splash screen

A splash screen can be added to your app project by dragging an image into the *Resources\Splash* folder of the

project, where its build action will automatically be set to **MauiSplashScreen**. This creates a corresponding entry in your project file:

```
<ItemGroup>
  <MauiSplashScreen Include="Resources\Splash\splashscreen.svg" />
</ItemGroup>
```

At build time, the splash screen image is resized to the correct size for the target platform and device. The resized splash screen is then added to your app package.

For more information, see [Add a splash screen to a .NET MAUI app project](#).

Raw assets

A raw asset file, such as HTML, JSON, and video, can be added to your app project by dragging it into the *Resources\Raw* folder of your project, where its build action will automatically be set to **MauiAsset**. This creates a corresponding entry per asset in your project file:

```
<MauiAsset Include="Resources\Raw\index.html" />
```

Raw assets can then be consumed by controls, as required:

```
<WebView Source="index.html" />
```

At build time, raw assets are copied to your app package.

CSS files

.NET MAUI apps can be partially styled with Cascading Style Sheet (CSS) files. CSS files can be added to your app project by dragging them into any folder of your project, and setting their build action to **MauiCss** in the **Properties** window.

CSS files must be loaded by the `StyleSheet` class before being added to a `ResourceDictionary`:

```
<Application ...>
  <Application.Resources>
    <StyleSheet Source="/Resources/styles.css" />
  </Application.Resources>
</Application>
```

For more information, see [Style apps with CSS](#).

App manifest

Each platform uses its own native app manifest file to specify information such as the app title, ID, version, and more..NET MAUI single project enables you to specify this common app data in a single location in the project file (.csproj).

To specify the shared app manifest data for a project, open the shortcut menu for the project in **Solution Explorer**, and then choose **Properties**. The app title, ID, and version can then be specified in **MAUI Shared > General**:

Application Title
The display name of the application.
MauiApp1

Application ID
The identifier of the application in reverse domain name format e.g. com.microsoft.maui.
com.companyname.mauiapp1

Application ID (GUID)
The identifier of the application in GUID format e.g. DA583B0B-9202-42A7-B68C-87225195B0D2.
442A62EB-F23C-4286-AF14-D7551EF08428

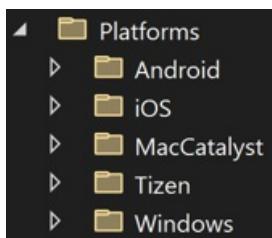
Application Display Version
The display version of the application. This should be at most a three part version number e.g. 1.0.0.
1.0

Application Version
The version of the application. This should be a single digit integer e.g. 1.
1

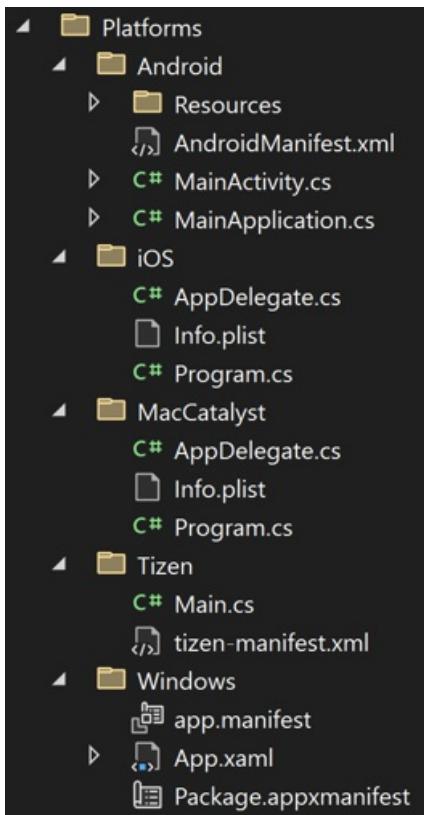
At build time the shared app manifest data is merged with platform-specific data in the native app manifest file, to produce the manifest file for the app package. For more information, see [Project configuration in .NET MAUI - MAUI Shared](#).

Platform-specific code

A .NET MAUI app project contains a *Platforms* folder, with each child folder representing a platform that .NET MAUI can target:



The folders for each platform contain platform-specific resources, and code that starts the app on each platform:



At build time, the build system only includes the code from each folder when building for that specific platform. For example, when you build for Android the files in the *Platforms\Android* folder will be built into the app package, but the files in the other *Platforms* folders won't be. This approach uses multi-targeting to target multiple platforms from a single project. Multi-targeting can be combined with partial classes and partial methods to invoke native platform functionality from cross-platform code. For more information, see [Invoke platform code](#).

In addition to this default multi-targeting approach, .NET MAUI apps can also be multi-targeted based on your own filename and folder criteria. This enables you to structure your .NET MAUI app project so that you don't have to place your platform code into child-folders of the *Platforms* folder. For more information, see [Configure multi-targeting](#).

Multi-targeting can also be combined with conditional compilation so that code is targeted to specific platforms:

```
#if ANDROID
    handler.NativeView.SetBackgroundColor(Colors.Red.ToNative());
#elif IOS
    handler.NativeView.BackgroundColor = Colors.Red.ToNative();
    handler.NativeView.BorderStyle = UIKit.UITextBorderStyle.Line;
#elif WINDOWS
    handler.NativeView.Background = Colors.Red.ToNative();
#endif
```

For more information about conditional compilation, see [Conditional compilation](#).

App entry point

While the *Platforms* folders contain platform-specific code that starts the app on each platform, .NET MAUI apps have a single cross-platform app entry point. Each platform entry point calls a `CreateMauiApp` method on the static `MauiProgram` class in your app project, and returns a `MauiApp`, which is the entry point for your app.

The `MauiProgram` class must at a minimum provide an app to run:

```
namespace MyMauiApp;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>();

        return builder.Build();
    }
}
```

The `App` class derives from the `Application` class:

```
namespace MyMauiApp;

public class App : Application
{
    public App()
    {
        InitializeComponent();

        MainPage = new AppShell();
    }
}
```

In the preceding example, the `MainPage` property is set to the `AppShell` object. `AppShell` is a subclassed `Shell` class that describes the visual hierarchy of the app. For more information, see [Create a .NET MAUI Shell app](#).

Control templates

9/20/2022 • 13 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) 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 app. 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 [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 your app-level resource dictionary, the template can be used anywhere in your app. If you define the template in a page, only that page can use the

control template. For more information about resources, see [Resource dictionaries](#).

The following XAML example shows a `ControlTemplate` for `CardView` objects:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="18"
                        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 [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://schemas.microsoft.com/dotnet/2021/maui"
    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}" />
    </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 screenshot shows the `CardViewControlTemplate` applied to the three `CardView` objects:



John Doe

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla elit dolor, convallis non interdum.



Jane Doe

Phasellus eu convallis mi. In tempus augue eu dignissim fermentum. Morbi ut lacus vitae eros lacinia.

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 [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://schemas.microsoft.com/dotnet/2021/maui"
    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="18"
                        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 screenshot shows the `CardViewControlTemplate` applied to the `CardView` objects:



John Doe

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla elit dolor, convallis non interdum.



Jane Doe

Phasellus eu convallis mi. In tempus augue eu dignissim fermentum. Morbi ut lacus vitae eros lacinia.

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://schemas.microsoft.com/dotnet/2021/maui"
    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"/>
</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 [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://schemas.microsoft.com/dotnet/2021/maui"
    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="18"
                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://schemas.microsoft.com/dotnet/2021/maui"
    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}" />
</StackLayout>
</ContentPage>

```

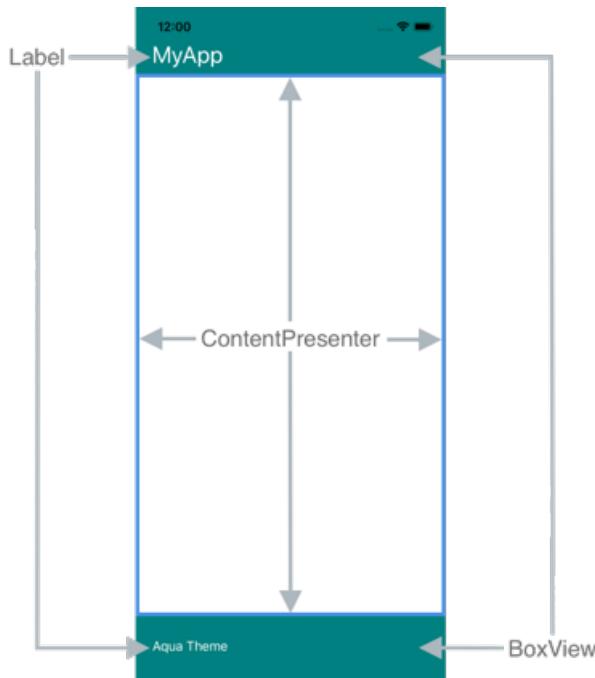
In this example, the visual structure of the `CardViewUI` object is redefined in a `ControlTemplate` that provides a more compact visual structure that's suitable for a condensed list:



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="24"
            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/dotnet/maui/"
            HorizontalOptions="End"
            VerticalOptions="Center" />
    </Grid>
</ControlTemplate>

```

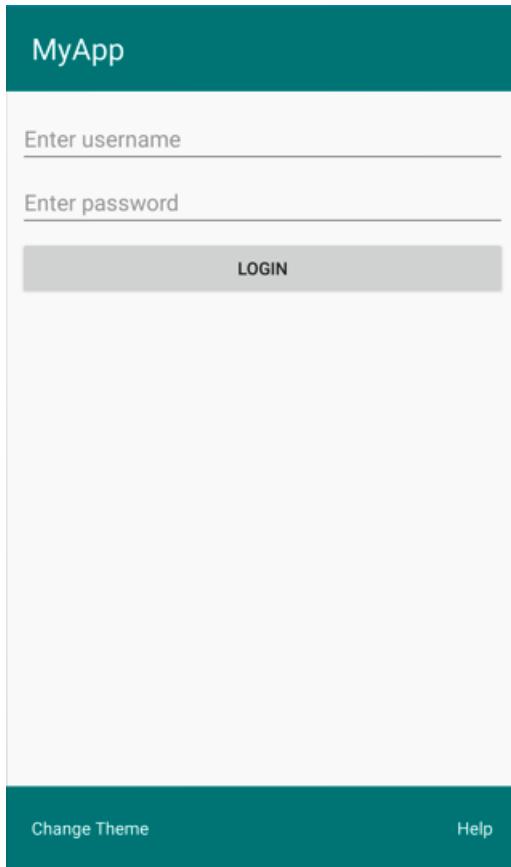
The following example shows `TealTemplate` assigned to the `ControlTemplate` property of a `ContentPage` derived page:

```

<controls:HeaderFooterPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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:

```

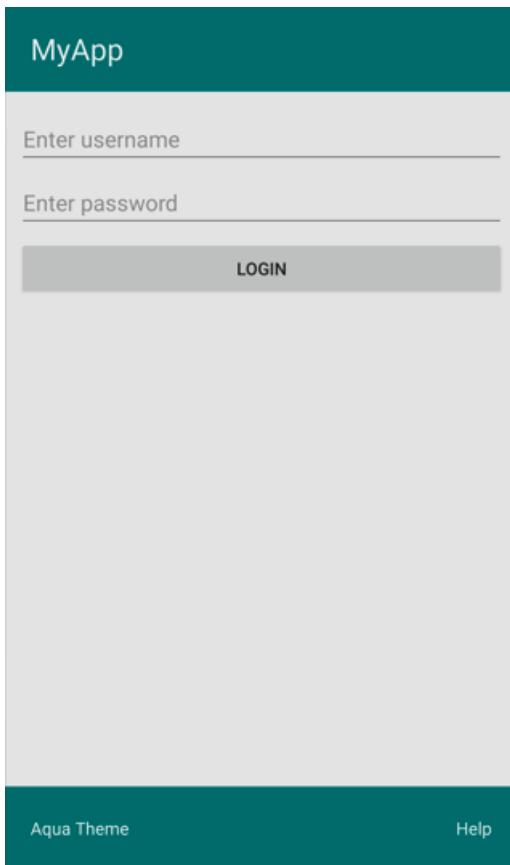
public partial class AccessTemplateElementPage : HeaderFooterPage
{
    Label themeLabel;

    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 screenshot shows 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://schemas.microsoft.com/dotnet/2021/maui"
    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="18"
                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 screenshot shows the page displaying the `People` collection:



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 [Relative bindings](#).

Data templates

9/20/2022 • 7 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) data templates provide the ability to define the presentation of data on supported controls.

Consider a `CollectionView` that displays a collection of `Person` objects. The following 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. A control that displays collections, such as `CollectionView`, can be used to display `Person` objects:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataTemplates"
    x:Class="DataTemplates.WithoutDataTemplatePage">
    <StackLayout>
        <CollectionView>
            <CollectionView.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>
            </CollectionView.ItemsSource>
        </CollectionView>
    </StackLayout>
</ContentPage>
```

In this example, items are added to the `CollectionView` by initializing its `ItemsSource` property from an array of `Person` objects. `CollectionView` calls `ToString` when displaying the objects in the collection. However, because there is no `Person.ToString` override, `ToString` returns the type name of each object:

DataTemplates.Person

DataTemplates.Person

DataTemplates.Person

DataTemplates.Person

DataTemplates.Person

The `Person` object can override the `ToString` method to display meaningful data:

```
public class Person
{
    ...
    public override string ToString ()
    {
        return Name;
    }
}
```

This results in the `CollectionView` displaying the `Person.Name` property value for each object in the collection:

Steve

John

Tom

Lucas

Tariq

Jane

The `Person.ToString` override could return a formatted string consisting of the `Name`, `Age`, and `Location` properties. However, this approach only offers 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.

Create a DataTemplate

A `DataTemplate` is used to specify the appearance of data, and typically uses data binding to display data. A common usage scenario for data templates is when displaying data from a collection of objects in a control such as a `CollectionView` or `CarouselView`. For example, when a `CollectionView` is bound to a collection of `Person` objects, the `CollectionView.ItemTemplate` property can be set to a `DataTemplate` that defines the appearance of each `Person` object in the `CollectionView`. The `DataTemplate` will contain objects that bind to property values of each `Person` object. For more information about data binding, see [Data binding](#).

A `DataTemplate` that's defined inline in a control is known as an *inline template*. Alternatively, data templates can be defined as a control-level, page-level, or app-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 controls on the page.
- A `DataTemplate` defined at the app-level can be applied to valid controls throughout the app.

NOTE

Data templates lower in the view hierarchy take precedence over those defined higher up when they share `x:Key` attributes. For example, an app-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.

A `DataTemplate` can be created inline, with a type, or as a resource, regardless of where it's defined.

Create an inline DataTemplate

An inline data template, which is one that's defined inline in a control, should be used if there's no need to reuse the data template elsewhere. The objects specified in the `DataTemplate` define the appearance of each item of data. A control such as `CollectionView` can then set its `ItemTemplate` property to the inline `DataTemplate`:

```
<CollectionView>
    <CollectionView.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>
    </CollectionView.ItemsSource>
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Grid>
                ...
                <Label Text="{Binding Name}" FontAttributes="Bold" />
                <Label Grid.Column="1" Text="{Binding Age}" />
                <Label Grid.Column="2" Text="{Binding Location}" HorizontalTextAlignment="End" />
            </Grid>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

In a `CollectionView`, the child of an inline `DataTemplate` must derive from `BindableObject`. In this example, a `Grid`, which derives from `Layout` is used. The `Grid` contains three `Label` objects that bind their `Text` properties to properties of each `Person` object in the collection. The following screenshot shows the resulting appearance:

Steve	21	USA
John	37	USA
Tom	42	UK
Lucas	29	Germany
Tariq	39	UK
Jane	30	USA

Create a DataTemplate with a type

A `DataTemplate` can be created with a custom view type. The advantage of this approach is that the appearance defined by the view can be reused by multiple data templates throughout an app. A control such as `CollectionView` can then set its `ItemTemplate` property to the `DataTemplate`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataTemplates"
    x:Class="DataTemplates.WithDataTemplatePageFromType">
    <StackLayout>
        <CollectionView>
            <CollectionView.ItemsSource>
                <x:Array Type="{x:Type local:Person}">
                    <local:Person Name="Steve" Age="21" Location="USA" />
                    ...
                </x:Array>
            </CollectionView.ItemsSource>
            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <local:PersonView />
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>
    </StackLayout>
</ContentPage>

```

In this example, the `CollectionView.ItemTemplate` property is set to a `DataTemplate` that's created from a custom type that defines the view appearance. The custom type must derive from `ContentView`:

```

<ContentView xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="DataTemplates.PersonView">
    <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>
</ContentView>

```

In this example, layout within the `ContentView` is managed by a `Grid`. The `Grid` contains three `Label` objects that bind their `Text` properties to properties of each `Person` object in the collection.

For more information about creating custom views, see [ContentView](#).

Create a DataTemplate as a resource

Data templates can be created as reusable objects in a `ResourceDictionary`. This is achieved by giving each `DataTemplate` a unique `x:Key` value, which provides it with a descriptive key in the `ResourceDictionary`. A control such as `CollectionView` can then set its `ItemTemplate` property to the `DataTemplate`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:DataTemplates"
    x:Class="DataTemplates.WithDataTemplateResource">
    <ContentPage.Resources>
        <ResourceDictionary>
            <DataTemplate x:Key="personTemplate">
                <Grid>
                    ...
                </Grid>
            </DataTemplate>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <CollectionView ItemTemplate="{StaticResource personTemplate}">
            <CollectionView.ItemsSource>
                <x:Array Type="{x:Type local:Person}">
                    <local:Person Name="Steve" Age="21" Location="USA" />
                    ...
                </x:Array>
            </CollectionView.ItemsSource>
        </CollectionView>
    </StackLayout>
</ContentPage>

```

In this example, the `DataTemplate` is assigned to the `CollectionView.ItemTemplate` property by using the `StaticResource` markup extension. While the `DataTemplate` is defined in the page's `ResourceDictionary`, it could also be defined at the control-level or app-level.

Create 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 data templates to be applied to the same type of object, to choose their appearance at runtime. A data template selector enables scenarios such as a `CollectionView` or `CarouselView` binding to a collection of objects where the appearance of each object can be chosen at runtime by the data template selector returning a specific `DataTemplate`.

A data template selector is implemented by creating a class that inherits from `DataTemplateSelector`. The `OnSelectTemplate` method should then be overridden to return a specific `DataTemplate`:

```

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;
    }
}

```

In this example, the `OnSelectTemplate` method returns a specific data template based on the value of the `DateOfBirth` property. The returned data template is defined by the `ValidTemplate` or `InvalidTemplate` property, which are set when consuming the data template selector.

Limitations

`DataTemplateSelector` objects 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 control virtualization.

Consume a DataTemplateSelector

A data template selector can be consumed by creating it as a resource and assigning its instance to .NET MAUI control properties of type `DataTemplate`, such as `CollectionView.ItemTemplate`.

The following example shows declaring `PersonDataTemplateSelector` as a page-level resource:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:local="clr-namespace:Selector"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Selector.MainPage">
<ContentPage.Resources>
    <DataTemplate x:Key="validPersonTemplate">
        <Grid>
            ...
        </Grid>
    </DataTemplate>
    <DataTemplate x:Key="invalidPersonTemplate">
        <Grid>
            ...
        </Grid>
    </DataTemplate>
    <local:PersonDataTemplateSelector x:Key="personDataTemplateSelector"
        ValidTemplate="{StaticResource validPersonTemplate}"
        InvalidTemplate="{StaticResource invalidPersonTemplate}" />
</ContentPage.Resources>
...
</ContentPage>
```

In this example, the page-level `ResourceDictionary` defines two `DataTemplate` objects and a `PersonDataTemplateSelector` object. The `PersonDataTemplateSelector` object sets its `ValidTemplate` and `InvalidTemplate` properties to the `DataTemplate` objects using the `StaticResource` markup extension. While the resources are defined in the page's `ResourceDictionary`, they could also be defined at the control-level or app-level.

The `PersonDataTemplateSelector` object can be consumed by assigning it to the `CollectionView.ItemTemplate` property:

```
<CollectionView x:Name="collectionView"
    ItemTemplate="{StaticResource personDataTemplateSelector}" />
```

At runtime, the `CollectionView` 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 returned `DataTemplate` is then applied to that object.

The following screenshot shows the result of the `CollectionView` applying the `PersonDataTemplateSelector` to each object in the underlying collection:

Kath	11/20/1985	France
Steve	1/15/1975	USA
Lucas	2/5/1988	Germany
John	2/20/1976	USA
Tariq	1/10/1987	UK
Jane	8/30/1982	USA
Tom	3/10/1977	UK

In this example, 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.

Triggers

9/20/2022 • 11 minutes to read • [Edit Online](#)

{ } [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) triggers allow you to express actions declaratively in XAML that change the appearance of controls based on events or data 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's `Triggers` collection, or add it to a page-level or app-level resource dictionary to be applied to multiple controls.

Property triggers

A `Trigger` represents a trigger that applies property values, or performs actions, when the specified property meets a specified condition.

The following 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 Setter elements are allowed -->
        </Trigger>
    </Entry.Triggers>
</Entry>
```

The trigger's declaration specifies the following:

- `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 that are applied when the trigger condition is met.

In addition, optional `EnterActions` and `ExitActions` collections can be specified. For more information, see [EnterActions and ExitActions](#).

Apply a trigger using a style

Triggers can also be added to a `Style` declaration on a control, in a page, or an application `ResourceDictionary`. The following example declares an *implicit* style for all `Entry` controls on the page:

```

<ContentPage.Resources>
    <Style TargetType="Entry">
        <Style.Triggers>
            <Trigger TargetType="Entry"
                Property="IsFocused"
                Value="True">
                <Setter Property="BackgroundColor"
                    Value="Yellow" />
                <!-- Multiple Setter elements are allowed -->
            </Trigger>
        </Style.Triggers>
    </Style>
</ContentPage.Resources>

```

Data triggers

A `DataTrigger` represents a trigger that applies property values, or performs actions, when the bound data meets a specified condition. The `Binding` markup extension is used to monitor for the specified condition.

The following example shows a `DataTrigger` that disables a `Button` when the `Entry` is empty:

```

<Entry x:Name="entry"
       Text=""
       Placeholder="Enter text" />
<Button Text="Save">
    <Button.Triggers>
        <DataTrigger TargetType="Button"
                     Binding="{Binding Source={x:Reference entry},
                                     Path=Text.Length}"
                     Value="0">
            <Setter Property="IsEnabled"
                   Value="False" />
        <!-- Multiple Setter elements are allowed -->
    </DataTrigger>
    </Button.Triggers>
</Button>

```

In this example, when the length of the `Entry` is zero, the trigger is activated.

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, optional `EnterActions` and `ExitActions` collections can be specified. For more information, see [EnterActions and ExitActions](#).

Event triggers

An `EventTrigger` represents a trigger that applies a set of actions in response to an event. Unlike `Trigger`, `EventTrigger` has no concept of termination of state, so the actions will not be undone once the condition that raised the event is no longer true.

An `EventTrigger` only requires an `Event` property to be set:

```
<EventTrigger Event="TextChanged">
    <local:NumericValidationTriggerAction />
</EventTrigger>
```

In this example, there are no `Setter` elements. Instead, there's a `NumericalValidationTriggerAction` object.

NOTE

Event triggers don't support `EnterActions` and `ExitActions`.

A trigger action implementation must:

- 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 classes 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 method is called whenever the trigger event occurs.
- Optionally expose properties that can be set in XAML when the trigger is declared.

The following example shows the `NumericValidationTriggerAction` class:

```
public class NumericValidationTriggerAction : TriggerAction<Entry>
{
    protected override void Invoke(Entry entry)
    {
        double result;
        bool isValid = Double.TryParse(entry.Text, out result);
        entry.TextColor = isValid ? Colors.Black : Colors.Red;
    }
}
```

WARNING

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.

Multi-triggers

A `MultiTrigger` represents a trigger that applies property values, or performs actions, when a set of conditions are satisfied. All the conditions must be true before the `setter` objects are applied.

The following example shows a `MultiTrigger` that binds to two `Entry` objects:

```

<Entry x:Name="email"
       Text="" />
<Entry x:Name="phone"
       Text="" />
<Button Text="Save">
    <Button.Triggers>
        <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>
    </Button.Triggers>
</Button>

```

In addition, the `MultiTrigger.Conditions` collection can also contain `PropertyCondition` objects:

```

<PropertyCondition Property="Text"
                    Value="OK" />

```

EnterActions and ExitActions

An alternative approach to implementing changes when a trigger occurs is by specifying `EnterActions` and `ExitActions` collections, and creating `TriggerAction<T>` implementations.

The `EnterActions` collection, of type `IList<TriggerAction>`, defines a collection that will be invoked when the trigger condition is met. The `ExitActions` collection, of type `IList<TriggerAction>`, defines a collection 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.

The following example shows a property trigger that specifies an `EnterAction` and an `ExitAction`:

```

<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>
        </Trigger>
    </Entry.Triggers>
</Entry>

```

A trigger action implementation must:

- 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 classes 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 method is called whenever the trigger event occurs.
- Optionally expose properties that can be set in XAML when the trigger is declared.

The following example shows the `FadeTriggerAction` class:

```
public class FadeTriggerAction : TriggerAction<VisualElement>
{
    public int StartsFrom { get; set; }

    protected override void Invoke(VisualElement sender)
    {
        sender.Animate("FadeTriggerAction", new Animation((d) =>
        {
            var val = StartsFrom == 1 ? d : 1 - d;
            sender.BackgroundColor = Color.FromRgb(1, val, 1);
        }),
        length: 1000, // milliseconds
        easing: Easing.Linear);
    }
}
```

NOTE

You can provide `EnterActions` and `ExitActions` as well as `Setter` objects in a trigger, but be aware that the `Setter` objects are called immediately (they do not wait for the `EnterAction` or `ExitAction` to complete).

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, .NET MAUI 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 [Visual states](#).

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 `VisualState` change is triggered, and the background color of the `Grid` becomes white.

In addition, every time a `VisualState` change occurs, the `IsActiveChanged` event for the `VisualState` is raised. Each `VisualState` 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 raised, the handler outputs whether the `VisualState` 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 `VisualStyle` change is triggered, and the `StackLayout` orientation changes to horizontal.

The `MinWindowHeight` and `MinWindowWidth` 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 `VisualStyle` 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 `VisualStyle` 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 `VisualStyle` 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 `VisualStyle` change is triggered, and the background color of the `Grid` becomes black.

Device state trigger

The `DeviceStateTrigger` triggers a `VisualStyle` 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 `VisualStyle` 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>
            </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, and to pale blue on 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.

.NET MAUI windows

9/20/2022 • 3 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Window` class provides the ability to create, configure, show, and manage multiple windows.

`Window` defines the following properties:

- `Title`, of type `string`, represents the title of the window.
- `Page`, of type `Page`, indicates the page being displayed by the window. This property is the content property of the `Window` class, and therefore does not need to be explicitly set.
- `FlowDirection`, of type `FlowDirection`, defines the direction in which the UI element of the window are laid out.
- `Overlays`, of type `IReadOnlyCollection<IWindowOverlay>`, represents the collection of window overlays.

These properties, with the exception of the `Overlays` property, are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `Window` class defines the following modal navigation events:

- `ModalPopped`, with `ModalPoppedEventArgs`, which is raised when a view has been popped modally.
- `ModalPopping`, with `ModalPoppingEventArgs`, which is raised when a view is modally popped.
- `ModalPushed`, with `ModalPushedEventArgs`, which is raised after a view has been pushed modally.
- `ModalPushing`, with `ModalPushingEventArgs`, which is raised when a view is modally pushed.
- `PopCanceled`, which is raised when a modal pop is cancelled.

The `Window` class also defines the following lifecycle events:

- `Created`, which is raised when the Window is created.
- `Resumed`, which is raised when the Window is resumed from a sleeping state.
- `Activated`, which is raised when the Window is activated.
- `Deactivated`, which is raised when the Window is deactivated.
- `Stopped`, which is raised when the Window is stopped.
- `Destroying`, which is raised when the Window is destroyed.
- `Backgrounding`, with `BackgroundingEventArgs`, which is raised when the Window is entering a background state.
- `DisplayDensityChanged`, with `DisplayDensityChangedEventArgs`, which is raised when the effective dots per inch (DPI) for the Window has changed.

For more information about the lifecycle events, and their associated overrides, see [App lifecycle](#).

Create a Window

By default, .NET MAUI creates a `Window` object when you set the `MainPage` property to a `Page` object in your `App` class. However, you can also override the `CreateWindow` method in your `App` class to create a `Window` object:

```

namespace MyMauiApp
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();

            MainPage = new MainPage();
        }

        protected override Window CreateWindow(IActivationState activationState)
        {
            Window window = base.CreateWindow(activationState);

            // Manipulate Window object

            return window;
        }
    }
}

```

While the `Window` class has a default constructor and a constructor that accepts a `Page` argument, which represents the root page of the app, you can also call the base `CreateWindow` method to return the .NET MAUI created `Window` object.

In addition, you can also create your own `Window`-derived object:

```

namespace MyMauiApp
{
    public class MyWindow : Window
    {
        public MyWindow() : base()
        {

        }

        public MyWindow(Page page) : base(page)
        {

        }

        // Override Window methods
    }
}

```

The `Window`-derived class can then be consumed by creating a `MyWindow` object in the `CreateWindow` override in your `App` class.

Regardless of how your `Window` object is created, it will be the parent of the root page in your app.

Multi-window support

Multiple windows can be simultaneously opened on Android, iOS on iPad (iPadOS), Mac Catalyst, and Windows. This can be achieved by creating a `Window` object and opening it using the `OpenWindow` method on the `Application` object:

```

Window secondWindow = new Window(new MyPage());
Application.Current.OpenWindow(secondWindow);

```

The `Application.Current.Windows` collection, of type `IReadOnlyList<Window>` maintains references to all `Window`

objects that are registered with the `Application` object.

Windows can be closed with the `Application.Current.CloseWindow` method:

```
// Close a specific window  
Application.Current.CloseWindow(secondWindow);  
  
// Close the active window  
Application.Current.CloseWindow(GetParentWindow());
```

IMPORTANT

Multi-window support works on Android and Windows without additional configuration. However, additional configuration is required on iPadOS and Mac Catalyst.

iPadOS and macOS configuration

To use multi-window support on iPadOS and Mac Catalyst, add a class named `SceneDelegate` to the **Platforms > iOS and Platforms > MacCatalyst** folders:

```
using Foundation;  
using Microsoft.Maui;  
using UIKit;  
  
namespace MyMauiApp;  
  
[Register("SceneDelegate")]  
public class SceneDelegate : MauiUISceneDelegate  
{  
}
```

Then, in the XML editor, open the **Platforms > iOS > Info.plist** file and the **Platforms > MacCatalyst > Info.plist** file and add the following XML to the end of each file:

```
<key>UIApplicationSceneManifest</key>  
<dict>  
    <key>UIApplicationSupportsMultipleScenes</key>  
    <true/>  
    <key>UISceneConfigurations</key>  
    <dict>  
        <key>UIWindowSceneSessionRoleApplication</key>  
        <array>  
            <dict>  
                <key>UISceneConfigurationName</key>  
                <string>__MAUI_DEFAULT_SCENE_CONFIGURATION__</string>  
                <key>UISceneDelegateClassName</key>  
                <string>SceneDelegate</string>  
            </dict>  
        </array>  
    </dict>  
</dict>
```

IMPORTANT

Multi-window support doesn't work on iOS for iPhone.

Basic animation

9/20/2022 • 8 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) animation classes target different properties of visual elements, with a typical basic animation progressively changing a property from one value to another over a period of time.

Basic animations can be created with extension methods provided by the `ViewExtensions` class, in the `Microsoft.Maui.Controls` namespace, which operate on `VisualElement` objects:

- `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 also 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.

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, when animation operations are combined with the `await` operator it 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](#).

Single animations

Each extension method in the `ViewExtensions` class implements a single animation operation that progressively changes a property from one value to another value over a period of time.

Rotation

Rotation is performed with the `RotateTo` method, which progressively changes the `Rotation` property of an element:

```
await image.RotateTo(360, 2000);
image.Rotation = 0;
```

In this example, an `Image` instance is rotated up to 360 degrees over 2 seconds (2000 milliseconds). The `RotateTo` method obtains the current `Rotation` property value of the element 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.

NOTE

In addition to the `RotateTo` method, there are also `RotateXTo` and `RotateYTo` methods that animate the `RotationX` and `RotationY` properties, respectively.

Relative rotation

Relative rotation is performed with the `RelRotateTo` method, which progressively changes the `Rotation` property of an element:

```
await image.RelRotateTo(360, 2000);
```

In this example, an `Image` instance is rotated 360 degrees from its starting position over 2 seconds (2000 milliseconds). The `RelRotateTo` method obtains the current `Rotation` property value of the element 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.

Scaling

Scaling is performed with the `ScaleTo` method, which progressively changes the `Scale` property of an element:

```
await image.ScaleTo(2, 2000);
```

In this example, an `Image` instance is scaled up to twice its size over 2 seconds (2000 milliseconds). The `ScaleTo` method obtains the current `Scale` property value of the element for the start of the animation, and then scales from that value to its first argument. This has the effect of expanding the size of the image to twice its size.

NOTE

In addition to the `ScaleTo` method, there are also `ScaleXTo` and `ScaleYTo` methods that animate the `ScaleX` and `ScaleY` properties, respectively.

Relative scaling

Relative scaling is performed with the `RelScaleTo` method, which progressively changes the `Scale` property of an element:

```
await image.RelScaleTo(2, 2000);
```

In this example, an `Image` instance is scaled up to twice its size over 2 seconds (2000 milliseconds). The `RelScaleTo` method obtains the current `Scale` property value of the element for the start of the animation, and then scales from that value to the value plus its first argument. 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 of a visual element 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.AbsoluteLayout.Width, layout.AbsoluteLayout.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.

Translation

Translation is performed with the `TranslateTo` method, which progressively changes the `TranslationX` and `TranslationY` properties of an element:

```
await image.TranslateTo(-100, -100, 1000);
```

In this example, the `Image` instance is translated horizontally and vertically over 1 second (1000 milliseconds). The `TranslateTo` method simultaneously translates the image 100 device-independent units to the left, and 100 device-independent units 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.

IMPORTANT

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

Fading is performed with the `FadeTo` method, which progressively changes the `Opacity` property of an element:

```
image.Opacity = 0;
await image.FadeTo(1, 4000);
```

In this example, the `Image` instance fades in over 4 seconds (4000 milliseconds). The `FadeTo` method obtains the current `Opacity` property value of the element for the start of the animation, and then fades in from that value to its first argument.

Compound animations

A compound animation is a sequential combination of animations, and can be created with the `await` operator:

```
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` instance 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 combining awaited and non-awaited animations:

```
image.RotateTo(360, 4000);
await image.ScaleTo(2, 2000);
await image.ScaleTo(1, 2000);
```

In this example, the `Image` instance 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 delays the second `ScaleTo` method until the first `ScaleTo` method has completed. At this point the `RotateTo` animation is half 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.

Run multiple animations concurrently

The `Task.WhenAny` and `Task.WhenAll` methods can be used to run multiple asynchronous methods concurrently, and therefore can 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 contains two tasks. The first task rotates an `Image` instance 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 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.

Canceling animations

An app can cancel one or more animations with a call to the `CancelAnimations` extension method:

```
image.CancelAnimations();
```

In this example, all animations that are running on the `Image` instance are immediately canceled.

Easing functions

9/20/2022 • 3 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) includes an `Easing` class that enables you to specify a transfer function that controls how animations speed up or slow down as they're running.

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](#).

Consume an easing function

The animation extension methods in the `ViewExtensions` class allow an easing function to be specified as the final method argument:

```
await image.TranslateTo(0, 200, 2000, Easing.BounceIn);
await image.ScaleTo(2, 2000, Easing.CubicIn);
await image.RotateTo(360, 2000, Easing.SinInOut);
await image.ScaleTo(1, 2000, Easing.CubicOut);
await image.TranslateTo(0, -200, 2000, Easing.BounceOut);
```

By specifying an easing function for an animation, the animation velocity becomes non-linear and produces the effect provided by the easing function. Omitting an easing function when creating an animation causes the animation to use the default `Linear` easing function, which produces a linear velocity.

For more information about using the animation extension methods in the `ViewExtensions` class, see [Basic animation](#). Easing functions can also be consumed by the `Animation` class. For more information, see [Custom animation](#).

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 a value between 0 and 1.

Custom easing method

A custom easing function can be defined as a method that takes a `double` argument, and returns a `double` result:

```
double CustomEase (double t)
{
    return t == 0 || t == 1 ? t : (int)(5 * t) / 5.0;
}

await image.TranslateTo(0, 200, 2000, (Easing)CustomEase);
```

In this example, 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>`:

```
Func<double, double> CustomEaseFunc = t => 9 * t * t * t - 13.5 * t * t + 5.5 * t;
await image.TranslateTo(0, 200, 2000, CustomEaseFunc);
```

In this example, 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:

```
await image.TranslateTo(0, 200, 2000, new Easing (t => 1 - Math.Cos (10 * Math.PI * t) * Math.Exp (-5 * t)));
```

In this example, 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 position.

Custom animation

9/20/2022 • 9 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Animation` class is the building block of all .NET MAUI animations, with the extension methods in the `ViewExtensions` class creating one or more `Animation` objects.

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.

NOTE

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);
```

In this example, an animation of the `Scale` property of an `Image` instance is defined from a value of 1 to a value of 2. The animated value 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:

```
animation.Commit(this, "SimpleAnimation", 16, 2000, Easing.Linear, (v, c) => image.Scale = 1, () => true);
```

NOTE

The `Commit` method doesn't 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.

In the above example, the overall effect is to create an animation that increases the `Scale` property of an `Image` instance 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 are `Animation` objects to which other `Animation` objects are added as children. 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:

```
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 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 of this example 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. Both scaling animations also use easing functions. The `SpringIn` easing function causes the `Image` instance 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 app can cancel a custom animation with a call to the `AbortAnimation` extension method:

```
this.AbortAnimation ("SimpleAnimation");
```

Because animations are uniquely identified by a combination of the animation owner, and the animation name, the owner and name specified when running the animation must be specified to cancel it. Therefore, this 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 = Colors.Black);
```

The resulting animation provides the appearance of advancing the page background through the colors of the

rainbow.

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. This is because different controls have different properties of type `Color`. While the `VisualElement` class defines a `BackgroundColor` property, this isn't always the desired `Color` property to animate.

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.Red + t * (toColor.Red - fromColor.Red),
                           fromColor.Green + t * (toColor.Green - fromColor.Green),
                           fromColor.Blue + t * (toColor.Blue - fromColor.Blue),
                           fromColor.Alpha + t * (toColor.Alpha - fromColor.Alpha));
        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, in this example 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 `Red`, `Green`, `Blue`, and `Alpha` values of the two supplied `Color` arguments. The `color` value is then passed to the callback method to be applied to a property. This approach allows the `ColorTo` method to animate any specified `Color` property:

```
await Task.WhenAll(
    label.ColorTo(Colors.Red, Colors.Blue, c => label.TextColor = c, 5000),
    label.ColorTo(Colors.Blue, Colors.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(Colors.Blue, Colors.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`.

Brushes

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

A .NET Multi-platform App UI (.NET MAUI) brush enables you to paint an area, such as the background of a control, using different approaches.

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 .NET MAUI:

- `SolidColorBrush`, which paints an area with a solid color. For more information, see [Solid color brushes](#).
- `LinearGradientBrush`, which paints an area with a linear gradient. For more information, see [Linear gradient brushes](#).
- `RadialGradientBrush`, which paints an area with a radial gradient. For more information, see [Radial gradient brushes](#).

Instances of these brush types can be assigned to the `Stroke` and `Fill` properties of a `Shape`, the `Stroke` property of a `Border`, the `Brush` property of a `Shadow`, 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.

Solid color brushes

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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

.NET MAUI 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

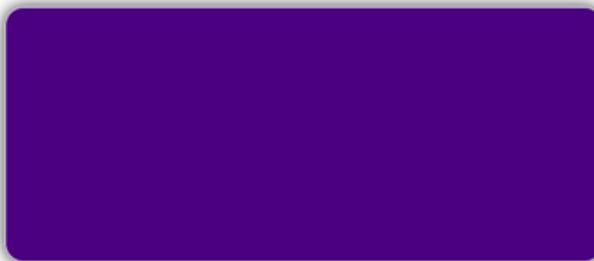
The `Brush` class defines a set of commonly used `SolidColorBrush` objects. The following example uses one of these predefined `SolidColorBrush` objects:

```
<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 = Colors.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](#).

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	#FFFFEBBD	DarkKhaki	#FFBDB076B	Green	#FF008000	Lime	#FF00FF00	Orchid	#FFDA70D6	Show	#FFFFFAFA
Aqua	#FF00FFFF	DarkMagenta	#FFB800BB	GreenYellow	#FFADFF2F	LimeGreen	#FF32CD32	PaleGoldenrod	#FFEE8AA	SpringGreen	#FF00FF7F
Aquamarine	#FF7FFF7F	DarkOliveGreen	#FF556B2F	Honeydew	#FFDFFF00	Linen	#FFFA0E6	PaleGreen	#FFFE4E1	SteelBlue	#FF4682B4
Azure	#FF00FFFF	DarkOrange	#FFF8FC00	HotPink	#FFFF69B4	Magenta	#FFFF00FF	PaleTurquoise	#FFAFEEEE	Tan	#FFD2B48C
Beige	#FFFF5555	DarkOrchid	#FF9932CC	IndianRed	#FCDCSC	Maroon	#FFB000FF	PaleVioletRed	#FDB7093	Teal	#FF008080
Bisque	#FFFFE4C4	DarkRed	#FFB80000	Indigo	#FF4B0082	MediumAquamarine	#FF66CDAA	PapayaWhip	#FFFEEFD5	Thistle	#FFDB8FD8
Black	#FF000000	DarkSalmon	#FFE9967A	Ivory	#FFFFF0F0	MediumBlue	#FF0000CD	PeachPuff	#FFFFDA89	Tomato	#FFFF6347
BlanchedAlmond	#FFFBEEBC	DarkSeaGreen	#FF88BCBF	Khaki	#FFFFF0F0	MediumOrchid	#FFBA55D3	Peru	#FFC0853F	Transparent	#00FFFFFF
Blue	#FF0000FF	DarkSlateBlue	#F48308B	Lavender	#FFEE66FA	MediumPurple	#FF9370DB	Pink	#FFFFCCB	Turquoise	#FF40E0D0
BlueViolet	#FFBA28E2	DarkSlateGray	#FF214F4F	LavenderBlush	#FFFFFOF5	MediumSeaGreen	#FF3C8371	Plum	#FFDDA0D0	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	#FF6F9EA0	DeepPink	#FFF1493	LightBlue	#FFADD8E6	MediumTurquoise	#FF4B01CC	Red	#FFFF0000	WhiteSmoke	#FF55F5F5
Chartreuse	#FF7FFF00	DeepSkyBlue	#FF00BFFF	LightCoral	#FFFB08080	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	#FFFF99D0	FloralWhite	#FFFFFA00	LightGreen	#FF90EE90	Moccasin	#FFFE4E85	SandyBrown	#FFFA4A60		
Crimson	#FFDC143C	ForestGreen	#FF28B222	LightPink	#FFFFB6C1	NavajoWhite	#FFFDDEAD	SeaGreen	#FF2E8B57		
Cyan	#FF00FFFF	Fuchsia	#FFF00FFF	LightSalmon	#FFFA0A7A	Navy	#FF000080	Seashell	#FFFFF5EE		
DarkBlue	#FF00008B	Gainsboro	#FDCCDCDC	LightSeaGreen	#FFB20BAA	OldLace	#FF0000B8	Sienna	#FFA0522D		
DarkCyan	#FF00888B	GhostWhite	#FFF8BF8F	LightSkyBlue	#FB7CECFA	Olive	#FFB08000	Silver	#FFCC00C0		
DarkGoldenrod	#FFB8860B	Gold	#FFFFD700	LightSlateGray	#FF778899	OliveDrab	#FFB8E223	SkyBlue	#FF87CEEB		
DarkGray	#FFA9A9A9	Goldenrod	#FFDAE520	LightSteelBlue	#FFB0C4DE	Orange	#FFFA500	SlateBlue	#FFB8A5CD		

Gradients

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 .NET MAUI provides the following gradient brushes:

- `LinearGradientBrush`, which paints an area with a linear gradient. For more information, see [Linear gradient brushes](#).
- `RadialGradientBrush`, which paints an area with a radial gradient. For more information, see [Radial gradient brushes](#).

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.
- `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.

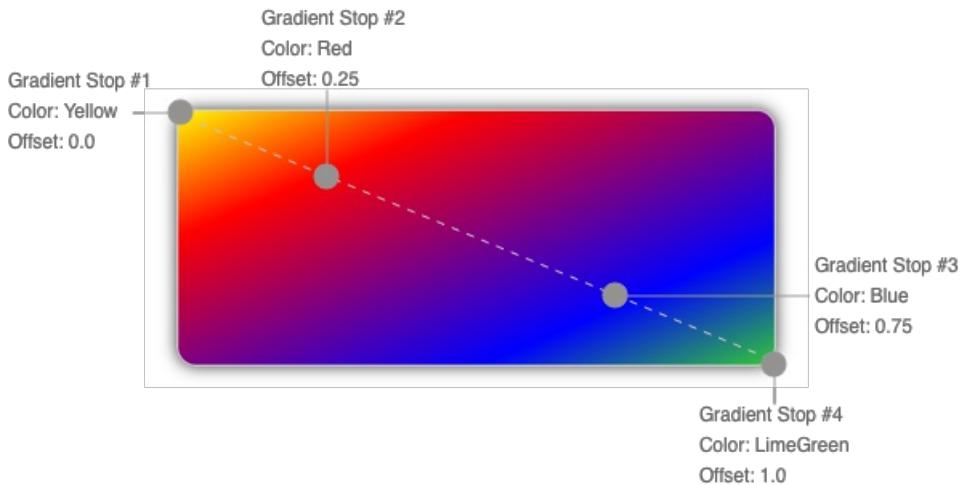
IMPORTANT

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.

Linear gradient brushes

9/20/2022 • 3 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [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.

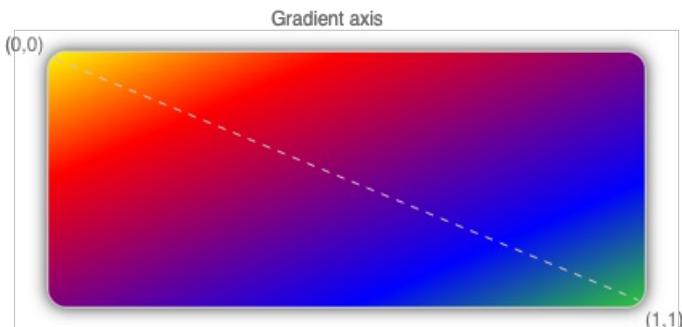
NOTE

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:



Radial gradient brushes

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [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.

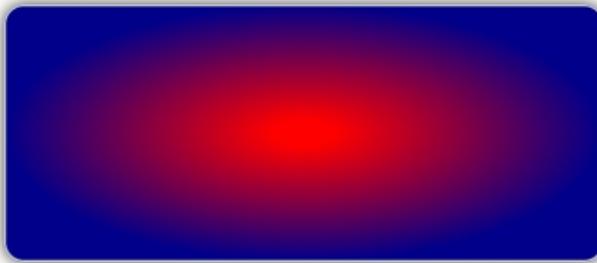
The following XAML example shows a `RadialGradientBrush` that's set as the `Background` of a `Frame`:

```

<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`:



Controls

9/20/2022 • 6 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The user interface of a .NET Multi-platform App UI (.NET MAUI) app is constructed of objects that map to the native controls of each target platform.

The main control groups used to create the user interface of a .NET MAUI app are pages, layouts, and views. A .NET MAUI page generally occupies the full screen or window. The page usually contains a layout, which contains views and possibly other layouts. Pages, layouts, and views derive from the `VisualElement` class. This class provides a variety of properties, methods, and events that are useful in derived classes.

NOTE

`ListView` and `TableView` also support the use of cells. Cells are specialized elements used for items in a table, that describe how each item should be rendered.

Pages

.NET MAUI apps consist of one or more pages. A page usually occupies all of the screen, or window, and each page typically contains at least one layout.

.NET MAUI contains the following pages:

PAGE	DESCRIPTION
<code>ContentPage</code>	<code>ContentPage</code> displays a single view, and is the most common page type. For more information, see ContentPage .
<code>FlyoutPage</code>	<code>FlyoutPage</code> is a page that manages two related pages of information – a flyout page that presents items, and a detail page that presents details about items on the flyout page. For more information, see FlyoutPage .
<code>NavigationPage</code>	<code>NavigationPage</code> provides a hierarchical navigation experience where you're able to navigate through pages, forwards and backwards, as desired. For more information, see NavigationPage .
<code>TabbedPage</code>	<code>TabbedPage</code> consists of a series of pages that are navigable by tabs across the top or bottom of the page, with each tab loading the page content. For more information, see TabbedPage .

Layouts

.NET MAUI layouts are used to compose user-interface controls into visual structures, and each layout typically contains multiple views. Layout classes typically contain logic to set the position and size of child elements.

.NET MAUI contains the following layouts:

LAYOUT	DESCRIPTION
AbsoluteLayout	<code>AbsoluteLayout</code> positions child elements at specific locations relative to its parent. For more information, see AbsoluteLayout .
BindableLayout	<code>BindableLayout</code> enables any layout class to generate its content by binding to a collection of items, with the option to set the appearance of each item. For more information, see BindableLayout .
FlexLayout	<code>FlexLayout</code> enables its children to be stacked or wrapped with different alignment and orientation options. <code>FlexLayout</code> is based on the CSS Flexible Box Layout Module, known as <i>flex layout</i> or <i>flex-box</i> . For more information, see FlexLayout .
Grid	<code>Grid</code> positions its child elements in a grid of rows and columns. For more information, see Grid .
HorizontalStackLayout	<code>HorizontalStackLayout</code> positions child elements in a horizontal stack. For more information, see HorizontalStackLayout .
StackLayout	<code>StackLayout</code> positions child elements in either a vertical or horizontal stack. For more information, see StackLayout .
VerticalStackLayout	<code>VerticalStackLayout</code> positions child elements in a vertical stack. For more information, see VerticalStackLayout .

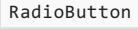
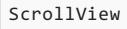
Views

.NET MAUI views are the UI objects such as labels, buttons, and sliders that are commonly known as *controls* or *widgets* in other environments.

.NET MAUI contains the following views:

VIEW	DESCRIPTION
ActivityIndicator	<code>ActivityIndicator</code> uses an animation to show that the app is engaged in a lengthy activity, without giving any indication of progress. For more information, see ActivityIndicator .
BlazorWebView	<code>BlazorWebView</code> enables you to host a Blazor web app in your .NET MAUI app. For more information, see BlazorWebView .
Border	<code>Border</code> is a container control that draws a border, background, or both, around another control. For more information, see Border .
BoxView	<code>BoxView</code> draws a rectangle or square, of a specified width, height, and color. For more information, see BoxView .

VIEW	DESCRIPTION
<code>Button</code>	<code>Button</code> displays text and responds to a tap or click that directs an app to carry out a task. For more information, see Button .
<code>CarouselView</code>	<code>CarouselView</code> displays a scrollable list of data items, where users swipe to move through the collection. For more information, see CarouselView .
<code>CheckBox</code>	<code>CheckBox</code> enables you to select a boolean value using a type of button that can either be checked or empty. For more information, see CheckBox .
<code>CollectionView</code>	<code>CollectionView</code> displays a scrollable list of selectable data items, using different layout specifications. For more information, see CollectionView .
<code>ContentView</code>	<code>ContentView</code> is a control that enables the creation of custom, reusable controls. For more information, see ContentView .
<code>DatePicker</code>	<code>DatePicker</code> enables you to select a date with the platform date picker. For more information, see DatePicker .
<code>Editor</code>	<code>Editor</code> enables you to enter and edit multiple lines of text. For more information, see Editor .
<code>Ellipse</code>	<code>Ellipse</code> displays an ellipse or circle. For more information, see Ellipse .
<code>Entry</code>	<code>Entry</code> enables you to enter and edit a single line of text. For more information, see Entry .
<code>Frame</code>	<code>Frame</code> is used to wrap a view or layout with a border that can be configured with color, shadow, and other options. For more information, see Frame .
<code>GraphicsView</code>	<code>GraphicsView</code> is a graphics canvas on which 2D graphics can be drawn using types from the <code>Microsoft.Maui.Graphics</code> namespace. For more information, see GraphicsView .
<code>Image</code>	<code>Image</code> displays an image that can be loaded from a local file, a URL, an embedded resource, or a stream. For more information, see Image .
<code>ImageButton</code>	<code>ImageButton</code> displays an image and responds to a tap or click that direct an app to carry out a task. For more information, see ImageButton .
<code>IndicatorView</code>	<code>IndicatorView</code> displays indicators that represent the number of items in a <code>CarouselView</code> . For more information, see IndicatorView .

VIEW	DESCRIPTION
	<code>Label</code> displays single-line and multi-line text. For more information, see Label .
	<code>Line</code> displays a line from a start point to an end point. For more information, see Line .
	<code>ListView</code> displays a scrollable list of selectable data items. For more information, see ListView .
	<code>Path</code> displays curves and complex shapes. For more information, see Path .
	<code>Picker</code> displays a short list of items, from which an item can be selected. For more information, see Picker .
	<code>Polygon</code> displays a polygon. For more information, see Polygon .
	<code>Polyline</code> displays a series of connected straight lines. For more information, see Polyline .
	<code>ProgressBar</code> uses an animation to show that the app is progressing through a lengthy activity. For more information, see ProgressBar .
	<code>RadioButton</code> is a type of button that allows the selection of one option from a set. For more information, see RadioButton .
	<code>Rectangle</code> displays a rectangle or square. For more information, see Rectangle .
	<code>RefreshView</code> is a container control that provides pull-to-refresh functionality for scrollable content. For more information, see RefreshView .
	<code>RoundRectangle</code> displays a rectangle or square with rounded corners. For more information, see Rectangle .
	<code>ScrollView</code> provides scrolling of its content, which is typically a layout. For more information, see ScrollView .
	<code>SearchBar</code> is a user input control used to initiate a search. For more information, see earchBar .
	<code>Slider</code> enables you to select a <code>double</code> value from a continuous range. For more information, see Slider .
	<code>Stepper</code> enables you to select a <code>double</code> value from a range of incremental values. For more information, see Stepper .

VIEW	DESCRIPTION
<code>SwipeView</code>	<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. For more information, see SwipeView .
<code>Switch</code>	<code>Switch</code> enables you to select a boolean value using a type of button that can either be on or off. For more information, see Switch .
<code>TableView</code>	<code>TableView</code> displays a table of scrollable items that can be grouped into sections. For more information, see TableView .
<code>TimePicker</code>	<code>TimePicker</code> enables you to select a time with the platform time picker. For more information, see TimePicker .
<code>WebView</code>	<code>WebView</code> displays web pages or local HTML content. For more information, see WebView .

Align and position .NET MAUI controls

9/20/2022 • 4 minutes to read • [Edit Online](#)

Every .NET Multi-platform App UI (.NET MAUI) control that derives from `View`, which includes views and layouts, has `HorizontalOptions` and `VerticalOptions` properties, of type `LayoutOptions`. The `LayoutOptions` structure encapsulates a view's preferred alignment, which determines its position and size 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).

In addition, the `Margin` and `Padding` properties position controls relative to adjacent, or child controls. For more information, see [Position controls](#).

Align views in layouts

The alignment of a `View`, relative to its parent, can be controlled 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 `Start`, `Center`, `End`, and `Fill`.

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.

NOTE

The default value of a view's `HorizontalOptions` and `VerticalOptions` properties is `LayoutOptions.Fill`.

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.

IMPORTANT

`LayoutOptions.Fill` generally overrides size requests specified using the `HeightRequest` and `WidthRequest` properties.

The following XAML 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>
    ...
    <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 following screenshot shows the resulting alignment of each `Label`:

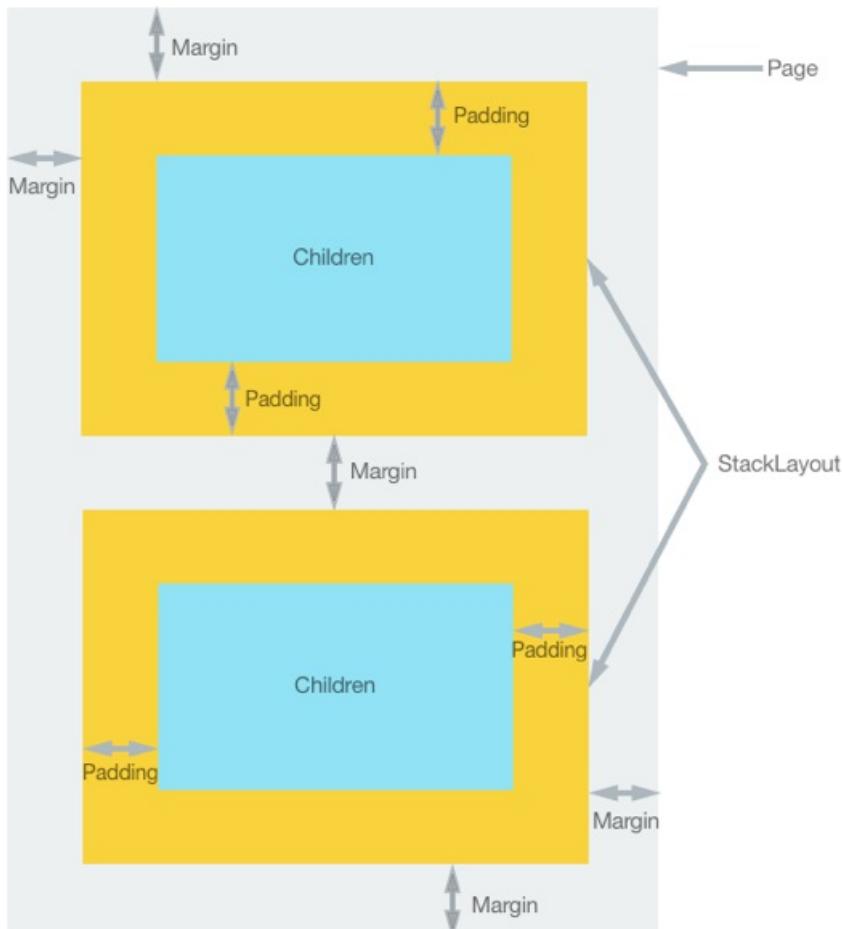


Position controls

The `Margin` and `Padding` properties position controls relative to adjacent, or child controls. 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 layouts and views.
- 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 pages, layouts, and views.

The following diagram illustrates the two concepts:



NOTE

`Margin` values are additive. Therefore, if two adjacent elements specify a margin of 20 device-independent units, the distance between the elements will be 40 device-independent units. In addition, margin and padding values are additive when both are applied, in that the distance between an element and any content will be the margin plus padding.

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 example shows all three possibilities:

```
<StackLayout Padding="0,20,0,0">
    <Label Text=".NET MAUI" Margin="20" />
    <Label Text=".NET iOS" Margin="10,15" />
    <Label Text=".NET Android" Margin="0,20,15,5" />
</StackLayout>
```

The equivalent C# code is:

```
StackLayout stackLayout = new StackLayout
{
    Padding = new Thickness(0,20,0,0)
};
stackLayout.Add(new Label { Text = ".NET MAUI", Margin = new Thickness(20) });
stackLayout.Add(new Label { Text = ".NET iOS", Margin = new Thickness(10,25) });
stackLayout.Add(new Label { Text = ".NET Android", Margin = new Thickness(0,20,15,5) });
```

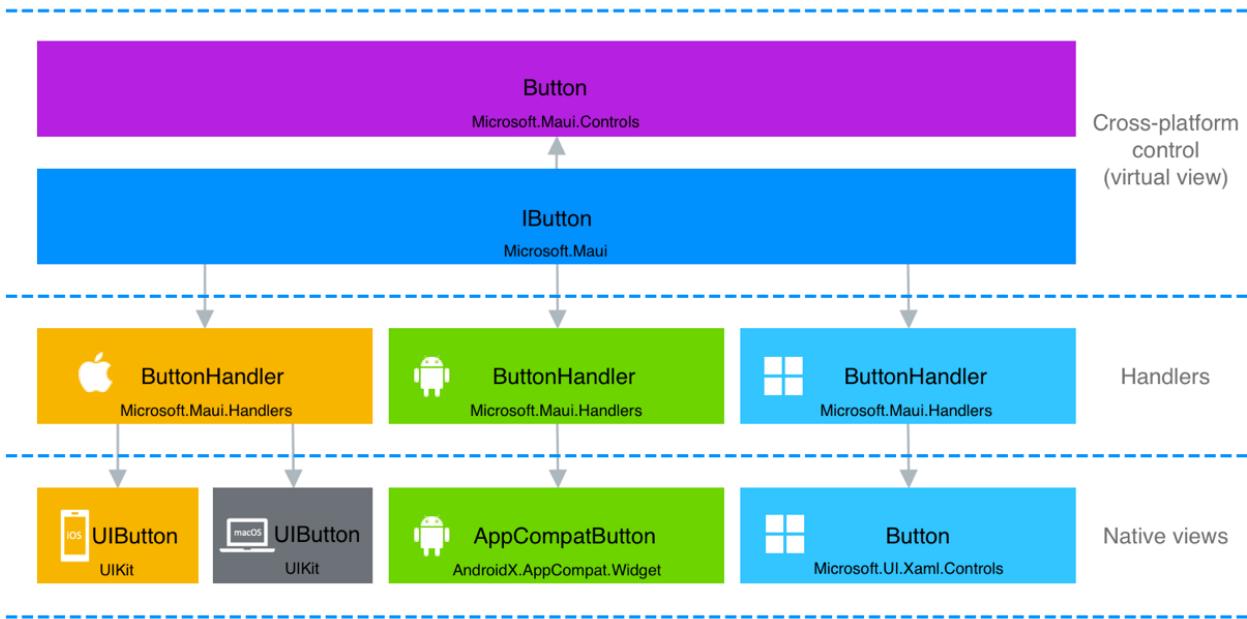
NOTE

`Thickness` values can be negative, which typically clips or overdraws the content.

Handlers

9/20/2022 • 5 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) provides a collection of cross-platform controls that can be used to display data, initiate actions, indicate activity, display collections, pick data, and more. Each control has an interface representation, that abstracts the control. Cross-platform controls that implement these interfaces are known as *virtual views*. *Handlers* map these virtual views to controls on each platform, which are known as *native views*. Handlers are also responsible for instantiating the underlying native view, and mapping the cross-platform control API to the native view API. For example, on iOS a handler maps a .NET MAUI `Button` to an iOS `UIButton`. On Android, the `Button` is mapped to an `AppCompatButton`:



.NET MAUI handlers are accessed through their control-specific interface, such as `IButton` for a `Button`. This avoids the cross-platform control having to reference its handler, and the handler having to reference the cross-platform control.

Each handler class exposes the native view for the cross-platform control via its `PlatformView` property. This property can be accessed to set native view properties, invoke native view methods, and subscribe to native view events. In addition, the cross-platform control implemented by the handler is exposed via its `VirtualView` property.

When creating a cross-platform control, whose implementation is provided on each platform by native views, a handler should be implemented that maps the cross-platform control API to the native view APIs. For more information, see [Create custom controls with handlers](#).

Handlers can also be customized to augment the appearance and behavior of existing cross-platform controls beyond the customization that's possible through the control's API. This handler customization modifies the native views for the cross-platform control. Handlers are global, and customizing a handler for a control will result in all controls of the same type being customized in your app. For more information, see [Customize .NET MAUI controls with handlers](#).

Mappers

A key concept of .NET MAUI handlers is mappers. Each handler typically provides a *property mapper*, and

sometimes a *command mapper*, that maps the cross-platform control's API to the native view's API.

A *property mapper* defines what Actions to take when a property change occurs in the cross-platform control. It's a `Dictionary` that maps the cross-platform control's properties to their associated Actions. Each platform handler then provides implementations of the Actions, which manipulate the native view API. This ensures that when a property is set on a cross-platform control, the underlying native view is updated as required.

A *command mapper* defines what Actions to take when the cross-platform control sends commands to native views. They're similar to property mappers, but allow for additional data to be passed. Commands, in this context, doesn't mean `ICommand` implementations. Instead, a command is just an instruction, and optionally its data, that's sent to a native view. The command mapper is a `Dictionary` that maps the cross-platform control's command to their associated Actions. Each handler then provides implementations of the Actions, which manipulate the native view API. This ensures that when a cross-platform control sends a command to its native view, the native view is updated as required. For example, when a `ScrollView` is scrolled, the `ScrollViewHandler` uses a command mapper to invoke an Action that accepts a scroll position argument. The Action then instructs the underlying native view to scroll to that position.

The advantage of using *mappers* to update native views is that native views can be decoupled from cross-platform controls. This removes the need for native views to subscribe to and unsubscribe from cross-platform control events. It also allows for easy customization because mappers can be modified without subclassing.

Handler lifecycle

All handler-based .NET MAUI controls support two handler lifecycle events:

- `HandlerChanging` is raised when a new handler is about to be created for a cross-platform control, and when an existing handler is about to be removed from a cross-platform control. The `HandlerChangingEventArgs` object that accompanies this event has `NewHandler` and `OldHandler` properties, of type `IElementHandler`. When the `NewHandler` property isn't `null`, the event indicates that a new handler is about to be created for a cross-platform control. When the `OldHandler` property isn't `null`, the event indicates that the existing native control is about to be removed from the cross-platform control, and therefore any native events should be unwired and other cleanup performed.
- `HandlerChanged` is raised after the handler for a cross-platform control has been created. This event indicates that the native control that implements the cross-platform control is available, and all the property values set on the cross-platform control have been applied to the native control.

NOTE

The `HandlerChanging` event is raised on a cross-platform control before the `HandlerChanged` event.

In addition to these events, each cross-platform control also has an overridable `OnHandlerChanging` method that's invoked when the `HandlerChanging` event is raised, and a `OnHandlerChanged` method that's invoked when the `HandlerChanged` event is raised.

Handler-based views

The following table lists the types that implement handler-based views in .NET MAUI:

VIEW	INTERFACE	HANDLER	PROPERTY MAPPER	COMMAND MAPPER
<code>ActivityIndicator</code>	<code>IActivityIndicatorView</code>	<code>ActivityIndicatorHandlerMapper</code>		<code>CommandMapper</code>
<code>BlazorWebView</code>	<code>IBlazorWebView</code>	<code>BlazorWebViewHandler</code>	<code>BlazorWebViewMapper</code>	

VIEW	INTERFACE	HANDLER	PROPERTY MAPPER	COMMAND MAPPER
Border	IBorderView	BorderHandler	Mapper	CommandMapper
Button	IButton	ButtonHandler	ImageButtonMapper .TextButtonMapper , Mapper	CommandMapper
CarouselView		CarouselViewHandler	Mapper	
CheckBox	ICheckBox	CheckBoxHandler	Mapper	CommandMapper
CollectionView		CollectionViewHandler	Mapper	
ContentView	IContentView	ContentViewHandler	Mapper	CommandMapper
DatePicker	IDatePicker	DatePickerHandler	Mapper	CommandMapper
Editor	IEditor	EditorHandler	Mapper	CommandMapper
Ellipse		ShapeViewHandler	Mapper	CommandMapper
Entry	IEntry	EntryHandler	Mapper	CommandMapper
GraphicsView	IGraphicsView	GraphicsViewHandler	Mapper	CommandMapper
Image	IIImage	ImageHandler	Mapper	CommandMapper
ImageButton	IIImageButton	ImageButtonHandler	ImageMapper , Mapper	
IndicatorView	IIndicatorView	IndicatorViewHandler	Mapper	CommandMapper
Label	ILabel	LabelHandler	Mapper	CommandMapper
Line		LineHandler	Mapper	
Path		PathHandler	Mapper	
Picker	IPicker	PickerHandler	Mapper	CommandMapper
Polygon		PolygonHandler	Mapper	
Polyline		PolylineHandler	Mapper	
ProgressBar	IProgress	ProgressBarHandler	Mapper	CommandMapper
RadioButton	IRadioButton	RadioButtonHandler	Mapper	CommandMapper
Rectangle		RectangleHandler	Mapper	

VIEW	INTERFACE	HANDLER	PROPERTY MAPPER	COMMAND MAPPER
RefreshView	IRefreshView	RefreshViewHandler	Mapper	CommandMapper
RoundRectangle		RoundRectangleHandler	Mapper	
ScrollView	IScrollView	ScrollViewHandler	Mapper	CommandMapper
SearchBar	ISearchBar	SearchBarHandler	Mapper	CommandMapper
Slider	ISlider	SliderHandler	Mapper	CommandMapper
Stepper	IStepper	StepperHandler	Mapper	CommandMapper
SwipeView	ISwipeView	SwipeViewHandler	Mapper	CommandMapper
Switch	ISwitch	SwitchHandler	Mapper	CommandMapper
TimePicker	ITimePicker	TimePickerHandler	Mapper	CommandMapper
WebView	IWebView	WebViewHandler	Mapper	CommandMapper

All handlers are in the `Microsoft.Maui.Handlers` namespace, with the following exceptions:

- `CarouselViewHandler` and `CollectionViewHandler` are in the `Microsoft.Maui.Controls.Handlers.Items` namespace.
- `LineHandler`, `PathHandler`, `PolygonHandler`, `PolylineHandler`, `RectangleHandler`, and `RoundRectangleHandler` are in the `Microsoft.Maui.Controls.Handlers` namespace.

The interfaces listed in the table above are in the `Microsoft.Maui` namespace.

Create a custom control using handlers

9/20/2022 • 48 minutes to read • [Edit Online](#)

 [Browse the sample](#)

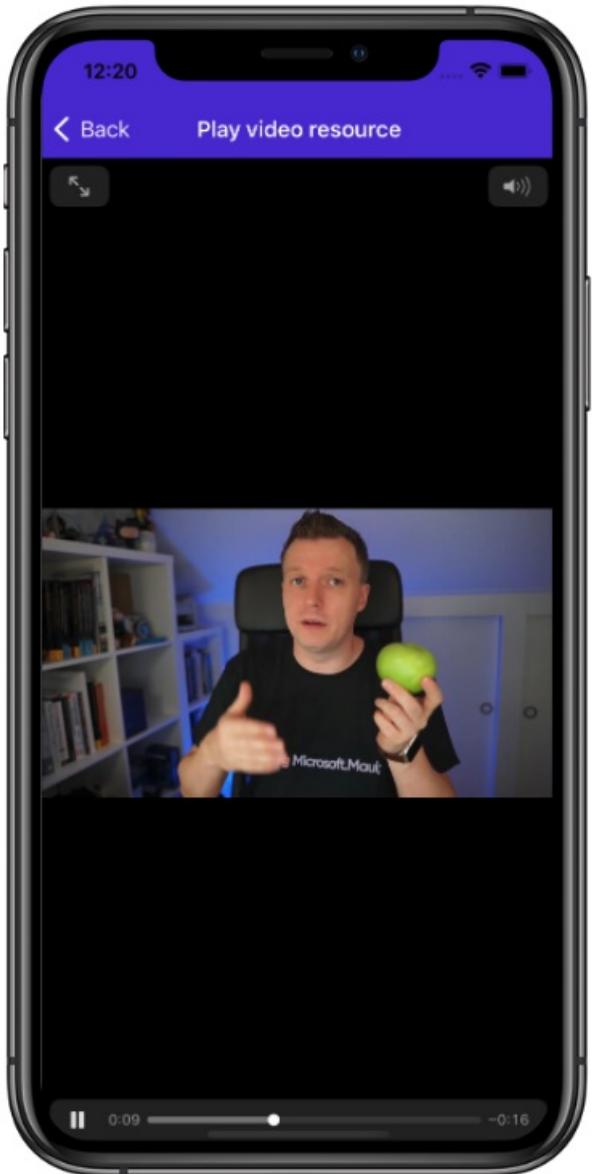
A standard requirement for apps is the ability to play videos. This article examines how to create a .NET Multi-platform App UI (.NET MAUI) cross-platform `video` control that uses a handler to map the cross-platform control API to the native views on Android, iOS, and Mac Catalyst that play videos. This control can play video from three sources:

- A URL, which represents a remote video.
- A resource, which is a file embedded in the app.
- A file, from the device's video library.

IMPORTANT

Unlike Android, iOS, and Mac Catalyst, WinUI 3 currently lacks a control capable of playing video. Therefore the Windows implementation of the `Video` control currently throws a [PlatformNotSupportedException](#).

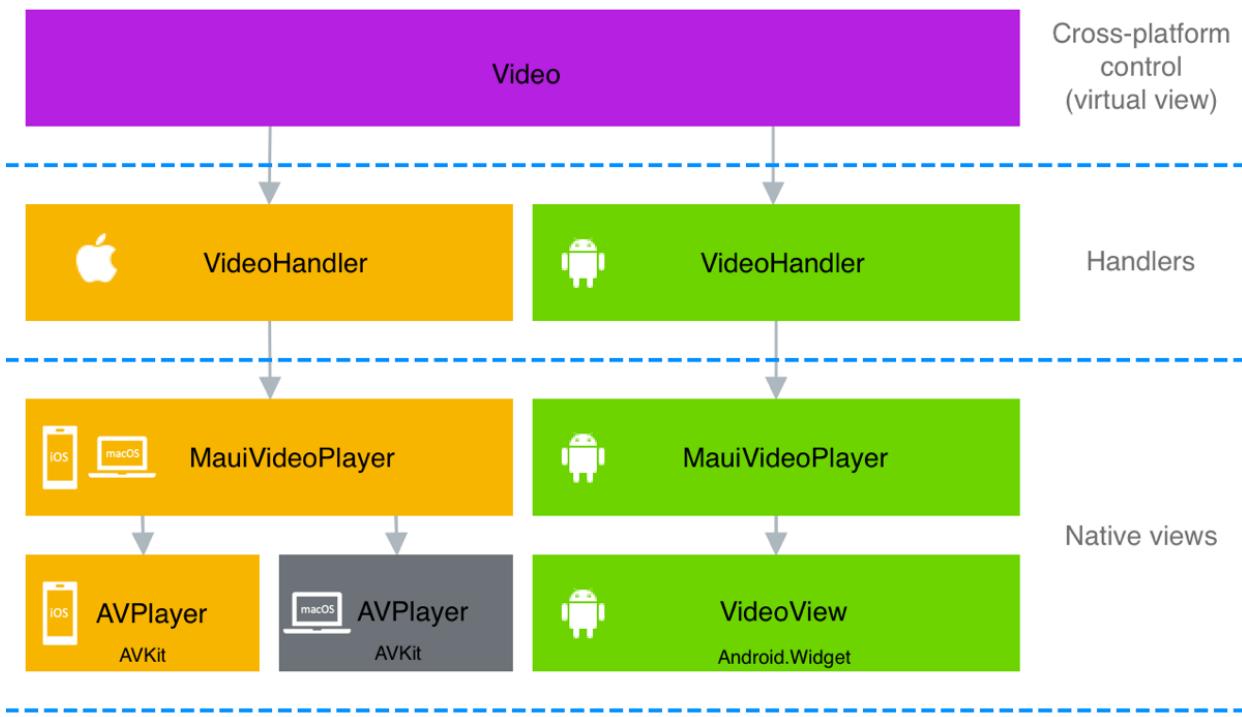
Video controls require *transport controls*, which are buttons for playing and pausing the video, and a positioning bar that shows the progress through the video and allows the user to move quickly to a different location. The `video` control can either use the transport controls and positioning bar provided by the platform, or you can supply custom transport controls and a positioning bar. The following screenshots show the control on iOS, with and without custom transport controls:





A more sophisticated video control would have additional features, such as a volume control, a mechanism to interrupt video playback when a call is received, and a way of keeping the screen active during playback.

The architecture of the `Video` control is shown in the following diagram:



The `Video` class provides the cross-platform API for the control. Mapping of the cross-platform API to the native view APIs is performed by the `VideoHandler` class on each platform, which maps the `Video` class to the `MauiVideoPlayer` class. On iOS and Mac Catalyst, the `MauiVideoPlayer` class uses the `AVPlayer` type to provide video playback. On Android, the `MauiVideoPlayer` class uses the `VideoView` type to provide video playback.

IMPORTANT

.NET MAUI decouples its handlers from its cross-platform controls through interfaces. This enables experimental frameworks such as Comet and Fabulous to provide their own cross-platform controls, that implement the interfaces, while still using .NET MAUI's handlers. Creating an interface for your cross-platform control is only necessary if you need to decouple your handler from its cross-platform control for a similar purpose, or for testing purposes.

The process for creating a cross-platform .NET MAUI custom control, whose platform implementations are provided by handlers, is as follows:

1. Create a class for the cross-platform control, which provides the control's public API. For more information, see [Create the cross-platform control](#).
2. Create any required additional cross-platform types.
3. Create a `partial` handler class. For more information, see [Create the handler](#).
4. In the handler class, create a `PropertyMapper` dictionary, which defines the Actions to take when cross-platform property changes occur. For more information, see [Create the property mapper](#).
5. Optionally, in your handler class, create a `CommandMapper` dictionary, which defines the Actions to take when the cross-platform control sends instructions to the native views that implement the cross-platform control. For more information, see [Create the command mapper](#).
6. Create `partial` handler classes for each platform that create the native views that implement the cross-platform control. For more information, see [Create the platform controls](#).
7. Register the handler using the `ConfigureMauiHandlers` and `AddHandler` methods in your app's `MauiProgram` class. For more information, see [Register the handler](#).

Then, the cross-platform control can be consumed. For more information, see [Consume the cross-platform control](#).

Create the cross-platform control

To create a cross-platform control, you should create a class that derives from `View`:

```
using System.ComponentModel;

namespace VideoDemos.Controls
{
    public class Video : View, IVideoController
    {
        public static readonly BindableProperty AreTransportControlsEnabledProperty =
            BindableProperty.Create(nameof(AreTransportControlsEnabled), typeof(bool), typeof(Video), true);

        public static readonly BindableProperty SourceProperty =
            BindableProperty.Create(nameof(Source), typeof(VideoSource), typeof(Video), null);

        public static readonly BindableProperty AutoPlayProperty =
            BindableProperty.Create(nameof(AutoPlay), typeof(bool), typeof(Video), true);

        public bool AreTransportControlsEnabled
        {
            get { return (bool)GetValue(AreTransportControlsEnabledProperty); }
            set { SetValue(AreTransportControlsEnabledProperty, value); }
        }

        [TypeConverter(typeof(VideoSourceConverter))]
        public VideoSource Source
        {
            get { return (VideoSource)GetValue(SourceProperty); }
            set { SetValue(SourceProperty, value); }
        }

        public bool AutoPlay
        {
            get { return (bool)GetValue(AutoPlayProperty); }
            set { SetValue(AutoPlayProperty, value); }
        }
        ...
    }
}
```

The control should provide a public API that will be accessed by its handler, and control consumers. Cross-platform controls should derive from `View`, which represents a visual element that's used to place layouts and views on the screen.

Create the handler

After creating your cross-platform control, you should create a `partial` class for your handler:

```

#if IOS || MACCATALYST
using PlatformView = VideoDemos.Platforms.MaciOS.MauiVideoPlayer;
#elif ANDROID
using PlatformView = VideoDemos.Platforms.Android.MauiVideoPlayer;
#elif WINDOWS
using PlatformView = Microsoft.UI.Xaml.FrameworkElement;
#endif (NETSTANDARD || !PLATFORM) || (NET6_0 && !IOS && !ANDROID)
using PlatformView = System.Object;
#endif
using VideoDemos.Controls;
using Microsoft.Maui.Handlers;

namespace VideoDemos.Handlers
{
    public partial class VideoHandler
    {
    }
}

```

The handler class is a partial class whose implementation will be completed on each platform with an additional partial class.

The conditional `using` statements define the `PlatformView` type on each platform. On Android, iOS, and Mac Catalyst, the native views are provided by the custom `MauiVideoPlayer` class. On Windows, which currently lacks a video control, there's no video player implementation. However, a native view must be specified for compilation purposes, and this is provided by the `FrameworkElement` class. The final conditional `using` statement defines `PlatformView` to be equal to `System.Object`. This is necessary so that the `PlatformView` type can be used within the handler for usage across all platforms. The alternative would be to have to define the `PlatformView` property once per platform, using conditional compilation.

Create the property mapper

Each handler typically provides a *property mapper*, which defines what Actions to take when a property change occurs in the cross-platform control. The `PropertyMapper` type is a `Dictionary` that maps the cross-platform control's properties to their associated Actions.

`PropertyMapper` is defined in .NET MAUI's generic `ViewHandler` class, and requires two generic arguments to be supplied:

- The class for the cross-platform control, which derives from `View`.
- The class for the handler.

The following code example shows the `VideoHandler` class extended with the `PropertyMapper` definition:

```

public partial class VideoHandler
{
    public static IPropertyMapper<Video, VideoHandler> PropertyMapper = new PropertyMapper<Video,
    VideoHandler>(ViewHandler.ViewMapper)
    {
        [nameof(Video.AreTransportControlsEnabled)] = MapAreTransportControlsEnabled,
        [nameof(Video.Source)] = MapSource,
        [nameof(Video.Position)] = MapPosition
    };

    public VideoHandler() : base(PropertyMapper)
    {
    }
}

```

The `PropertyMapper` is a `Dictionary` whose key is a `string` and whose value is a generic `Action`. The `string` represents the cross-platform control's property name, and the `Action` represents a `static` method that requires the handler and cross-platform control as arguments. For example, the signature of the `MapSource` method is `public static void MapSource(VideoHandler handler, Video video)`.

Each platform handler must provide implementations of the Actions, which manipulate the native view APIs. This ensures that when a property is set on a cross-platform control, the underlying native view will be updated as required. The advantage of this approach is that it allows for easy cross-platform control customization, because the property mapper can be modified by cross-platform control consumers without subclassing.

Create the command mapper

Each handler can also provide a *command mapper*, which defines what Actions to take when the cross-platform control sends commands to native views. Command mappers are similar to property mappers, but allow for additional data to be passed. In this context, a command is an instruction, and optionally its data, that's sent to a native view. The `CommandMapper` type is a `Dictionary` that maps cross-platform control members to their associated Actions.

`CommandMapper` is defined in .NET MAUI's generic `ViewHandler` class, and requires two generic arguments to be supplied:

- The class for the cross-platform control, which derives from `View`.
- The class for the handler.

The following code example shows the `VideoHandler` class extended with the `CommandMapper` definition:

```
public partial class VideoHandler
{
    public static IPropertyMapper<Video, VideoHandler> PropertyMapper = new PropertyMapper<Video,
    VideoHandler>(ViewHandler.ViewMapper)
    {
        [nameof(Video.AreTransportControlsEnabled)] = MapAreTransportControlsEnabled,
        [nameof(Video.Source)] = MapSource,
        [nameof(Video.Position)] = MapPosition
    };

    public static CommandMapper<Video, VideoHandler> CommandMapper = new(ViewCommandMapper)
    {
        [nameof(Video.UpdateStatus)] = MapUpdateStatus,
        [nameof(Video.PlayRequested)] = MapPlayRequested,
        [nameof(Video.PauseRequested)] = MapPauseRequested,
        [nameof(Video.StopRequested)] = MapStopRequested
    };

    public VideoHandler() : base(PropertyMapper, CommandMapper)
    {
    }
}
```

The `CommandMapper` is a `Dictionary` whose key is a `string` and whose value is a generic `Action`. The `string` represents the cross-platform control's command name, and the `Action` represents a `static` method that requires the handler, cross-platform control, and optional data as arguments. For example, the signature of the `MapPlayRequested` method is

```
public static void MapPlayRequested(VideoHandler handler, Video video, object? args).
```

Each platform handler must provide implementations of the Actions, which manipulate the native view APIs. This ensures that when a command is sent from the cross-platform control, the underlying native view will be manipulated as required. The advantage of this approach is that it removes the need for native views to

subscribe to and unsubscribe from cross-platform control events. In addition, it allows for easy customization because the command mapper can be modified by cross-platform control consumers without subclassing.

Create the platform controls

After creating the mappers for your handler, you must provide handler implementations on all platforms. This can be accomplished by adding partial class handler implementations in the child folders of the *Platforms* folder. Alternatively you could configure your project to support filename-based multi-targeting, or folder-based multi-targeting, or both.

The sample app is configured to support filename-based multi-targeting, so that the handler classes all are located in a single folder:



The `videoHandler` class containing the mappers is named `VideoHandler.cs`. Its platform implementations are in the `VideoHandler.Android.cs`, `VideoHandler.MacOS.cs`, and `VideoHandler.Windows.cs` files. This filename-based multi-targeting is configured by adding the following XML to the project file, as children of the `<Project>` node:

```
<!-- Android -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-android')) != true">
  <Compile Remove="**\**\*.Android.cs" />
  <None Include="**\**\*.Android.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- iOS and Mac Catalyst -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-ios')) != true AND
$(TargetFramework.StartsWith('net6.0-maccatalyst')) != true">
  <Compile Remove="**\**\*.MaciOS.cs" />
  <None Include="**\**\*.MaciOS.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Windows -->
<ItemGroup Condition="$(TargetFramework.Contains('-windows')) != true ">
  <Compile Remove="**\*.Windows.cs" />
  <None Include="**\*.Windows.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>
```

For more information about configuring multi-targeting, see [Configure multi-targeting](#).

Each platform handler class should be a partial class and derive from the generic `ViewHandler` class, which requires two type arguments:

- The class for the cross-platform control, which derives from `View`.
- The type of the native view that implements the cross-platform control on the platform. This should be identical to the type of the `PlatformView` property in the handler.

IMPORTANT

The `ViewHandler` class provides `VirtualView` and `PlatformView` properties. The `VirtualView` property is used to access the cross-platform control from its handler. The `PlatformView` property, is used to access the native view on each platform that implements the cross-platform control.

Each of the platform handler implementations should override the following methods:

- `CreatePlatformView`, which should create and return the native view that implements the cross-platform control.
- `ConnectHandler`, which should perform any native view setup, such as initializing the native view and performing event subscriptions.
- `DisconnectHandler`, which should perform any native view cleanup, such as unsubscribing from events and disposing objects.

IMPORTANT

The `DisconnectHandler` method is intentionally not invoked by .NET MAUI. Instead, you must invoke it yourself from a suitable location in your app's lifecycle. For more information, see [Native view cleanup](#).

Each platform handler should also implement the Actions that are defined in the mapper dictionaries.

In addition, each platform handler should also provide code, as required, to implement the functionality of the cross-platform control on the platform. Alternatively, this can be provided by an additional type, which is the approach adopted here.

Android

Video is played on Android with a `videoView`. However, here, the `videoView` has been encapsulated in a `MauiVideoPlayer` type to keep the native view separated from its handler. The following example shows the `VideoHandler` partial class for Android, with its three overrides:

```
#nullable enable
using Microsoft.Maui.Handlers;
using VideoDemos.Controls;
using VideoDemos.Platforms.Android;

namespace VideoDemos.Handlers
{
    public partial class VideoHandler : ViewHandler<Video, MauiVideoPlayer>
    {
        protected override MauiVideoPlayer CreatePlatformView() => new MauiVideoPlayer(Context,
VirtualView);

        protected override void ConnectHandler(MauiVideoPlayer platformView)
        {
            base.ConnectHandler(platformView);

            // Perform any control setup here
        }

        protected override void DisconnectHandler(MauiVideoPlayer platformView)
        {
            platformView.Dispose();
            base.DisconnectHandler(platformView);
        }
        ...
    }
}
```

`VideoHandler` derives from the `ViewHandler` class, with the generic `Video` argument specifying the cross-platform control type, and the `MauiVideoPlayer` argument specifying the type that encapsulates the `VideoView` native view.

The `CreatePlatformView` override creates and returns a `MauiVideoPlayer` object. The `ConnectHandler` override is

the location to perform any required native view setup. The `DisconnectHandler` override is the location to perform any native view cleanup, and so calls the `Dispose` method on the `MauiVideoPlayer` instance.

The platform handler also has to implement the Actions defined in the property mapper dictionary:

```
public partial class VideoHandler : ViewHandler<Video, MauiVideoPlayer>
{
    ...
    public static void MapAreTransportControlsEnabled(VideoHandler handler, Video video)
    {
        handler.PlatformView?.UpdateTransportControlsEnabled();
    }

    public static void MapSource(VideoHandler handler, Video video)
    {
        handler.PlatformView?.UpdateSource();
    }

    public static void MapPosition(VideoHandler handler, Video video)
    {
        handler.PlatformView?.UpdatePosition();
    }
    ...
}
```

Each Action is executed in response to a property changing on the cross-platform control, and is a `static` method that requires handler and cross-platform control instances as arguments. In each case, the Action calls a method defined in the `MauiVideoPlayer` type.

The platform handler also has to implement the Actions defined in the command mapper dictionary:

```

public partial class VideoHandler : ViewHandler<Video, MauiVideoPlayer>
{
    ...
    public static void MapUpdateStatus(VideoHandler handler, Video video, object? args)
    {
        handler.PlatformView?.UpdateStatus();
    }

    public static void MapPlayRequested(VideoHandler handler, Video video, object? args)
    {
        if (args is not VideoPositionEventArgs)
            return;

        TimeSpan position = ((VideoPositionEventArgs)args).Position;
        handler.PlatformView?.PlayRequested(position);
    }

    public static void MapPauseRequested(VideoHandler handler, Video video, object? args)
    {
        if (args is not VideoPositionEventArgs)
            return;

        TimeSpan position = ((VideoPositionEventArgs)args).Position;
        handler.PlatformView?.PauseRequested(position);
    }

    public static void MapStopRequested(VideoHandler handler, Video video, object? args)
    {
        if (args is not VideoPositionEventArgs)
            return;

        TimeSpan position = ((VideoPositionEventArgs)args).Position;
        handler.PlatformView?.StopRequested(position);
    }
    ...
}

```

Each Action is executed in response to a command being sent from the cross-platform control, and is a `static` method that requires handler and cross-platform control instances, and optional data as arguments. In each case, the Action calls a method defined in the `MauiVideoPlayer` class, after extracting the optional data.

On Android, the `MauiVideoPlayer` class encapsulates the `VideoView` to keep the native view separated from its handler:

```

using Android.Content;
using Android.Views;
using Android.Widget;
using AndroidX.CoordinatorLayout.Widget;
using VideoDemos.Controls;
using Color = Android.Graphics.Color;
using Uri = Android.Net.Uri;

namespace VideoDemos.Platforms.Android
{
    public class MauiVideoPlayer : CoordinatorLayout
    {
        VideoView _videoView;
        MediaController _mediaController;
        bool _isPrepared;
        Context _context;
        Video _video;

        public MauiVideoPlayer(Context context, Video video) : base(context)
        {
            _context = context;
            _video = video;

            SetBackgroundColor(Color.Black);

            // Create a RelativeLayout for sizing the video
            RelativeLayout relativeLayout = new RelativeLayout(_context)
            {
                LayoutParameters = new CoordinatorLayout.LayoutParams(LayoutParameters.MatchParent,
                LayoutParameters.MatchParent)
                {
                    Gravity = (int)GravityFlags.Center
                }
            };

            // Create a VideoView and position it in the RelativeLayout
            _videoView = new VideoView(context)
            {
                LayoutParameters = new RelativeLayout.LayoutParams(LayoutParameters.MatchParent,
                LayoutParameters.MatchParent)
            };

            // Add to the layouts
            relativeLayout.AddView(_videoView);
            AddView(relativeLayout);

            // Handle events
            _videoView.Prepared += OnVideoViewPrepared;
        }

        ...
    }
}

```

`MauiVideoPlayer` derives from `CoordinatorLayout`, because the root native view in a .NET MAUI app on Android is `CoordinatorLayout`. While the `MauiVideoPlayer` class could derive from other native Android types, it can be difficult to control native view positioning in some scenarios.

The `videoView` could be added directly to the `CoordinatorLayout`, and positioned in the layout as required. However, here, an Android `RelativeLayout` is added to the `CoordinatorLayout`, and the `VideoView` is added to the `RelativeLayout`. Layout parameters are set on both the `RelativeLayout` and `VideoView` so that the `VideoView` is centered in the page, and expands to fill the available space while maintaining its aspect ratio.

The constructor also subscribes to the `VideoView.Prepared` event. This event is raised when the video is ready for playback, and is unsubscribed from in the `Dispose` override:

```
public class MauiVideoPlayer : CoordinatorLayout
{
    VideoView _videoView;
    Video _video;
    ...

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            _videoView.Prepared -= OnVideoViewPrepared;
            _videoView.Dispose();
            _videoView = null;
            _video = null;
        }

        base.Dispose(disposing);
    }
    ...
}
```

In addition to unsubscribing from the `Prepared` event, the `Dispose` override also performs native view cleanup.

NOTE

The `Dispose` override is called by the handler's `DisconnectHandler` override.

The platform transport controls include buttons that play, pause, and stop the video, and are provided by Android's `MediaController` type. If the `Video.AreTransportControlsEnabled` property is set to `true`, a `MediaController` is set as the media player of the `VideoView`. This occurs because when the `AreTransportControlsEnabled` property is set, the handler's property mapper ensures that the `MapAreTransportControlsEnabled` method is invoked, which in turn calls the `updateTransportControlsEnabled` method in `MauiVideoPlayer`:

```
public class MauiVideoPlayer : CoordinatorLayout
{
    VideoView _videoView;
    MediaController _mediaController;
    Video _video;
    ...

    public void UpdateTransportControlsEnabled()
    {
        if (_video.AreTransportControlsEnabled)
        {
            _mediaController = new MediaController(_context);
            _mediaController.SetMediaPlayer(_videoView);
            _videoView.SetMediaController(_mediaController);
        }
        else
        {
            _videoView.SetMediaController(null);
            if (_mediaController != null)
            {
                _mediaController.SetMediaPlayer(null);
                _mediaController = null;
            }
        }
    }
    ...
}
```

The transport controls fade out if they're not used but can be restored by tapping on the video.

If the `Video.AreTransportControlsEnabled` property is set to `false`, the `MediaController` is removed as the media player of the `VideoView`. In this scenario, you can then control video playback programmatically or supply your own transport controls. For more information, see [Create custom transport controls](#).

iOS and Mac Catalyst

Video is played on iOS and Mac Catalyst with an `AVPlayer` and an `AVPlayerViewController`. However, here, these types are encapsulated in a `MauiVideoPlayer` type to keep the native views separated from their handler. The following example shows the `VideoHandler` partial class for iOS, with its three overrides:

```

using Microsoft.Maui.Handlers;
using VideoDemos.Controls;
using VideoDemos.Platforms.MacOS;

namespace VideoDemos.Handlers
{
    public partial class VideoHandler : ViewHandler<Video, MauiVideoPlayer>
    {
        protected override MauiVideoPlayer CreatePlatformView() => new MauiVideoPlayer(VirtualView);

        protected override void ConnectHandler(MauiVideoPlayer platformView)
        {
            base.ConnectHandler(platformView);

            // Perform any control setup here
        }

        protected override void DisconnectHandler(MauiVideoPlayer platformView)
        {
            platformView.Dispose();
            base.DisconnectHandler(platformView);
        }

        ...
    }
}

```

`VideoHandler` derives from the `ViewHandler` class, with the generic `Video` argument specifying the cross-platform control type, and the `MauiVideoPlayer` argument specifying the type that encapsulates the `AVPlayer` and `AVPlayerViewController` native views.

The `CreatePlatformView` override creates and returns a `MauiVideoPlayer` object. The `ConnectHandler` override is the location to perform any required native view setup. The `DisconnectHandler` override is the location to perform any native view cleanup, and so calls the `Dispose` method on the `MauiVideoPlayer` instance.

The platform handler also has to implement the Actions defined in the property mapper dictionary:

```

public partial class VideoHandler : ViewHandler<Video, MauiVideoPlayer>
{
    ...

    public static void MapAreTransportControlsEnabled(VideoHandler handler, Video video)
    {
        handler?.PlatformView.UpdateTransportControlsEnabled();
    }

    public static void MapSource(VideoHandler handler, Video video)
    {
        handler?.PlatformView.UpdateSource();
    }

    public static void MapPosition(VideoHandler handler, Video video)
    {
        handler?.PlatformView.UpdatePosition();
    }

    ...
}

```

Each Action is executed in response to a property changing on the cross-platform control, and is a `static` method that requires handler and cross-platform control instances as arguments. In each case, the Action calls a method defined in the `MauiVideoPlayer` type.

The platform handler also has to implement the Actions defined in the command mapper dictionary:

```

public partial class VideoHandler : ViewHandler<Video, MauiVideoPlayer>
{
    ...
    public static void MapUpdateStatus(VideoHandler handler, Video video, object? args)
    {
        handler.PlatformView?.UpdateStatus();
    }

    public static void MapPlayRequested(VideoHandler handler, Video video, object? args)
    {
        if (args is not VideoPositionEventArgs)
            return;

        TimeSpan position = ((VideoPositionEventArgs)args).Position;
        handler.PlatformView?.PlayRequested(position);
    }

    public static void MapPauseRequested(VideoHandler handler, Video video, object? args)
    {
        if (args is not VideoPositionEventArgs)
            return;

        TimeSpan position = ((VideoPositionEventArgs)args).Position;
        handler.PlatformView?.PauseRequested(position);
    }

    public static void MapStopRequested(VideoHandler handler, Video video, object? args)
    {
        if (args is not VideoPositionEventArgs)
            return;

        TimeSpan position = ((VideoPositionEventArgs)args).Position;
        handler.PlatformView?.StopRequested(position);
    }
    ...
}

```

Each Action is executed in response to a command being sent from the cross-platform control, and is a `static` method that requires handler and cross-platform control instances, and optional data as arguments. In each case, the Action calls a method defined in the `MauiVideoPlayer` class, after extracting the optional data.

On iOS and Mac Catalyst, the `MauiVideoPlayer` class encapsulates the `AVPlayer` and `AVPlayerViewController` types to keep the native views separated from their handler:

```

using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using System.Diagnostics;
using UIKit;
using VideoDemos.Controls;

namespace VideoDemos.Platforms.MacOS
{
    public class MauiVideoPlayer : UIView
    {
        AVPlayer _player;
        AVPlayerViewController _playerViewController;
        Video _video;
        ...

        public MauiVideoPlayer(Video video)
        {
            _video = video;

            // Create AVPlayerViewController
            _playerViewController = new AVPlayerViewController();

            // Set Player property to AVPlayer
            _player = new AVPlayer();
            _playerViewController.Player = _player;

            // Use the View from the controller as the native view
            _playerViewController.View.Frame = this.Bounds;
            AddSubview(_playerViewController.View);
        }
        ...
    }
}

```

`MauiVideoPlayer` derives from `UIView`, which is the base class on iOS and Mac Catalyst for objects that display content and handle user interaction with that content. The constructor creates an `AVPlayer` object, which manages the playback and timing of a media file, and sets it as the `Player` property value of an `AVPlayerViewController`. The `AVPlayerViewController` displays content from the `AVPlayer` and presents transport controls and other features. The size and location of the control is then set, which ensures that the video is centered in the page and expands to fill the available space while maintaining its aspect ratio. The native view, which is the view from the `AVPlayerViewController`, is then added to the page.

The `Dispose` method is responsible for performing native view cleanup:

```

public class MauiVideoPlayer : UIView
{
    AVPlayer _player;
    AVPlayerViewController _playerViewController;
    Video _video;
    ...

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (_player != null)
            {
                _player.ReplaceCurrentItemWithPlayerItem(null);
                _player.Dispose();
            }
            if (_playerViewController != null)
                _playerViewController.Dispose();

            _video = null;
        }

        base.Dispose(disposing);
    }
    ...
}

```

In some scenarios, videos continue playing after a video playback page has been navigated away from. To stop the video, the `ReplaceCurrentItemWithPlayerItem` is set to `null` in the `Dispose` override, and other native view cleanup is performed.

NOTE

The `Dispose` override is called by the handler's `DisconnectHandler` override.

The platform transport controls include buttons that play, pause, and stop the video, and are provided by the `AVPlayerViewController` type. If the `Video.AreTransportControlsEnabled` property is set to `true`, the `AVPlayerViewController` will display its playback controls. This occurs because when the `AreTransportControlsEnabled` property is set, the handler's property mapper ensures that the `MapAreTransportControlsEnabled` method is invoked, which in turn calls the `UpdateTransportControlsEnabled` method in `MauiVideoPlayer`:

```

public class MauiVideoPlayer : UIView
{
    AVPlayerViewController _playerViewController;
    Video _video;
    ...

    public void UpdateTransportControlsEnabled()
    {
        _playerViewController.ShowsPlaybackControls = _video.AreTransportControlsEnabled;
    }
    ...
}

```

The transport controls fade out if they're not used but can be restored by tapping on the video.

If the `Video.AreTransportControlsEnabled` property is set to `false`, the `AVPlayerViewController` doesn't show its playback controls. In this scenario, you can then control video playback programmatically or supply your own

transport controls. For more information, see [Create custom transport controls](#).

Windows

WinUI 3 currently lacks a control capable of playing video. However, it's still necessary to provide a handler implementation on Windows that overrides the `CreatePlatformView` method and that provides methods for the Actions defined in the property mapper and command mapper dictionaries. In all cases, these methods throw `PlatformNotSupportedException` exceptions:

```
using Microsoft.Maui.Handlers;
using Microsoft.UI.Xaml;
using VideoDemos.Controls;

namespace VideoDemos.Handlers
{
    public partial class VideoHandler : ViewHandler<Video, FrameworkElement>
    {
        protected override FrameworkElement CreatePlatformView() => throw new
PlatformNotSupportedException("No MediaElement control on Windows.");
        public static void MapAreTransportControlsEnabled(VideoHandler handler, Video video) => throw new
PlatformNotSupportedException("No MediaElement control on Windows.");
        public static void MapSource(VideoHandler handler, Video video) => throw new
PlatformNotSupportedException("No MediaElement control on Windows.");
        public static void MapPosition(VideoHandler handler, Video video) => throw new
PlatformNotSupportedException("No MediaElement control on Windows.");
        public static void MapUpdateStatus(VideoHandler handler, Video video, object? arg) => throw new
PlatformNotSupportedException("No MediaElement control on Windows.");
        public static void MapPlayRequested(VideoHandler handler, Video video, object? arg) => throw new
PlatformNotSupportedException("No MediaElement control on Windows.");
        public static void MapPauseRequested(VideoHandler handler, Video video, object? arg) => throw new
PlatformNotSupportedException("No MediaElement control on Windows.");
        public static void MapStopRequested(VideoHandler handler, Video video, object? arg) => throw new
PlatformNotSupportedException("No MediaElement control on Windows.");
    }
}
```

A type must still be specified that represents the native view on Windows for the `Video` control, and this is provided here by the `FrameworkElement` class, which is a base type for WinUI controls.

Play a video

The `video` class defines a `Source` property, which is used to specify the source of the video file, and an `AutoPlay` property. `AutoPlay` defaults to `true`, which means that the video should begin playing automatically after `Source` has been set. For the definition of these properties, see [Create the cross-platform control](#).

The `Source` property is of type `VideoSource`, which is an abstract class that consists of three static methods that instantiate the three classes that derive from `VideoSource`:

```

using System.ComponentModel;

namespace VideoDemos.Controls
{
    [TypeConverter(typeof(VideoSourceConverter))]
    public abstract class VideoSource : Element
    {
        public static VideoSource FromUri(string uri)
        {
            return new UriVideoSource { Uri = uri };
        }

        public static VideoSourceFromFile(string file)
        {
            return new FileVideoSource { File = file };
        }

        public static VideoSource FromResource(string path)
        {
            return new ResourceVideoSource { Path = path };
        }
    }
}

```

The `VideoSource` class includes a `TypeConverter` attribute that references `VideoSourceConverter`:

```

using System.ComponentModel;

namespace VideoDemos.Controls
{
    public class VideoSourceConverter : TypeConverter, IExtendedTypeConverter
    {
        object IExtendedTypeConverter.ConvertFromInvariantString(string value, IServiceProvider serviceProvider)
        {
            if (!string.IsNullOrWhiteSpace(value))
            {
                Uri uri;
                return Uri.TryCreate(value, UriKind.Absolute, out uri) && uri.Scheme != "file" ?
                    VideoSource.FromUri(value) : VideoSource.FromResource(value);
            }
            throw new InvalidOperationException("Cannot convert null or whitespace to VideoSource.");
        }
    }
}

```

The type converter is invoked when the `Source` property is set to a string in XAML. The `ConvertFromInvariantString` method attempts to convert the string to a `Uri` object. If it succeeds, and the scheme isn't `file`, then the method returns a `UriVideoSource`. Otherwise it returns a `ResourceVideoSource`.

Play a web video

The `UriVideoSource` class is used to specify a remote video with a URI. It defines a `Uri` property of type `string`:

```
namespace VideoDemos.Controls
{
    public class UriVideoSource : VideoSource
    {
        public static readonly BindableProperty UriProperty =
            BindableProperty.Create(nameof(Uri), typeof(string), typeof(UriVideoSource));

        public string Uri
        {
            get { return (string)GetValue(UriProperty); }
            set { SetValue(UriProperty, value); }
        }
    }
}
```

When the `Source` property is set to a `UriVideoSource`, the handler's property mapper ensures that the `MapSource` method is invoked:

```
public static void MapSource(VideoHandler handler, Video video)
{
    handler?.PlatformView.UpdateSource();
}
```

The `MapSource` method in turns calls the `UpdateSource` method on the handler's `PlatformView` property. The `PlatformView` property, which is of type `MauiVideoPlayer`, represents the native view that provides the video player implementation on each platform.

Android

Video is played on Android with a `videoView`. The following code example shows how the `UpdateSource` method processes the `Source` property when it's of type `UriVideoSource`:

```

using Android.Content;
using Android.Views;
using Android.Widget;
using AndroidX.CoordinatorLayout.Widget;
using VideoDemos.Controls;
using Color = Android.Graphics.Color;
using Uri = Android.Net.Uri;

namespace VideoDemos.Platforms.Android
{
    public class MauiVideoPlayer : CoordinatorLayout
    {
        VideoView _videoView;
        bool _isPrepared;
        Video _video;
        ...

        public void UpdateSource()
        {
            _isPrepared = false;
            bool hasSetSource = false;

            if (_video.Source is UriVideoSource)
            {
                string uri = (_video.Source as UriVideoSource).Uri;
                if (!string.IsNullOrWhiteSpace(uri))
                {
                    _videoView.SetVideoURI(Uri.Parse(uri));
                    hasSetSource = true;
                }
            }
            ...

            if (hasSetSource && _video.AutoPlay)
            {
                _videoView.Start();
            }
            ...
        }
    }
}

```

When processing objects of type `UriVideoSource`, the `SetVideoUri` method of `VideoView` is used to specify the video to be played, with an Android `Uri` object created from the string URI.

The `AutoPlay` property has no equivalent on `VideoView`, so the `Start` method is called if a new video has been set.

iOS and Mac Catalyst

To play a video on iOS and Mac Catalyst, an object of type `AVAsset` is created to encapsulate the video, and that is used to create an `AVPlayerItem`, which is then handed off to the `AVPlayer` object. The following code example shows how the `UpdateSource` method processes the `Source` property when it's of type `UriVideoSource`:

```

using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using System.Diagnostics;
using UIKit;
using VideoDemos.Controls;

namespace VideoDemos.Platforms.MacOS
{
    public class MauiVideoPlayer : UIView
    {
        AVPlayer _player;
        AVPlayerItem _playerItem;
        Video _video;
        ...

        public void UpdateSource()
        {
            AVAsset asset = null;

            if (_video.Source is UriVideoSource)
            {
                string uri = (_video.Source as UriVideoSource).Uri;
                if (!string.IsNullOrWhiteSpace(uri))
                    asset = AVAsset.FromUrl(new NSUrl(uri));
            }
            ...

            if (asset != null)
                _playerItem = new AVPlayerItem(asset);
            else
                _playerItem = null;

            _player.ReplaceCurrentItemWithPlayerItem(_playerItem);
            if (_playerItem != null && _video.AutoPlay)
            {
                _player.Play();
            }
        }
        ...
    }
}

```

When processing objects of type `UriVideoSource`, the static `AVAsset.FromUrl` method is used to specify the video to be played, with an iOS `NSURL` object created from the string URI.

The `AutoPlay` property has no equivalent in the iOS video classes, so the property is examined at the end of the `UpdateSource` method to call the `Play` method on the `AVPlayer` object.

In some cases on iOS, videos continue playing after the video playback page has been navigated away from. To stop the video, the `ReplaceCurrentItemWIthPlayerItem` is set to `null` in the `Dispose` override:

```

protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        if (_player != null)
        {
            _player.ReplaceCurrentItemWithPlayerItem(null);
            ...
        }
        ...
    }
    base.Dispose(disposing);
}

```

Play a video resource

The `ResourceVideoSource` class is used to access video files that are embedded in the app. It defines a `Path` property of type `string`:

```

namespace VideoDemos.Controls
{
    public class ResourceVideoSource : VideoSource
    {
        public static readonly BindableProperty PathProperty =
            BindableProperty.Create(nameof(Path), typeof(string), typeof(ResourceVideoSource));

        public string Path
        {
            get { return (string)GetValue(PathProperty); }
            set { SetValue(PathProperty, value); }
        }
    }
}

```

When the `Source` property is set to a `ResourceVideoSource`, the handler's property mapper ensures that the `MapSource` method is invoked:

```

public static void MapSource(VideoHandler handler, Video video)
{
    handler?.PlatformView.UpdateSource();
}

```

The `MapSource` method in turns calls the `UpdateSource` method on the handler's `PlatformView` property. The `PlatformView` property, which is of type `MauiVideoPlayer`, represents the native view that provides the video player implementation on each platform.

Android

The following code example shows how the `UpdateSource` method processes the `Source` property when it's of type `ResourceVideoSource`:

```

using Android.Content;
using Android.Views;
using Android.Widget;
using AndroidX.CoordinatorLayout.Widget;
using VideoDemos.Controls;
using Color = Android.Graphics.Color;
using Uri = Android.Net.Uri;

namespace VideoDemos.Platforms.Android
{
    public class MauiVideoPlayer : CoordinatorLayout
    {
        VideoView _videoView;
        bool _isPrepared;
        Context _context;
        Video _video;
        ...

        public void UpdateSource()
        {
            _isPrepared = false;
            bool hasSetSource = false;
            ...

            else if (_video.Source is ResourceVideoSource)
            {
                string package = Context.PackageName;
                string path = (_video.Source as ResourceVideoSource).Path;
                if (!string.IsNullOrWhiteSpace(path))
                {
                    string assetFilePath = "content://" + package + "/" + path;
                    _videoView.SetVideoPath(assetFilePath);
                    hasSetSource = true;
                }
            }
            ...
        }
        ...
    }
}

```

When processing objects of type `ResourceVideoSource`, the `SetVideoPath` method of `videoView` is used to specify the video to be played, with a string argument combining the app's package name with the video's filename.

A resource video file is stored in the package's *assets* folder, and requires a content provider to access it. The content provider is provided by the `VideoProvider` class, which creates an `AssetFileDescriptor` object that provides access to the video file:

```

using Android.Content;
using Android.Content.Res;
using Android.Database;
using Debug = System.Diagnostics.Debug;
using Uri = Android.Net.Uri;

namespace VideoDemos.Platforms.Android
{
    [ContentProvider(new string[] { "com.companyname.videodemos" })]
    public class VideoProvider : ContentProvider
    {
        public override AssetFileDescriptor OpenAssetFile(Uri uri, string mode)
        {
            var assets = Context.Assets;
            string fileName = uri.LastPathSegment;
            if (fileName == null)
                throw new FileNotFoundException();

            AssetFileDescriptor afd = null;
            try
            {
                afd = assets.OpenFd(fileName);
            }
            catch (IOException ex)
            {
                Debug.WriteLine(ex);
            }
            return afd;
        }

        public override bool OnCreate()
        {
            return false;
        }
        ...
    }
}

```

iOS and Mac Catalyst

The following code example shows how the `UpdateSource` method processes the `Source` property when it's of type `ResourceVideoSource`:

```

using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using System.Diagnostics;
using UIKit;
using VideoDemos.Controls;

namespace VideoDemos.Platforms.MacOS
{
    public class MauiVideoPlayer : UIView
    {
        Video _video;
        ...

        public void UpdateSource()
        {
            AVAsset asset = null;
            ...

            else if (_video.Source is ResourceVideoSource)
            {
                string path = (_video.Source as ResourceVideoSource).Path;
                if (!string.IsNullOrWhiteSpace(path))
                {
                    string directory = Path.GetDirectoryName(path);
                    string filename = Path.GetFileNameWithoutExtension(path);
                    string extension = Path.GetExtension(path).Substring(1);
                    NSUrl url = NSBundle.MainBundle.GetUrlForResource(filename, extension, directory);
                    asset = AVAsset.FromUrl(url);
                }
            }
            ...
        }
        ...
    }
}

```

When processing objects of type `FileVideoSource`, the `GetUrlForResource` method of `NSBundle` is used to retrieve the file from the app package. The complete path must be divided into a filename, extension, and directory.

In some cases on iOS, videos continue playing after the video playback page has been navigated away from. To stop the video, the `ReplaceCurrentItemWithPlayerItem` is set to `null` in the `Dispose` override:

```

protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        if (_player != null)
        {
            _player.ReplaceCurrentItemWithPlayerItem(null);
            ...
        }
        ...
    }
    base.Dispose(disposing);
}

```

Play a video file from the device's library

The `FileVideoSource` class is used to access videos in the device's video library. It defines a `File` property of type `string`:

```
namespace VideoDemos.Controls
{
    public class FileVideoSource : VideoSource
    {
        public static readonly BindableProperty FileProperty =
            BindableProperty.Create(nameof(File), typeof(string), typeof(FileVideoSource));

        public string File
        {
            get { return (string)GetValue(FileProperty); }
            set { SetValue(FileProperty, value); }
        }
    }
}
```

When the `Source` property is set to a `FileVideoSource`, the handler's property mapper ensures that the `MapSource` method is invoked:

```
public static void MapSource(VideoHandler handler, Video video)
{
    handler?.PlatformView.UpdateSource();
}
```

The `MapSource` method in turns calls the `UpdateSource` method on the handler's `PlatformView` property. The `PlatformView` property, which is of type `MauiVideoPlayer`, represents the native view that provides the video player implementation on each platform.

Android

The following code example shows how the `updateSource` method processes the `Source` property when it's of type `FileVideoSource`:

```

using Android.Content;
using Android.Views;
using Android.Widget;
using AndroidX.CoordinatorLayout.Widget;
using VideoDemos.Controls;
using Color = Android.Graphics.Color;
using Uri = Android.Net.Uri;

namespace VideoDemos.Platforms.Android
{
    public class MauiVideoPlayer : CoordinatorLayout
    {
        VideoView _videoView;
        bool _isPrepared;
        Video _video;
        ...

        public void UpdateSource()
        {
            _isPrepared = false;
            bool hasSetSource = false;
            ...

            else if (_video.Source is FileVideoSource)
            {
                string filename = (_video.Source as FileVideoSource).File;
                if (!string.IsNullOrWhiteSpace(filename))
                {
                    _videoView.SetVideoPath(filename);
                    hasSetSource = true;
                }
            }
            ...
        }
        ...
    }
}

```

When processing objects of type `FileVideoSource`, the `SetVideoPath` method of `VideoView` is used to specify the video file to be played.

iOS and Mac Catalyst

The following code example shows how the `UpdateSource` method processes the `Source` property when it's of type `FileVideoSource`:

```

using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using System.Diagnostics;
using UIKit;
using VideoDemos.Controls;

namespace VideoDemos.Platforms.MacOS
{
    public class MauiVideoPlayer : UIView
    {
        Video _video;
        ...

        public void UpdateSource()
        {
            AVAsset asset = null;
            ...

            else if (_video.Source is FileVideoSource)
            {
                string uri = (_video.Source as FileVideoSource).File;
                if (!string.IsNullOrWhiteSpace(uri))
                    asset = AVAsset.FromUrl(new NSUrl(uri));
            }
            ...
        }
        ...
    }
}

```

When processing objects of type `FileVideoSource`, the static `AVAsset.FromUrl` method is used to specify the video file to be played, with an iOS `NSURL` object created from the string URI.

Create custom transport controls

The transport controls of a video player include buttons that play, pause, and stop the video. These buttons are often identified with familiar icons rather than text, and the play and pause buttons are often combined into one button.

By default, the `Video` control displays transport controls supported by each platform. However, when you set the `AreTransportControlsEnabled` property to `false`, these controls are suppressed. You can then control video playback programmatically or supply your own transport controls.

Implementing your own transport controls requires the `Video` class to be able to notify its native views to play, pause, or stop the video, and know the current status of video playback. The `Video` class defines methods named `Play`, `Pause`, and `Stop` that raise a corresponding event, and send a command to the `VideoHandler`:

```

namespace VideoDemos.Controls
{
    public class Video : View, IVideoController
    {
        ...
        public event EventHandler<VideoPositionEventArgs> PlayRequested;
        public event EventHandler<VideoPositionEventArgs> PauseRequested;
        public event EventHandler<VideoPositionEventArgs> StopRequested;

        public void Play()
        {
            VideoPositionEventArgs args = new VideoPositionEventArgs(Position);
            PlayRequested?.Invoke(this, args);
            Handler?.Invoke(nameof(Video.PlayRequested), args);
        }

        public void Pause()
        {
            VideoPositionEventArgs args = new VideoPositionEventArgs(Position);
            PauseRequested?.Invoke(this, args);
            Handler?.Invoke(nameof(Video.PauseRequested), args);
        }

        public void Stop()
        {
            VideoPositionEventArgs args = new VideoPositionEventArgs(Position);
            StopRequested?.Invoke(this, args);
            Handler?.Invoke(nameof(Video.StopRequested), args);
        }
    }
}

```

The `videoPositionEventArgs` class defines a `Position` property that can be set through its constructor. This property represents the position at which video playback was started, paused, or stopped.

The final line in the `Play`, `Pause`, and `Stop` methods sends a command and associated data to `VideoHandler`. The `CommandMapper` for `VideoHandler` maps command names to Actions that are executed when a command is received. For example, when `VideoHandler` receives the `PlayRequested` command, it executes its `MapPlayRequested` method. The advantage of this approach is that it removes the need for native views to subscribe to and unsubscribe from cross-platform control events. In addition, it allows for easy customization because the command mapper can be modified by cross-platform control consumers without subclassing. For more information about `CommandMapper`, see [Create the command mapper](#).

The `MauiVideoPlayer` implementation on Android, iOS and Mac Catalyst, has `PlayRequested`, `PauseRequested`, and `StopRequested` methods that are executed in response to the `Video` control sending `PlayRequested`, `PauseRequested`, and `StopRequested` commands. Each method invokes a method on its native view to play, pause, or stop the video. For example, the following code shows the `PlayRequested`, `PauseRequested`, and `StopRequested` methods on iOS and Mac Catalyst:

```

using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using System.Diagnostics;
using UIKit;
using VideoDemos.Controls;

namespace VideoDemos.Platforms.MacOS
{
    public class MauiVideoPlayer : UIView
    {
        AVPlayer _player;
        ...

        public void PlayRequested(TimeSpan position)
        {
            _player.Play();
            Debug.WriteLine($"Video playback from {position.Hours}:{position.Minutes}:{position.Seconds}.");
        }

        public void PauseRequested(TimeSpan position)
        {
            _player.Pause();
            Debug.WriteLine($"Video paused at {position.Hours}:{position.Minutes}:{position.Seconds}.");
        }

        public void StopRequested(TimeSpan position)
        {
            _player.Pause();
            _player.Seek(new CMTime(0, 1));
            Debug.WriteLine($"Video stopped at {position.Hours}:{position.Minutes}:{position.Seconds}.");
        }
    }
}

```

Each of the three methods logs the position at which the video was played, paused, or stopped, using the data that's sent with the command.

This mechanism ensures that when the `Play`, `Pause`, or `Stop` method is invoked on the `Video` control, its native view is instructed to play, pause, or stop the video and log the position at which the video was played, paused, or stopped. This all happens using a decoupled approach, without native views having to subscribe to cross-platform events.

Video status

Implementing play, pause, and stop functionality isn't sufficient for supporting custom transport controls. Often the play and pause functionality should be implemented with the same button, which changes its appearance to indicate whether the video is currently playing or paused. In addition, the button shouldn't even be enabled if the video hasn't yet loaded.

These requirements imply that that video player needs to make available a current status indicating if it's playing or paused, or if it's not yet ready to play a video. This status can be represented by an enumeration:

```

public enum VideoStatus
{
    NotReady,
    Playing,
    Paused
}

```

The `Video` class defines a read-only bindable property named `Status` of type `VideoStatus`. This property is defined as read-only because it should only be set from the control's handler:

```

namespace VideoDemos.Controls
{
    public class Video : View, IVideoController
    {
        ...
        private static readonly BindablePropertyKey StatusPropertyKey =
            BindableProperty.CreateReadOnly(nameof(Status), typeof(VideoStatus), typeof(Video),
            VideoStatus.NotReady);

        public static readonly BindableProperty StatusProperty = StatusPropertyKey.BindableProperty;

        public VideoStatus Status
        {
            get { return (VideoStatus)GetValue(StatusProperty); }
        }

        IVideoController.Status
        {
            get { return Status; }
            set { SetValue(StatusPropertyKey, value); }
        }
        ...
    }
}

```

Usually, a read-only bindable property would have a private `set` accessor on the `Status` property to allow it to be set from within the class. However, for a `View` derivative supported by handlers, the property must be set from outside the class but only by the control's handler.

For this reason, another property is defined with the name `IVideoController.Status`. This is an explicit interface implementation, and is made possible by the `IVideoController` interface that the `Video` class implements:

```

public interface IVideoController
{
    VideoStatus Status { get; set; }
    TimeSpan Duration { get; set; }
}

```

This interface makes it possible for a class external to `Video` to set the `Status` property by referencing the `IVideoController` interface. The property can be set from other classes and the handler, but it's unlikely to be set inadvertently. Most importantly, the `Status` property can't be set through a data binding.

To assist the handler implementations in keeping the `Status` property updated, the `Video` class defines an `UpdateStatus` event and command:

```

using System.ComponentModel;

namespace VideoDemos.Controls
{
    public class Video : View, IVideoController
    {
        ...
        public event EventHandler UpdateStatus;

        IDispatcherTimer _timer;

        public Video()
        {
            _timer = Dispatcher.CreateTimer();
            _timer.Interval = TimeSpan.FromMilliseconds(100);
            _timer.Tick += OnTimerTick;
            _timer.Start();
        }

        ~Video() => _timer.Tick -= OnTimerTick;

        void OnTimerTick(object sender, EventArgs e)
        {
            UpdateStatus?.Invoke(this, EventArgs.Empty);
            Handler?.Invoke(nameof(Video.UpdateStatus));
        }
        ...
    }
}

```

The `OnTimerTick` event handler is executed every tenth of a second, which raises the `UpdateStatus` event and invokes the `UpdateStatus` command.

When the `UpdateStatus` command is sent from the `Video` control to its handler, the handler's command mapper ensures that the `MapUpdateStatus` method is invoked:

```

public static void MapUpdateStatus(VideoHandler handler, Video video, object? args)
{
    handler.PlatformView?.UpdateStatus();
}

```

The `MapUpdateStatus` method in turns calls the `UpdateStatus` method on the handler's `PlatformView` property. The `PlatformView` property, which is of type `MauiVideoPlayer`, encapsulates the native views that provide the video player implementation on each platform.

Android

The following code example shows the `UpdateStatus` method on Android sets the `Status` property:

```

using Android.Content;
using Android.Views;
using Android.Widget;
using AndroidX.CoordinatorLayout.Widget;
using VideoDemos.Controls;
using Color = Android.Graphics.Color;
using Uri = Android.Net.Uri;

namespace VideoDemos.Platforms.Android
{
    public class MauiVideoPlayer : CoordinatorLayout
    {
        VideoView _videoView;
        bool _isPrepared;
        Video _video;
        ...

        public MauiVideoPlayer(Context context, Video video) : base(context)
        {
            _video = video;
            ...
            _videoView.Prepared += OnVideoViewPrepared;
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                _videoView.Prepared -= OnVideoViewPrepared;
                ...
            }

            base.Dispose(disposing);
        }

        void OnVideoViewPrepared(object sender, EventArgs args)
        {
            _isPrepared = true;
            ((IVideoController)_video).Duration = TimeSpan.FromMilliseconds(_videoView.Duration);
        }

        public void UpdateStatus()
        {
            VideoStatus status = VideoStatus.NotReady;

            if (_isPrepared)
                status = _videoView.isPlaying ? VideoStatus.Playing : VideoStatus.Paused;

            ((IVideoController)_video).Status = status;
            ...
        }
        ...
    }
}

```

The `videoView.isPlaying` property is a Boolean that indicates if the video is playing or paused. To determine if the `videoView` can't play or pause the video, its `Prepared` event must be handled. This event is raised when the media source is ready for playback. The event is subscribed to in the `MauiVideoPlayer` constructor, and unsubscribed from in its `Dispose` override. The `UpdateStatus` method then uses the `_isPrepared` field and the `VideoView.isPlaying` property to set the `Status` property on the `video` object by casting it to `IVideoController`.

iOS and Mac Catalyst

The following code example shows the `UpdateStatus` method on iOS and Mac Catalyst sets the `Status`

property:

```
using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using System.Diagnostics;
using UIKit;
using VideoDemos.Controls;

namespace VideoDemos.Platforms.MaciOS
{
    public class MauiVideoPlayer : UIView
    {
        AVPlayer _player;
        Video _video;
        ...

        public void UpdateStatus()
        {
            VideoStatus videoStatus = VideoStatus.NotReady;

            switch (_player.Status)
            {
                case AVPlayerStatus.ReadyToPlay:
                    switch (_player.TimeControlStatus)
                    {
                        case AVPlayerTimeControlStatus.Playing:
                            videoStatus = VideoStatus.Playing;
                            break;

                        case AVPlayerTimeControlStatus.Paused:
                            videoStatus = VideoStatus.Paused;
                            break;
                    }
                    break;
            }

            ((IVideoController)_video).Status = videoStatus;
            ...
        }
        ...
    }
}
```

Two properties of `AVPlayer` must be accessed to set the `Status` property - the `Status` property of type `AVPlayerStatus` and the `TimeControlStatus` property of type `AVPlayerTimeControlStatus`. The `Status` property can then be set on the `Video` object by casting it to `IVideoController`.

Positioning bar

The transport controls implemented by each platform include a positioning bar. This bar resembles a slider or scroll bar, and shows the current location of the video within its total duration. Users can manipulate the positioning bar to move forwards or backwards to a new position in the video.

Implementing your own positioning bar requires the `Video` class to know the duration of the video, and its current position within that duration.

Duration

One item of information that the `Video` control needs to support a custom positioning bar is the duration of the video. The `Video` class defines a read-only bindable property named `Duration`, of type `TimeSpan`. This property is defined as read-only because it should only be set from the control's handler:

```

namespace VideoDemos.Controls
{
    public class Video : View, IVideoController
    {
        ...
        private static readonly BindablePropertyKey DurationPropertyKey =
            BindableProperty.CreateReadOnly(nameof(Duration), typeof(TimeSpan), typeof(Video), new
TimeSpan(),
                propertyChanged: (bindable, oldValue, newValue) => ((Video)bindable).SetTimeToEnd());

        public static readonly BindableProperty DurationProperty = DurationPropertyKey.BindableProperty;

        public TimeSpan Duration
        {
            get { return (TimeSpan)GetValue(DurationProperty); }
        }

        TimeSpan IVideoController.Duration
        {
            get { return Duration; }
            set { SetValue(DurationPropertyKey, value); }
        }
        ...
    }
}

```

Usually, a read-only bindable property would have a private `set` accessor on the `Duration` property to allow it to be set from within the class. However, for a `View` derivative supported by handlers, the property must be set from outside the class but only by the control's handler.

NOTE

The property-changed event handler for the `Duration` bindable property calls a method named `SetTimeToEnd`, which is described in [Calculating time to end](#).

For this reason, another property is defined with the name `IVideoController.Duration`. This is an explicit interface implementation, and is made possible by the `IVideoController` interface that the `Video` class implements:

```

public interface IVideoController
{
    VideoStatus Status { get; set; }
    TimeSpan Duration { get; set; }
}

```

This interface makes it possible for a class external to `Video` to set the `Duration` property by referencing the `IVideoController` interface. The property can be set from other classes and the handler, but it's unlikely to be set inadvertently. Most importantly, the `Duration` property can't be set through a data binding.

The duration of a video isn't available immediately after the `Source` property of the `Video` control is set. The video must be partially downloaded before the native view can determine its duration.

Android

On Android, the `VideoView.Duration` property reports a valid duration in milliseconds after the `VideoView.Prepared` event has been raised. The `MauiVideoPlayer` class uses the `Prepared` event handler to obtain the `Duration` property value:

```

using Android.Content;
using Android.Views;
using Android.Widget;
using AndroidX.CoordinatorLayout.Widget;
using VideoDemos.Controls;
using Color = Android.Graphics.Color;
using Uri = Android.Net.Uri;

namespace VideoDemos.Platforms.Android
{
    public class MauiVideoPlayer : CoordinatorLayout
    {
        VideoView _videoView;
        Video _video;
        ...

        void OnVideoViewPrepared(object sender, EventArgs args)
        {
            ...
            ((IVideoController)_video).Duration = TimeSpan.FromMilliseconds(_videoView.Duration);
        }
        ...
    }
}

```

iOS and Mac Catalyst

On iOS and Mac Catalyst, the duration of a video is obtained from the `AVPlayerItem.Duration` property, but not immediately after the `AVPlayerItem` is created. It's possible to set an iOS observer for the `Duration` property, but the `MauiVideoPlayer` class obtains the duration in the `UpdateStatus` method that's called 10 times a second:

```

using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using System.Diagnostics;
using UIKit;
using VideoDemos.Controls;

namespace VideoDemos.Platforms.MaciOS
{
    public class MauiVideoPlayer : UIView
    {
        AVPlayerItem _playerItem;
        ...

        TimeSpan ConvertTime(CMTime cmTime)
        {
            return TimeSpan.FromSeconds(Double.IsNaN(cmTime.Seconds) ? 0 : cmTime.Seconds);
        }

        public void UpdateStatus()
        {
            ...
            if (_playerItem != null)
            {
                ((IVideoController)_video).Duration = ConvertTime(_playerItem.Duration);
                ...
            }
        }
        ...
    }
}

```

The `convertTime` method converts a `CMTime` object to a `TimeSpan` value.

Position

The `Video` control also needs a `Position` property that increases from zero to `Duration` as the video plays. The `Video` class implements this property as a bindable property with public `get` and `set` accessors:

```
namespace VideoDemos.Controls
{
    public class Video : View, IVideoController
    {
        ...
        public static readonly BindableProperty PositionProperty =
            BindableProperty.Create(nameof(Position), typeof(TimeSpan), typeof(Video), new TimeSpan(),
                propertyChanged: (bindable, oldValue, newValue) => ((Video)bindable).SetTimeToEnd());

        public TimeSpan Position
        {
            get { return (TimeSpan)GetValue(PositionProperty); }
            set { SetValue(PositionProperty, value); }
        }
        ...
    }
}
```

The `get` accessor returns the current position of the video as its playing. The `set` accessor responds to user manipulation of the positioning bar by moving the video position forwards or backwards.

NOTE

The property-changed event handler for the `Position` bindable property calls a method named `SetTimeToEnd`, which is described in [Calculating time to end](#).

On Android, iOS and Mac Catalyst, the property that obtains the current position only has a `get` accessor. Instead, a `Seek` method is available to set the position. This seems to be a more sensible approach than using a single `Position` property, which has an inherent problem. As a video plays, a `Position` property must be continually updated to reflect the new position. But you don't want most changes of the `Position` property to cause the video player to move to a new position in the video. If that happens, the video player would respond by seeking to the last value of the `Position` property, and the video wouldn't advance.

Despite the difficulties of implementing a `Position` property with `get` and `set` accessors, this approach is used because it can utilize data binding. The `Position` property of the `Video` control can be bound to a `Slider` that's used both to display the position and to seek a new position. However, several precautions are necessary when implementing the `Position` property, to avoid feedback loops.

Android

On Android, the `VideoView.CurrentPosition` property indicates the current position of the video. The `MauiVideoPlayer` class sets the `Position` property in the `UpdateStatus` method at the same time as it sets the `Duration` property:

```

using Android.Content;
using Android.Views;
using Android.Widget;
using AndroidX.CoordinatorLayout.Widget;
using VideoDemos.Controls;
using Color = Android.Graphics.Color;
using Uri = Android.Net.Uri;

namespace VideoDemos.Platforms.Android
{
    public class MauiVideoPlayer : CoordinatorLayout
    {
        VideoView _videoView;
        Video _video;
        ...

        public void UpdateStatus()
        {
            ...
            TimeSpan timeSpan = TimeSpan.FromMilliseconds(_videoView.CurrentPosition);
            _video.Position = timeSpan;
        }

        public void UpdatePosition()
        {
            if (Math.Abs(_videoView.CurrentPosition - _video.Position.TotalMilliseconds) > 1000)
            {
                _videoView.SeekTo((int)_video.Position.TotalMilliseconds);
            }
        }
        ...
    }
}

```

Every time the `Position` property is set by the `UpdateStatus` method, the `Position` property fires a `PropertyChanged` event, which causes the property mapper for the handler to call the `updatePosition` method. The `updatePosition` method should do nothing for most of the property changes. Otherwise, with every change in the video's position it would be moved to same position it just reached. To avoid this feedback loop, the `UpdatePosition` only calls the `Seek` method on the `VideoView` object when the difference between the `Position` property and the current position of the `VideoView` is greater than one second.

iOS and Mac Catalyst

On iOS and Mac Catalyst, the `AVPlayerItem.CurrentTime` property indicates the current position of the video. The `MauiVideoPlayer` class sets the `Position` property in the `UpdateStatus` method at the same time as it sets the `Duration` property:

```

using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using System.Diagnostics;
using UIKit;
using VideoDemos.Controls;

namespace VideoDemos.Platforms.MacOS
{
    public class MauiVideoPlayer : UIView
    {
        AVPlayer _player;
        AVPlayerItem _playerItem;
        Video _video;
        ...

        TimeSpan ConvertTime(CMTime cmTime)
        {
            return TimeSpan.FromSeconds(Double.IsNaN(cmTime.Seconds) ? 0 : cmTime.Seconds);
        }

        public void UpdateStatus()
        {
            ...
            if (_playerItem != null)
            {
                ...
                _video.Position = ConvertTime(_playerItem.CurrentTime);
            }
        }

        public void UpdatePosition()
        {
            TimeSpan controlPosition = ConvertTime(_player.CurrentTime);
            if (Math.Abs((controlPosition - _video.Position).TotalSeconds) > 1)
            {
                _player.Seek(CMTime.FromSeconds(_video.Position.TotalSeconds, 1));
            }
        }
        ...
    }
}

```

Every time the `Position` property is set by the `UpdateStatus` method, the `Position` property fires a `PropertyChanged` event, which causes the property mapper for the handler to call the `updatePosition` method. The `updatePosition` method should do nothing for most of the property changes. Otherwise, with every change in the video's position it would be moved to same position it just reached. To avoid this feedback loop, the `UpdatePosition` only calls the `Seek` method on the `AVPlayer` object when the difference between the `Position` property and the current position of the `AVPlayer` is greater than one second.

Calculating time to end

Sometimes video players show the time remaining in the video. This value begins at the video's duration when the video begins, and decreases down to zero when the video ends.

The `Video` class includes a read-only `TimeToEnd` property that's calculated based on changes to the `Duration` and `Position` properties:

```
namespace VideoDemos.Controls
{
    public class Video : View, IVideoController
    {
        ...
        private static readonly BindablePropertyKey TimeToEndPropertyKey =
            BindableProperty.CreateReadOnly(nameof(TimeToEnd), typeof(TimeSpan), typeof(Video), new
TimeSpan());
        public static readonly BindableProperty TimeToEndProperty = TimeToEndPropertyKey.BindableProperty;

        public TimeSpan TimeToEnd
        {
            get { return (TimeSpan)GetValue(TimeToEndProperty); }
            private set { SetValue(TimeToEndPropertyKey, value); }
        }

        void SetTimeToEnd()
        {
            TimeToEnd = Duration - Position;
        }
        ...
    }
}
```

The `SetTimeToEnd` method is called from the property-changed event handlers of the `Duration` and `Position` properties.

Custom positioning bar

A custom positioning bar can be implemented by creating a class that derives from `Slider`, which contains `Duration` and `Position` properties of type `TimeSpan`:

```

namespace VideoDemos.Controls
{
    public class PositionSlider : Slider
    {
        public static readonly BindableProperty DurationProperty =
            BindableProperty.Create(nameof(Duration), typeof(TimeSpan), typeof(PositionSlider), new
TimeSpan(1),
            propertyChanged: (bindable, oldValue, newValue) =>
            {
                double seconds = ((TimeSpan)newValue).TotalSeconds;
                ((Slider)bindable).Maximum = seconds <= 0 ? 1 : seconds;
            });

        public static readonly BindableProperty PositionProperty =
            BindableProperty.Create(nameof(Position), typeof(TimeSpan), typeof(PositionSlider), new
TimeSpan(0),
            defaultBindingMode: BindingMode.TwoWay,
            propertyChanged: (bindable, oldValue, newValue) =>
            {
                double seconds = ((TimeSpan)newValue).TotalSeconds;
                ((Slider)bindable).Value = seconds;
            });

        public TimeSpan Duration
        {
            get { return (TimeSpan)GetValue(DurationProperty); }
            set { SetValue(DurationProperty, value); }
        }

        public TimeSpan Position
        {
            get { return (TimeSpan)GetValue(PositionProperty); }
            set { SetValue(PositionProperty, value); }
        }

        public PositionSlider()
        {
            PropertyChanged += (sender, args) =>
            {
                if (args.PropertyName == "Value")
                {
                    TimeSpan newPosition = TimeSpan.FromSeconds(Value);
                    if (Math.Abs(newPosition.TotalSeconds - Position.TotalSeconds) / Duration.TotalSeconds >
0.01)
                        Position = newPosition;
                }
            };
        }
    }
}

```

The property-changed event handler for the `Duration` property sets the `Maximum` property of the `Slider` to the `TotalSeconds` property of the `TimeSpan` value. Similarly, the property-changed event handler for the `Position` property sets the `Value` property of the `Slider`. This is the mechanism by which the `Slider` tracks the position of `PositionSlider`.

The `PositionSlider` is updated from the underlying `Slider` in only one scenario, which is when the user manipulates the `Slider` to indicate that the video should be advanced or reversed to a new position. This is detected in the `PropertyChanged` handler in the `PositionSlider` constructor. This event handler checks for a change in the `Value` property, and if it's different from the `Position` property, then the `Position` property is set from the `Value` property.

Register the handler

A custom control and its handler must be registered with an app, before it can be consumed. This should occur in the `CreateMauiApp` method in the `MauiProgram` class in your app project, which is the cross-platform entry point for the app:

```
using VideoDemos.Controls;
using VideoDemos.Handlers;

namespace VideoDemos;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts =>
        {
            fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
        })
            .ConfigureMauiHandlers(handlers =>
        {
            handlers.AddHandler(typeof(Video), typeof(VideoHandler));
        });

        return builder.Build();
    }
}
```

The handler is registered with the `ConfigureMauiHandlers` and `AddHandler` method. The first argument to the `AddHandler` method is the cross-platform control type, with the second argument being its handler type.

Consume the cross-platform control

After registering the handler with your app, the cross-platform control can be consumed.

Play a web video

The `video` control can play a video from a URL, as shown in the following example:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="clr-namespace:VideoDemos.Controls"
    x:Class="VideoDemos.Views.PlayWebVideoPage"
    Unloaded="OnContentPageUnloaded"
    Title="Play web video">
    <controls:Video x:Name="video"
        Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4" />
</ContentPage>
```

In this example, the `VideoSourceConverter` class converts the string that represents the URI to a `UriVideoSource`. The video then begins loading and starts playing once a sufficient quantity of data has been downloaded and buffered. On each platform, the transport controls fade out if they're not used but can be restored by tapping on the video.

Play a video resource

Video files that are embedded in the `Resources\Raw` folder of the app, with a `MauiAsset` build action, can be

played by the `Video` control:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="clr-namespace:VideoDemos.Controls"
    x:Class="VideoDemos.Views.PlayVideoResourcePage"
    Unloaded="OnContentPageUnloaded"
    Title="Play video resource">
    <controls:Video x:Name="video"
        Source="video.mp4" />
</ContentPage>
```

In this example, the `VideoSourceConverter` class converts the string that represents the filename of the video to a `ResourceVideoSource`. For each platform, the video begins playing almost immediately after the video source is set because the file is in the app package and doesn't need to be downloaded. On each platform, the transport controls fade out if they're not used but can be restored by tapping on the video.

Play a video file from the device's library

Video files that are stored on the device can be retrieved and then played by the `Video` control:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="clr-namespace:VideoDemos.Controls"
    x:Class="VideoDemos.Views.PlayLibraryVideoPage"
    Unloaded="OnContentPageUnloaded"
    Title="Play library video">
    <Grid RowDefinitions="*,Auto">
        <controls:Video x:Name="video" />
        <Button Grid.Row="1"
            Text="Show Video Library"
            Margin="10"
            HorizontalOptions="Center"
            Clicked="OnShowVideoLibraryClicked" />
    </Grid>
</ContentPage>
```

When the `Button` is tapped its `Clicked` event handler is executed, which is shown in the following code example:

```
async void OnShowVideoLibraryClicked(object sender, EventArgs e)
{
    Button button = sender as Button;
    button.IsEnabled = false;

    var pickedVideo = await MediaPicker.PickVideoAsync();
    if (!string.IsNullOrWhiteSpace(pickedVideo?.FileName))
    {
        video.Source = new FileVideoSource
        {
            File = pickedVideo.FullPath
        };
    }

    button.IsEnabled = true;
}
```

The `Clicked` event handler uses .NET MAUI's `MediaPicker` class to let the user pick a video file from the device. The picked video file is then encapsulated as a `FileVideoSource` object and set as the `Source` property of the `Video` control. For more information about the `MediaPicker` class, see [Media picker](#). For each platform, the video begins playing almost immediately after the video source is set because the file is on the device and

doesn't need to be downloaded. On each platform, the transport controls fade out if they're not used but can be restored by tapping on the video.

Configure the Video control

You can prevent a video from automatically starting by setting the `AutoPlay` property to `false`:

```
<controls:Video x:Name="video"
    Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4"
    AutoPlay="False" />
```

Similarly, you can suppress the transport controls by setting the `AreTransportControlsEnabled` property to `false`:

```
<controls:Video x:Name="video"
    Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4"
    AreTransportControlsEnabled="False" />
```

If you set `AutoPlay` and `AreTransportControlsEnabled` to `false`, the video won't begin playing and there will be no way to start it playing. In this scenario you'd need to call the `Play` method from the code-behind file, or create your own transport controls.

Use custom transport controls

The following XAML example shows custom transport controls that play, pause, and stop the video:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="clr-namespace:VideoDemos.Controls"
    x:Class="VideoDemos.Views.CustomTransportPage"
    Unloaded="OnContentPageUnloaded"
    Title="Custom transport controls">
    <Grid RowDefinitions="*,Auto">
        <controls:Video x:Name="video"
            AutoPlay="False"
            AreTransportControlsEnabled="False"
            Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4" />
        <ActivityIndicator Color="Gray"
            IsVisible="False">
            <ActivityIndicator.Triggers>
                <DataTrigger TargetType="ActivityIndicator"
                    Binding="{Binding Source={x:Reference video},
                        Path=Status}"
                    Value="{x:Static controls:VideoStatus.NotReady}">
                    <Setter Property="IsVisible"
                        Value="True" />
                    <Setter Property="IsRunning"
                        Value="True" />
                </DataTrigger>
            </ActivityIndicator.Triggers>
        </ActivityIndicator>
        <Grid Grid.Row="1"
            Margin="0,10"
            ColumnDefinitions="0.5*,0.5*"
            BindingContext="{x:Reference video}">
            <Button Text="ꖶ️ Play"
                HorizontalOptions="Center"
                Clicked="OnPlayPauseButtonClicked">
                <Button.Triggers>
                    <DataTrigger TargetType="Button"
                        Binding="{Binding Status}"
                        Value="{x:Static controls:VideoStatus.Playing}">
                        <Setter Property="Text"
                            Value="⏸ Pause" />
                    </DataTrigger>
                    <DataTrigger TargetType="Button"
                        Binding="{Binding Status}"
                        Value="{x:Static controls:VideoStatus.NotReady}">
                        <Setter Property="isEnabled"
                            Value="False" />
                    </DataTrigger>
                </Button.Triggers>
            </Button>
            <Button Grid.Column="1"
                Text="⏹ Stop"
                HorizontalOptions="Center"
                Clicked="OnStopButtonClicked">
                <Button.Triggers>
                    <DataTrigger TargetType="Button"
                        Binding="{Binding Status}"
                        Value="{x:Static controls:VideoStatus.NotReady}">
                        <Setter Property="isEnabled"
                            Value="False" />
                    </DataTrigger>
                </Button.Triggers>
            </Button>
        </Grid>
    </Grid>
</ContentPage>

```

In this example, the `Video` control sets the `AreTransportControlsEnabled` property to `false` and defines a `Button` that plays and pauses the video, and a `Button` that stop video playback. Button appearance is defined

using unicode characters and their text equivalents, to create buttons that consist of an icon and text:



When the video is playing, the play button is updated to a pause button:



The UI also includes an `ActivityIndicator` that's displayed while the video is loading. Data triggers are used to enable and disable the `ActivityIndicator` and the buttons, and to switch the first button between play and pause. For more information about data triggers, see [Data triggers](#).

The code-behind file defines the event handlers for the button `Clicked` events:

```
public partial class CustomTransportPage : ContentPage
{
    ...
    void OnPlayPauseButtonClicked(object sender, EventArgs args)
    {
        if (video.Status == VideoStatus.Playing)
        {
            video.Pause();
        }
        else if (video.Status == VideoStatus.Paused)
        {
            video.Play();
        }
    }

    void OnStopButtonClicked(object sender, EventArgs args)
    {
        video.Stop();
    }
    ...
}
```

Custom positioning bar

The following example shows a custom positioning bar, `PositionSlider`, being consumed in XAML:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="clr-namespace:VideoDemos.Controls"
    x:Class="VideoDemos.Views.CustomPositionBarPage"
    Unloaded="OnContentPageUnloaded"
    Title="Custom position bar">
    <Grid RowDefinitions="*,Auto,Auto">
        <controls:Video x:Name="video"
            AreTransportControlsEnabled="False"
            Source="{StaticResource ElephantsDream}" />
        ...
        <Grid Grid.Row="1"
            Margin="10,0"
            ColumnDefinitions="0.25*,0.25*,0.25*,0.25*"
            BindingContext="{x:Reference video}">
            <Label Text="{Binding Path=Position,
                StringFormat='0:hh\\:mm\\:ss'}"
                HorizontalOptions="Center"
                VerticalOptions="Center" />
        ...
        <Label Grid.Column="3"
            Text="{Binding Path=TimeToEnd,
                StringFormat='0:hh\\:mm\\:ss'}"
                HorizontalOptions="Center"
                VerticalOptions="Center" />
    </Grid>
    <controls:PositionSlider Grid.Row="2"
        Margin="10,0,10,10"
        BindingContext="{x:Reference video}"
        Duration="{Binding Duration}"
        Position="{Binding Position}">
        <controls:PositionSlider.Triggers>
            <DataTrigger TargetType="controls:PositionSlider"
                Binding="{Binding Status}"
                Value="{x:Static controls:VideoStatus.NotReady}">
                <Setter Property="IsEnabled"
                    Value="False" />
            </DataTrigger>
        </controls:PositionSlider.Triggers>
    </controls:PositionSlider>
</Grid>
</ContentPage>

```

The `Position` property of the `Video` object is bound to the `Position` property of the `PositionSlider`, without performance issues, because the `Video.Position` property is changed by the `MauiVideoPlayer.UpdateStatus` method on each platform, which is only called 10 times a second. In addition, two `Label` objects display the `Position` and `TimeToEnd` properties values from the `video` object.

Native view cleanup

Each platform's handler implementation overrides the `DisconnectHandler` implementation, which is used to perform native view cleanup such as unsubscribing from events and disposing objects. However, this override is intentionally not invoked by .NET MAUI. Instead, you must invoke it yourself from a suitable location in your app's lifecycle. This will often be when the page containing the `Video` control is navigated away from, which causes the page's `Unloaded` event to be raised.

An event handler for the page's `Unloaded` event can be registered in XAML:

```
<ContentPage ...>
    xmlns:controls="clr-namespace:VideoDemos.Controls"
    Unloaded="OnContentPageUnloaded">
        <controls:Video x:Name="video"
            ...
        />
</ContentPage>
```

The event handler for the `Unloaded` event can then invoke the `DisconnectHandler` method on its `Handler` instance:

```
void OnContentPageUnloaded(object sender, EventArgs e)
{
    video.Handler?.DisconnectHandler();
}
```

In addition to cleaning up native view resources, invoking the handler's `DisconnectHandler` method also ensures that videos stop playing on backwards navigation on iOS.

Customize controls with handlers

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

Handlers can be customized to augment the appearance and behavior of a cross-platform control beyond the customization that's possible through the control's API. This customization, which modifies the native views for the cross-platform control, is achieved by modifying the mapper for a handler with one of the following methods:

- `PrependToMapping`, which modifies the mapper for a handler before the .NET MAUI control mappings have been applied.
- `ModifyMapping`, which modifies an existing mapping.
- `AppendToMapping`, which modifies the mapper for a handler after the .NET MAUI control mappings have been applied.

Each of these methods has an identical signature that requires two arguments:

- A `string`-based key. When modifying one of the mappings provided by .NET MAUI, the key used by .NET MAUI must be specified. The key values used by .NET MAUI control mappings are based on interface and property names, for example `nameof(IEntry.IsPassword)`. The interfaces, and their properties, that abstract each cross-platform control can be found [here](#). Otherwise, this key can be an arbitrary value that doesn't have to correspond to the name of a property exposed by a type. For example, `MyCustomization` can be specified as a key, with any native view modification being performed as the customization.
- An `Action` that represents the method that performs the handler customization. The `Action` specifies two arguments:
 - A `handler` argument that provides an instance of the handler being customized.
 - A `view` argument that provides an instance of the cross-platform control that the handler implements.

IMPORTANT

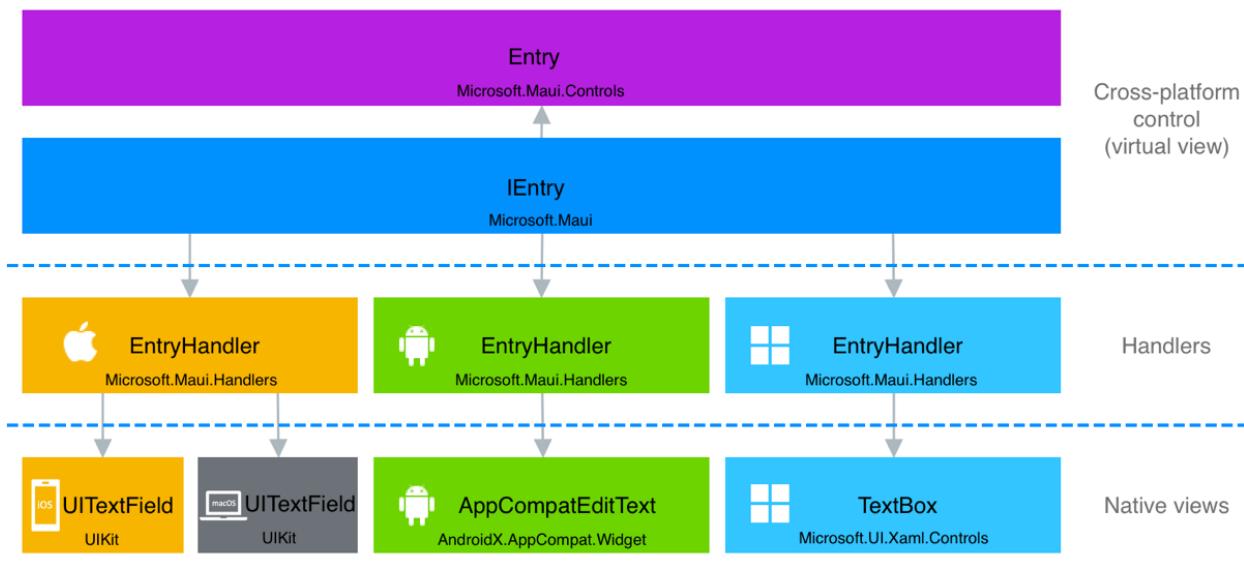
Handler customizations are global and aren't scoped to a specific control instance. Handler customization is allowed to happen anywhere in your app. Once a handler is customized, it affects all controls of that type, everywhere in your app.

Each handler class exposes the native view for the cross-platform control via its `PlatformView` property. This property can be accessed to set native view properties, invoke native view methods, and subscribe to native view events. In addition, the cross-platform control implemented by the handler is exposed via its `VirtualView` property.

Handlers can be customized per platform by using conditional compilation, to multi-target code based on the platform. Alternatively, you can use partial classes to organize your code into platform-specific folders and files. For more information about conditional compilation, see [Conditional compilation](#).

Customize a control

The .NET MAUI `Entry` is a single-line text input control, that implements the `IEntry` interface. On iOS, the `EntryHandler` maps the `Entry` to an iOS `UITextField`. On Android, the `Entry` is mapped to an `AppCompatEditText`, and on Windows the `Entry` is mapped to a `TextBox`:



The `Entry` property mapper, in the `EntryHandler` class, maps the cross-platform control properties to the native view API. This ensures that when a property is set on an `Entry`, the underlying native view is updated as required.

The property mapper can be modified to customize `Entry` on each platform:

```

namespace CustomizeHandlersDemo.Views;

public partial class CustomizeEntryPage : ContentPage
{
    public CustomizeEntryPage()
    {
        InitializeComponent();
        ModifyEntry();
    }

    void ModifyEntry()
    {
        Microsoft.Maui.Handlers.EntryHandler.Mapper.AppendToMapping("MyCustomization", (handler, view) =>
        {
#if ANDROID
            handler.PlatformView.SetselectAllOnFocus(true);
#endif
#if IOS || MACCATALYST
            handler.PlatformView.EditingDidBegin += (s, e) =>
            {
                handler.PlatformView.PerformSelector(new ObjCRuntime.Selector("selectAll"), null, 0.0f);
            };
#endif
#if WINDOWS
            handler.PlatformView.GotFocus += (s, e) =>
            {
                handler.PlatformView.SelectAll();
            };
#endif
        });
    }
}
  
```

In this example, the `Entry` customization occurs in a page class. Therefore, all `Entry` controls on Android, iOS, and Windows will be customized once an instance of the `CustomizeEntryPage` is created. Customization is performed by accessing the handlers `PlatformView` property, which provides access to the native view that implements the cross-platform control on each platform. Native code then customizes the handler by selecting all of the text in the `Entry` when it gains focus.

For more information about mappers, see [Mappers](#).

Customize a specific control instance

Handlers are global, and customizing a handler for a control will result in all controls of the same type being customized in your app. However, handlers for specific control instances can be customized by subclassing the control, and then by modifying the handler for the base control type only when the control is of the subclassed type. For example, to customize a specific `Entry` control on a page that contains multiple `Entry` controls, you should first subclass the `Entry` control:

```
namespace CustomizeHandlersDemo.Controls
{
    internal class MyEntry : Entry
    {
    }
}
```

You can then customize the `EntryHandler`, via its property mapper, to perform the desired modification only to `MyEntry` instances:

```
Microsoft.Maui.Handlers.EntryHandler.Mapper.AppendToMapping("MyCustomization", (handler, view) =>
{
    if (view is MyEntry)
    {
        #if ANDROID
            handler.PlatformView.SetSelectAllOnFocus(true);
        #elif IOS || MACCATALYST
            handler.PlatformView.EditingDidBegin += (s, e) =>
            {
                handler.PlatformView.PerformSelector(new ObjCRuntime.Selector("selectAll"), null, 0.0f);
            };
        #elif WINDOWS
            handler.PlatformView.GotFocus += (s, e) =>
            {
                handler.PlatformView.SelectAll();
            };
        #endif
    }
});
```

If the handler customization is performed in your `App` class, any `MyEntry` instances in the app will be customized as per the handler modification.

Customize a control using the handler lifecycle

All handler-based .NET MAUI controls support `HandlerChanging` and `HandlerChanged` events. The `HandlerChanged` event is raised when the native view that implements the cross-platform control is available and initialized. The `HandlerChanging` event is raised when the control's handler is about to be removed from the cross-platform control. For more information about handler lifecycle events, see [Handler lifecycle](#).

The handler lifecycle can be used to perform handler customization. For example, to subscribe to, and unsubscribe from, native view events you must register event handlers for the `HandlerChanged` and `HandlerChanging` events on the cross-platform control being customized:

```
<Entry HandlerChanged="OnEntryHandlerChanged"
       HandlerChanging="OnEntryHandlerChanging" />
```

Handlers can be customized per platform by using conditional compilation, or by using partial classes to organize your code into platform-specific folders and files. Each approach will be discussed in turn, by customizing an `Entry` so that all of its text is selected when it gains focus.

Conditional compilation

The code-behind file containing the event handlers for the `HandlerChanged` and `HandlerChanging` events is shown in the following example, which uses conditional compilation:

```
#if ANDROID
using AndroidX.AppCompat.Widget;
#elif IOS || MACCATALYST
using UIKit;
#elif WINDOWS
using Microsoft.UI.Xaml.Controls;
using Microsoft.UI.Xaml;
#endif

namespace CustomizeHandlersDemo.Views;

public partial class CustomizeEntryHandlerLifecyclePage : ContentPage
{
    public CustomizeEntryHandlerLifecyclePage()
    {
        InitializeComponent();
    }

    void OnEntryHandlerChanged(object sender, EventArgs e)
    {
        Entry entry = sender as Entry;
#if ANDROID
        (entry.Handler.PlatformView as AppCompatEditText).SetSelectAllOnFocus(true);
#elif IOS || MACCATALYST
        (entry.Handler.PlatformView as UITextField).EditingDidBegin += OnEditingDidBegin;
#elif WINDOWS
        (entry.Handler.PlatformView as TextBox).GotFocus += OnGotFocus;
#endif
    }

    void OnEntryHandlerChanging(object sender, HandlerChangingEventArgs e)
    {
        if (e.OldHandler != null)
        {
#if IOS || MACCATALYST
            (e.OldHandler.PlatformView as UITextField).EditingDidBegin -= OnEditingDidBegin;
#elif WINDOWS
            (e.OldHandler.PlatformView as TextBox).GotFocus -= OnGotFocus;
#endif
        }
    }

#if IOS || MACCATALYST
    void OnEditingDidBegin(object sender, EventArgs e)
    {
        var nativeView = sender as UITextField;
        nativeView.PerformSelector(new ObjCRuntime.Selector("selectAll"), null, 0.0f);
    }
#elif WINDOWS
    void OnGotFocus(object sender, RoutedEventArgs e)
    {
        var nativeView = sender as TextBox;
        nativeView.SelectAll();
    }
#endif
}
```

The `HandlerChanged` event is raised after the native view that implements the cross-platform control has been created and initialized. Therefore, its event handler is where native event subscriptions should be performed. This requires casting the `PlatformView` property of the handler to the type, or base type, of the native view so that native events can be accessed. In this example, on iOS, Mac Catalyst, and Windows, the `OnHandlerChanged` event subscribes to native view events that are raised when the native views that implement the `Entry` gain focus.

The `onEditingDidBegin` and `OnGotFocus` event handlers access the native view for the `Entry` on their respective platforms, and select all text that's in the `Entry`.

The `HandlerChanging` event is raised before the existing handler is removed from the cross-platform control, and before the new handler for the cross-platform control is created. Therefore, its event handler is where native event subscriptions should be removed, and other cleanup should be performed. The `HandlerChangingEventArgs` object that accompanies this event has `OldHandler` and `NewHandler` properties, which will be set to the old and new handlers respectively. In this example, the `OnHandlerChanging` event removes the subscription to the native view events on iOS, Mac Catalyst, and Windows.

Partial classes

Rather than using conditional compilation, it's also possible to use partial classes to organize your control customization code into platform-specific folders and files. With this approach, your customization code is separated into a cross-platform partial class and a platform-specific partial class. The following example shows the cross-platform partial class:

```
namespace CustomizeHandlersDemo.Views;

public partial class CustomizeEntryPartialMethodsPage : ContentPage
{
    public CustomizeEntryPartialMethodsPage()
    {
        InitializeComponent();
    }

    partial void ChangedHandler(object sender, EventArgs e);
    partial void ChangingHandler(object sender, HandlerChangingEventArgs e);

    void OnEntryHandlerChanged(object sender, EventArgs e) => ChangedHandler(sender, e);
    void OnEntryHandlerChanging(object sender, HandlerChangingEventArgs e) => ChangingHandler(sender, e);
}
```

IMPORTANT

The cross-platform partial class shouldn't be placed in any of the *Platforms* child folders of your project.

In this example, the two event handlers call partial methods named `ChangedHandler` and `ChangingHandler`, whose signatures are defined in the cross-platform partial class. The partial method implementations are then defined in the platform-specific partial classes, which should be placed in the correct *Platforms* child folders to ensure that the build system only attempts to build native code when building for the specific platform. For example, the following code shows the `CustomizeEntryPartialMethodsPage` class in the *Platforms > Windows* folder of the project:

```
using Microsoft.UI.Xaml;
using Microsoft.UI.Xaml.Controls;

namespace CustomizeHandlersDemo.Views
{
    public partial class CustomizeEntryPartialMethodsPage : ContentPage
    {
        partial void ChangedHandler(object sender, EventArgs e)
        {
            Entry entry = sender as Entry;
            (entry.Handler.PlatformView as TextBox).GotFocus += OnGotFocus;
        }

        partial void ChangingHandler(object sender, HandlerChangingEventArgs e)
        {
            if (e.OldHandler != null)
            {
                (e.OldHandler.PlatformView as TextBox).GotFocus -= OnGotFocus;
            }
        }

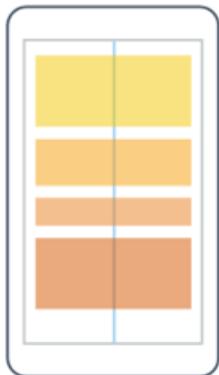
        void OnGotFocus(object sender, RoutedEventArgs e)
        {
            var nativeView = sender as TextBox;
            nativeView.SelectAll();
        }
    }
}
```

The advantage of this approach is that conditional compilation isn't required, and that the partial methods don't have to be implemented on each platform. If an implementation isn't provided on a platform, then the method and all calls to the method are removed at compile time. For information about partial methods, see [Partial methods](#).

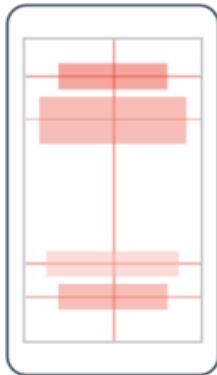
For information about the organization of the *Platforms* folder in a .NET MAUI project, see [Partial classes and methods](#). For information about how to configure multi-targeting so that you don't have to place platform code into sub-folders of the *Platforms* folder, see [Configure multi-targeting](#).

Layouts

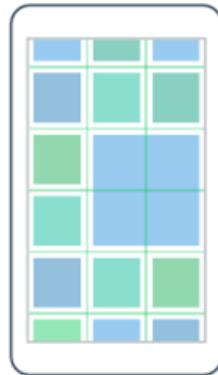
9/20/2022 • 7 minutes to read • [Edit Online](#)



StackLayout



AbsoluteLayout



Grid



FlexLayout

.NET Multi-platform App UI (.NET MAUI) 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.

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://schemas.microsoft.com/dotnet/2021/maui"
    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](#).

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

HorizontalStackLayout

A [HorizontalStackLayout](#) organizes child views in a one-dimensional horizontal stack, and is a more performant alternative to a [StackLayout](#). [HorizontalStackLayout](#) is typically used to arrange a subsection of the UI on a page.

The following XAML shows how to create a [HorizontalStackLayout](#) containing different child views:

```
<HorizontalStackLayout Margin="20">
    <Rectangle Fill="Red"
        HeightRequest="30"
        WidthRequest="30" />
    <Label Text="Red"
        FontSize="18" />
</HorizontalStackLayout>
```

In a [HorizontalStackLayout](#), if an element's size is not explicitly set, it expands to fill the available height.

For more information, see [HorizontalStackLayout](#).

VerticalStackLayout

A [VerticalStackLayout](#) organizes child views in a one-dimensional vertical stack, and is a more performant alternative to a [StackLayout](#). [VerticalStackLayout](#) is typically used to arrange a subsection of the UI on a page.

The following XAML shows how to create a [VerticalStackLayout](#) containing three [Label](#) objects:

```

<VerticalStackLayout Margin="20,35,20,25">
    <Label Text="The VerticalStackLayout has its Margin property set, to control the rendering position of the VerticalStackLayout." />
    <Label Text="The Padding property can be set to specify the distance between the VerticalStackLayout and its children." />
    <Label Text="The Spacing property can be set to specify the distance between views in the VerticalStackLayout." />
</VerticalStackLayout>

```

In a `VerticalStackLayout`, if an element's size is not explicitly set, it expands to fill the available width.

For more information, see [VerticalStackLayout](#).

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.

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 [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 [FlexLayout](#).

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 [AbsoluteLayout](#).

BindableLayout

A `BindableLayout` enables any layout class that derives from the `Layout` 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`.

A bindable layout is populated with data by setting its `ItemsSource` property to any collection that implements `IEnumerable`, and attaching it to a `Layout`-derived class. The appearance of each item in the bindable layout can be defined by setting the `BindableLayout.ItemTemplate` attached property to a `DataTemplate`.

The following XAML shows how to bind a `StackLayout` to a collection of items, and define their appearance with a `DataTemplate`:

```

<StackLayout BindableLayout.ItemsSource="{Binding User.TopFollowers}"
            Orientation="Horizontal">
    <BindableLayout.ItemTemplate>
        <DataTemplate>
            <Image Source="{Binding}"
                  Aspect="AspectFill"
                  WidthRequest="44"
                  HeightRequest="44" />
        </DataTemplate>
    </BindableLayout.ItemTemplate>
</StackLayout>

```

Bindable layouts should only be used when the collection of items to be displayed is small, and scrolling and selection isn't required.

For more information, see [BindableLayout](#).

Input transparency

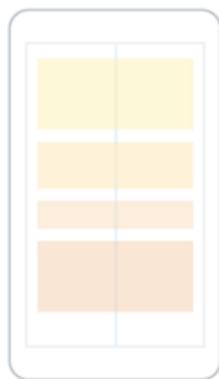
Each visual element has an `InputTransparent` bindable property that's used to define whether the element can receive input. Its default value is `false`, ensuring that the element can receive input. When this property is `true` on an element, the element won't receive any input. Instead, input will be passed to any elements that are visually behind the element.

The `Layout` class, from which all layouts derive, has a `CascadeInputTransparent` bindable property that controls whether child elements inherit the input transparency of the layout. Its default value is `true`, ensuring that setting the `InputTransparent` property to `true` on a layout class will result in all elements within the layout not receiving any input.

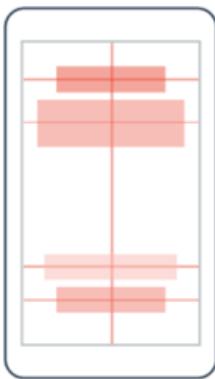
AbsoluteLayout

9/20/2022 • 7 minutes to read • [Edit Online](#)

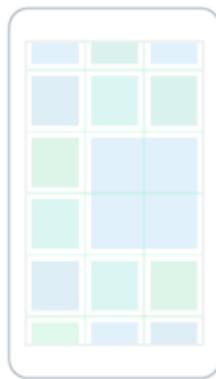
 [Browse the sample](#)



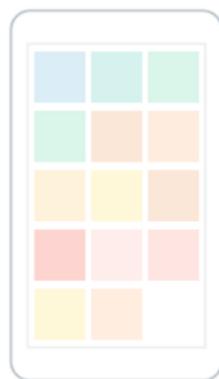
StackLayout



AbsoluteLayout



Grid



FlexLayout

The .NET Multi-platform App UI (.NET MAUI) `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 `Rect`, 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 [.NET MAUI Attached Properties](#).

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 an `AbsoluteLayout` and setting the `AbsoluteLayout.LayoutBounds` 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="AbsoluteLayoutDemos.Views.XAML.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.LayoutBounds` 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.LayoutBounds` 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 StylishHeaderDemoPage : ContentPage
{
    public StylishHeaderDemoPage()
    {
        AbsoluteLayout absoluteLayout = new AbsoluteLayout
        {
            Margin = new Thickness(20)
        };

        absoluteLayout.Add(new BoxView
        {
            Color = Colors.Silver
        }, new Rect(0, 10, 200, 5));
        absoluteLayout.Add(new BoxView
        {
            Color = Colors.Silver
        }, new Rect(0, 20, 200, 5));
        absoluteLayout.Add(new BoxView
        {
            Color = Colors.Silver
        }, new Rect(10, 0, 5, 65));
        absoluteLayout.Add(new BoxView
        {
            Color = Colors.Silver
        }, new Rect(20, 0, 5, 65));

        absoluteLayout.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 `Rect` 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 layout, using the `AbsoluteLayout.SetLayoutBounds` method. The first argument to this method is the child, and the second is a `Rect` 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 `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.

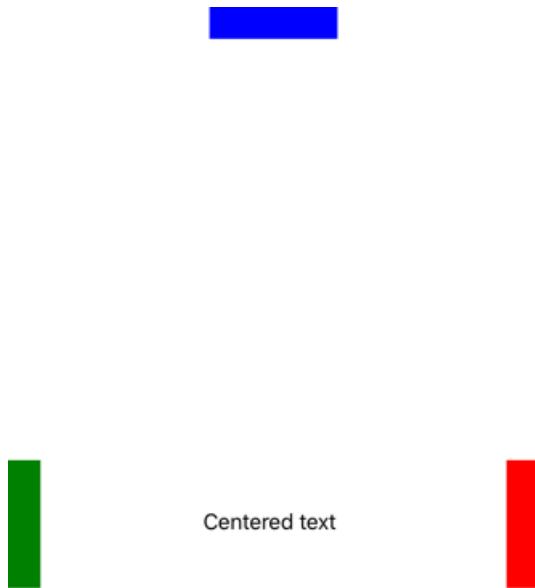
For example, if you use the `SizeProportional` flag and set the width of a child to 0.25 and the height to 0.1, the child will be one-quarter of the width of the `AbsoluteLayout` and one-tenth the height. The `PositionProportional` flag is similar. A position of (0,0) puts the child in the upper-left 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="AbsoluteLayoutDemos.Views.XAML.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 ProportionalDemoPage : ContentPage
{
    public ProportionalDemoPage()
    {
        BoxView blue = new BoxView { Color = Colors.Blue };
        AbsoluteLayout.SetLayoutBounds(blue, new Rect(0.5, 0, 100, 25));
        AbsoluteLayout.SetLayoutFlags(blue, AbsoluteLayoutFlags.PositionProportional);

        BoxView green = new BoxView { Color = Colors.Green };
        AbsoluteLayout.SetLayoutBounds(green, new Rect(0, 0.5, 25, 100));
        AbsoluteLayout.SetLayoutFlags(green, AbsoluteLayoutFlags.PositionProportional);

        BoxView red = new BoxView { Color = Colors.Red };
        AbsoluteLayout.SetLayoutBounds(red, new Rect(1, 0.5, 25, 100));
        AbsoluteLayout.SetLayoutFlags(red, AbsoluteLayoutFlags.PositionProportional);

        BoxView black = new BoxView { Color = Colors.Black };
        AbsoluteLayout.SetLayoutBounds(black, new Rect(0.5, 1, 100, 25));
        AbsoluteLayout.SetLayoutFlags(black, AbsoluteLayoutFlags.PositionProportional);

        Label label = new Label { Text = "Centered text" };
        AbsoluteLayout.SetLayoutBounds(label, new Rect(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 `Rect` 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.

BindableLayout

9/20/2022 • 6 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) bindable layouts enable any layout class that derives from the `Layout` 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`, `HorizontalStackLayout`, `StackLayout`, and `VerticalStackLayout` classes, which all derive from the `Layout` class.

When the `BindableLayout.ItemsSource` property is set to a collection of items and attached to a `Layout`-derived class, each item in the collection is added to the `Layout`-derived class for display. The `Layout`-derived class will then update its child views when the underlying collection changes.

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` class, it's not always practical to do so, particularly for the `AbsoluteLayout` and `Grid` 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 `BindableLayout.ItemsSource` property to any collection that implements `IEnumerable`, and attaching it to a `Layout`-derived class:

```
<Grid BindableLayout.ItemsSource="{Binding Items}" />
```

The equivalent C# code is:

```
IEnumerable<string> items = ...;
Grid 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>
            <Image Source="{Binding}"
                  Aspect="AspectFill"
                  WidthRequest="44"
                  HeightRequest="44"
                  ...
                  />
        </DataTemplate>
    </BindableLayout.ItemTemplate>
</StackLayout>
```

The equivalent C# code is:

```
DataTemplate imageTemplate = ...;
StackLayout stackLayout = new StackLayout();
BindableLayout.SetItemsSource(stackLayout, viewModel.User.TopFollowers);
BindableLayout.SetItemTemplate(stackLayout, imageTemplate);
```

In this example, every item in the `TopFollowers` collection will be displayed by an `Image` view defined in the `DataTemplate`:

Top Followers



For more information about data templates, see [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 { ... };
FlexLayout flexLayout = new FlexLayout();
BindableLayout.SetItemsSource(flexLayout, viewModel.User.FavoriteTech);
BindableLayout.SetItemTemplateSelector(flexLayout, dataTemplateSelector);
```

The following example shows the `TechItemTemplateSelector` class:

```
public class TechItemTemplateSelector : DataTemplateSelector
{
    public DataTemplate DefaultTemplate { get; set; }
    public DataTemplate MAUITemplate { get; set; }

    protected override DataTemplate OnSelectTemplate(object item, BindableObject container)
    {
        return (string)item == ".NET MAUI" ? MAUITemplate : DefaultTemplate;
    }
}
```

The `TechItemTemplateSelector` class defines `DefaultTemplate` and `MAUITemplate` `DataTemplate` properties that are set to different data templates. The `OnSelectTemplate` method returns the `MAUITemplate`, which displays an item in dark red with a heart next to it, when the item is equal to ".NET MAUI". When the item isn't equal to ".NET MAUI", the `OnSelectTemplate` method returns the `DefaultTemplate`, which displays an item using the default color of a `Label`:

Favorite Tech

.NET MAUI❤️ C# XAML SkiaSharp Azure

For more information about data template selectors, see [Create a 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:

Achievements

No achievements

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.

Achievements

None.

Try harder and return later?

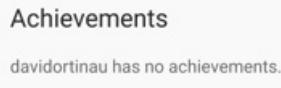
Similarly, the `EmptyViewTemplate` can be set to a `DataTemplate`, which will be displayed when the `ItemsSource` property is `null`, or when the collection specified by the `ItemsSource` property is `null` or empty. The `DataTemplate` can contain a single view, or a view that contains multiple child 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:

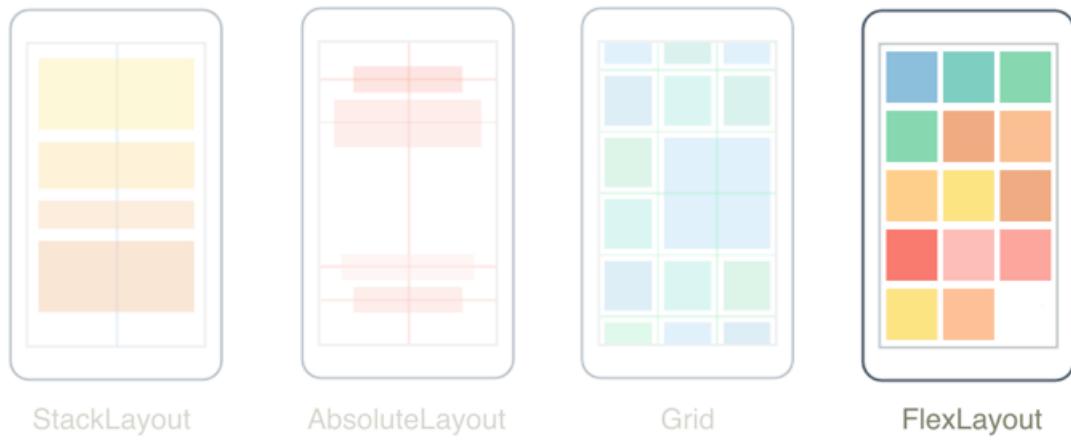
```
void ToggleEmptyView(bool isToggled)
{
    object view = isToggled ? Resources["BasicEmptyView"] : Resources["AdvancedEmptyView"];
    BindableLayout.SetEmptyView(stackLayout, view);
}
```

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.

FlexLayout

9/20/2022 • 14 minutes to read • [Edit Online](#)

 [Browse the sample](#)



The .NET Multi-platform App UI (.NET MAUI) `FlexLayout` is a layout that can arrange its children horizontally and vertically in a stack, and can also wrap its children if there are too many to fit in a single row or column. In addition, `FlexLayout` can control orientation and alignment, and adapt to different screen sizes. `FlexLayout` is based on the Cascading Style Sheets (CSS) [Flexible Box Layout Module](#).

The `FlexLayout` class defines the following properties:

- `AlignContent`, of type `FlexAlignContent`, which determines how the layout engine will distribute space between and around children that have been laid out on multiple lines. The default value of this property is `Stretch`. For more information, see [AlignContent](#).
- `AlignItems`, of type `FlexAlignItems`, which indicates how the layout engine will distribute space between and around children along the cross axis. The default value of this property is `Stretch`. For more information, see [AlignItems](#).
- `Direction`, of type `FlexDirection`, which defines the direction and main axis of children. The default value of this property is `Row`. For more information, see [Direction](#).
- `JustifyContent`, of type `FlexJustify`, which specifies how space is distributed between and around children along the main axis. The default value of this property is `Start`. For more information, see [JustifyContent](#).
- `Position`, of type `FlexPosition`, which determines whether the position of children are relative to each other, or by using fixed values. The default value of this property is `Relative`.
- `Wrap`, of type `FlexWrap`, which controls whether children are laid out in a single line or in multiple lines. The default value of this property is `NoWrap`. For more information, see [Wrap](#).
- `AlignSelf`, of type `FlexAlignSelf`, which is an attached property that indicates how the layout engine will distribute space between and around children for a specific child along the cross axis. The default value of this property is `Auto`. For more information, see [AlignSelf](#).
- `Basis`, of type `FlexBasis`, which is an attached property that defines the initial main axis dimension of the child. The default value of this property is `Auto`. For more information, see [Basis](#).
- `Grow`, of type `float`, which is an attached property that specifies the amount of available space the child should use on the main axis. The default value of this property is 0.0. A validation callback ensures that when the property is set, its value is greater than or equal to 0. For more information, see [Grow](#).
- `Order`, of type `int`, which is an attached property that determines whether the child should be laid out

before or after other children in the container. The default value of this property is 0. For more information, see [Order](#).

- `Shrink`, of type `float`, which is an attached property that controls how a child should shrink so that all children can fit inside the container. The default value of this property is 1.0. A validation callback ensures that when the property is set, its value is greater than or equal to 0. For more information, see [Shrink](#).

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings and styled.

IMPORTANT

When items in a `FlexLayout` are arranged in a column, the `FlexLayout` has a vertical *main axis* and a horizontal *cross axis*. When items in a `FlexLayout` are arranged in a row, the `FlexLayout` has a horizontal *main axis* and a vertical *cross axis*.

Orientation and alignment

The `Direction`, `Wrap`, `JustifyContent`, `AlignItems`, `AlignContent`, and `Position` bindable properties can be set on a `FlexLayout` to control orientation and alignment of all children.

Direction

The `Direction` property, of type `FlexDirection`, defines the direction and main axis of children. The `FlexDirection` enumeration defines the following members:

- `Column`, which indicates that children should be stacked vertically.
- `ColumnReverse` (or "column-reverse" in XAML), which indicates that children should be stacked vertically in reverse order.
- `Row`, which indicates that children should be stacked horizontally. This is the default value of the `Direction` property.
- `RowReverse` (or "row-reverse" in XAML), which indicates that children should be stacked horizontally in reverse order.

When the `Direction` property is set to `Column`, or `ColumnReverse`, the main-axis will be the y-axis and items will be stacked vertically. When the `Direction` property is set to `Row`, or `RowReverse`, the main-axis will be the x-axis and children will be stacked horizontally.

NOTE

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 the two additional strings shown in parentheses.

Wrap

The `Wrap` property, of type `FlexWrap`, controls whether children are laid out in a single line or in multiple lines. The `FlexWrap` enumeration defines the following members:

- `NoWrap`, which indicates that children are laid out in a single line. This is the default value of the `Wrap` property.
- `Wrap`, which indicates that items are laid out in multiple lines if needed.
- `Reverse` (or "wrap-reverse" in XAML), which indicates that items are laid out in multiple lines if needed, in reverse order.

When the `Wrap` property is set to `NoWrap` and the main axis is constrained, and the main axis is not wide or tall

enough to fit all the children, the `FlexLayout` attempts to make the items smaller. You can control the shrink factor of children with the `Shrink` attached bindable property.

When the `Wrap` property is set to `Wrap` or `WrapReverse`, the `AlignContent` property can be used to specify how the lines should be distributed.

JustifyContent

The `JustifyContent` property, of type `FlexJustify`, specifies how space is distributed between and around children along the main axis. The `FlexJustify` enumeration defines the following members:

- `Start` (or "flex-start" in XAML), which indicates that children should be aligned at the start. This is the default value of the `JustifyContent` property.
- `Center`, which indicates that children should be aligned around the center.
- `End` (or "flex-end" in XAML), which indicates that children should be aligned at the end.
- `SpaceBetween` (or "space-between" in XAML), which indicates that children should be evenly distributed, with the first child being at the start and the last child being at the end.
- `SpaceAround` (or "space-around" in XAML), which indicates that children should be evenly distributed, with the first and last children having a half-size space.
- `SpaceEvenly`, which indicates that children should be evenly distributed, with all children having equal space around them.

AlignItems

The `AlignItems` property, of type `FlexAlignItems`, indicates how the layout engine will distribute space between and around children along the cross axis. The `FlexAlignItems` enumeration defines the following members:

- `Stretch`, which indicates that children should be stretched out. This is the default value of the `AlignItems` property.
- `Center`, which indicates that children should be aligned around the center.
- `Start` (or "flex-start" in XAML), which indicates that children should be aligned at the start.
- `End` (or "flex-end" in XAML), which indicates that children should be aligned at the end.

This is one of two properties that indicates how children are aligned on the cross axis. Within each row, children are stretched or aligned on the start, center, or end of each item.

For any individual child, the `AlignItems` setting can be overridden with the `AlignSelf` attached bindable property.

AlignContent

The `AlignContent` property, of type `FlexAlignContent`, determines how the layout engine will distribute space between and around children that have been laid out on multiple lines. The `FlexAlignContent` enumeration defines the following members:

- `Stretch`, which indicates that children should be stretched out. This is the default value of the `AlignContent` property.
- `Center`, which indicates that children should be aligned around the center.
- `Start` (or "flex-start" in XAML), which indicates that children should be aligned at the start.
- `End` (or "flex-end" in XAML), which indicates that children should be aligned at the end.
- `SpaceBetween` (or "space-between" in XAML), which indicates that children should be evenly distributed, with the first child being at the start and the last child being at the end.
- `SpaceAround` (or "space-around" in XAML), which indicates that children should be evenly distributed, with the first and last children having a half-size space.
- `SpaceEvenly`, which indicates that children should be evenly distributed, with all children having equal space

around them.

The `AlignContent` property has no effect when there is only one row or column.

Child alignment and sizing

The `AlignSelf`, `Order`, `Basis`, `Grow`, and `Shrink` attached bindable properties can be set on children of the `FlexLayout` to control child orientation, alignment, and sizing.

AlignSelf

The `AlignSelf` property, of type `FlexAlignSelf`, indicates how the layout engine will distribute space between and around children for a specific child along the cross axis. The `FlexAlignSelf` enumeration defines the following members:

- `Auto`, which indicates that a child should be aligned according to the alignment value of its parent. This is the default value of the `AlignSelf` property.
- `Stretch`, which indicates that a child should be stretched out.
- `Center`, which indicates that a child should be aligned around the center.
- `Start` (or "flex-start" in XAML), which indicates that a child should be aligned at the start.
- `End` (or "flex-end" in XAML), which indicates that a child should be aligned at the end.

For any individual child of the `FlexLayout`, this property overrides the `AlignItems` property set on the `FlexLayout`. The default setting of `Auto` means to use the `AlignItems` setting.

In XAML, this property is set on a child without any reference to its `FlexLayout` parent:

```
<Label FlexLayout.AlignSelf="Center"
      ... />
```

The equivalent C# code is:

```
Label label = new Label();
FlexLayout.SetAlignSelf(label, FlexAlignSelf.Center);
```

Order

The `order` property, of type `int`, enables you to change the order that children of the `FlexLayout` are arranged. The default value of this property is 0.

Usually, children are arranged in the order in which they are added to the `FlexLayout`. However, this order can be overridden by setting this property to a non-zero integer value on one or more children. The `FlexLayout` then arranges its children based on their `order` property values. Children with the same `order` property values are arranged in the order in which they are added to the `FlexLayout`.

Basis

The `Basis` property, of type `FlexBasis`, defines the amount of space that's allocated to a child on the main axis. The value specified by this property is the size along the main axis of the parent `FlexLayout`. Therefore, this property indicates the width of a child when children are arranged in rows, or the height of a child when children are arranged in columns. This property is called *basis* because it specifies a size that is the basis of all subsequent layout.

The `FlexBasis` type is a structure that enables size to be specified in device-independent units, or as a percentage of the size of the `FlexLayout`. The default value of the `Basis` property is `Auto`, which means that the child's requested width or height is used.

In XAML, you can use a number for a size in device-independent units:

```
<Label FlexLayout.Basis="40"  
      ... />
```

The equivalent C# code is:

```
FlexLayout.SetBasis(label, 40);
```

In XAML, a percentage can be specified as follows:

```
<Label FlexLayout.Basis="25%"  
      ... />
```

The equivalent C# code is:

```
FlexLayout.SetBasis(label, new FlexBasis(0.25f, true));
```

The first argument to the `FlexBasis` constructor is a fractional `float` value that must be in the range of 0 to 1.

The second argument indicates that the size is relative, rather than absolute.

Grow

The `Grow` property, of type `float`, specifies the amount of available space the child should use on the main axis. The default value of this property is 0.0, and its value must be greater than or equal to 0.

The `Grow` property is used when the `Wrap` property is set to `NoWrap` and a row of children has a total width less than the width of the `FlexLayout`, or a column of children has a shorter height than the `FlexLayout`. The `Grow` property indicates how to apportion the leftover space among the children. If a single child is given a positive `Grow` value, then that child takes up all the remaining space. Alternatively, the remaining space can also be allocated among two or more children.

Shrink

The `Shrink` property, of type `float`, controls how a child should shrink so that all children can fit inside the container. The default value of this property is 1.0, and its value must be greater than or equal to 0.

The `Shrink` property is used 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.

TIP

The `Grow` and `Shrink` values can both be set to accommodate situations where the aggregate child sizes might sometimes be less than or sometimes greater than the size of the `FlexLayout`.

Examples

The following examples demonstrate common uses of `FlexLayout`.

Stack

A `FlexLayout` can substitute for a `StackLayout`:

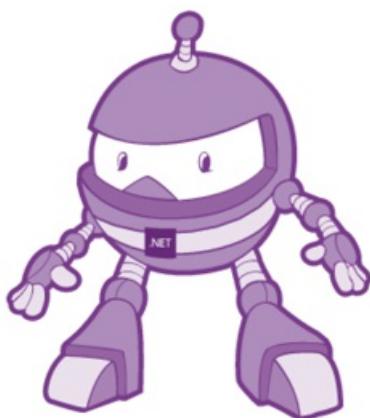
```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="FlexLayoutDemos.Views.SimpleStackPage"
    Title="Simple Stack">
    <FlexLayout Direction="Column"
        AlignItems="Center"
        JustifyContent="SpaceEvenly">
        <Label Text="FlexLayout in Action"
            FontSize="18" />
        <Image Source="dotnet_bot_branded.png"
            HeightRequest="300" />
        <Button Text="Do-Nothing Button" />
        <Label Text="Another Label" />
    </FlexLayout>
</ContentPage>

```

In this example, the `Direction` property is set to `Column`, which causes the children of the `FlexLayout` to be arranged in a single column. The `AlignItems` property is set to `Center`, which causes each child to be horizontally centered. The `JustifyContent` property is set to `SpaceEvenly` which allocates all leftover vertical space equally between all the children, above the first child and below the last child:

FlexLayout in Action



Do-Nothing Button

Another Label

NOTE

The `AlignSelf` attached property can be used to override the `AlignItems` property for a specific child.

Wrap items

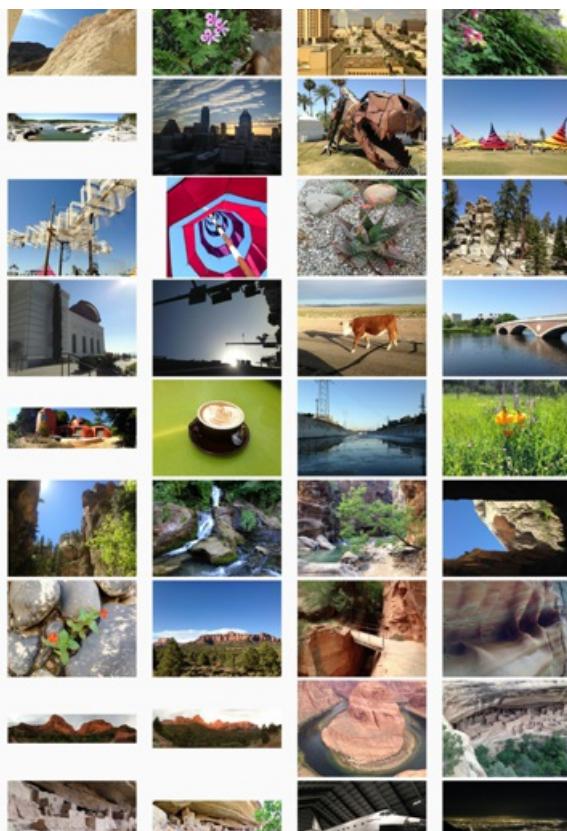
A `FlexLayout` can wrap its children to additional rows or columns:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="FlexLayoutDemos.Views.PhotoWrappingPage"
    Title="Photo Wrapping">
    <Grid>
        <ScrollView>
            <FlexLayout x:Name="flexLayout"
                Wrap="Wrap"
                JustifyContent="SpaceAround" />
        </ScrollView>
        ...
    </Grid>
</ContentPage>

```

In this example, the `Direction` property of the `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 set to `Wrap`, which causes children to wrap to the next row if there are too many children to fit on a row. The `JustifyContent` property is set to `SpaceAround` which allocates all leftover space on the main axis so that each child is surrounded by the same amount of space:



The code-behind file for this example retrieves a collection of photos and adds them to the `FlexLayout`.

In addition, the `FlexLayout` is a child of a `ScrollView`. Therefore, 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.

Page layout

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. This layout can be realized with a `FlexLayout`.

The following example shows an implementation of this layout using a `FlexLayout` nested in another:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="FlexLayoutDemos.Views.HolyGrailLayoutPage"
    Title="Holy Grail Layout">

    <FlexLayout Direction="Column">

        <!-- Header -->
        <Label Text="HEADER"
            FontSize="18"
            BackgroundColor="Aqua"
            HorizontalTextAlignment="Center" />

        <!-- Body -->
        <FlexLayout FlexLayout.Grow="1">

            <!-- Content -->
            <Label Text="CONTENT"
                FontSize="18"
                BackgroundColor="Gray"
                HorizontalTextAlignment="Center"
                VerticalTextAlignment="Center"
                FlexLayout.Grow="1" />

            <!-- Navigation items-->
            <BoxView FlexLayout.Basis="50"
                FlexLayout.Order="-1"
                Color="Blue" />

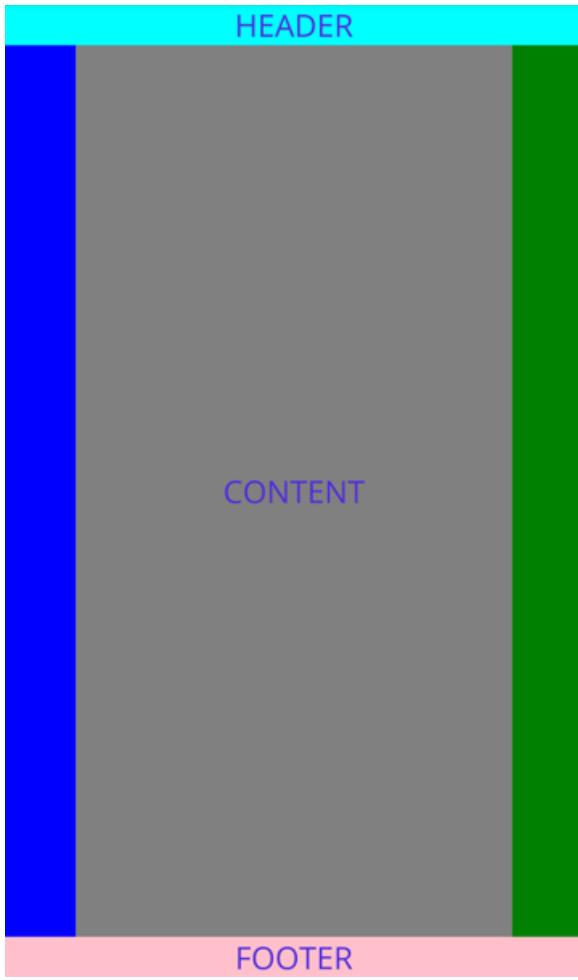
            <!-- Aside items -->
            <BoxView FlexLayout.Basis="50"
                Color="Green" />

        </FlexLayout>

        <!-- Footer -->
        <Label Text="FOOTER"
            FontSize="18"
            BackgroundColor="Pink"
            HorizontalTextAlignment="Center" />
    </FlexLayout>
</ContentPage>

```

The navigation and aside areas are rendered with a `BoxView` on the left and right. The first `FlexLayout` 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:



In this example, the `Order` property is set on the first `BoxView` to a value less than its siblings to cause it to appear as the first item in the row. The `Basis` property is set on both `BoxView` objects to give them a width of 50 device-independent units. The `Grow` property is set on the nested `FlexLayout` to indicate that this `FlexLayout` should occupy all of the unused vertical space within the outer `FlexLayout`. In addition, the `Grow` property is set on the `Label` representing the content, to indicate that this content is to occupy all of the unused horizontal space within the nested `FlexLayout`.

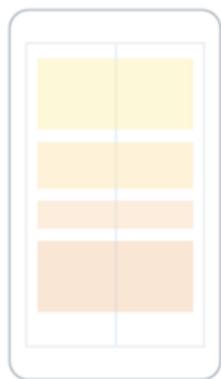
NOTE

There's also a `Shrink` property that you can use when the size of children exceeds the size of the `FlexLayout` but wrapping is not desired.

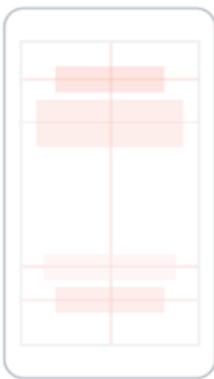
Grid

9/20/2022 • 14 minutes to read • [Edit Online](#)

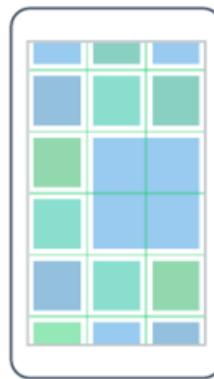
 [Browse the sample](#)



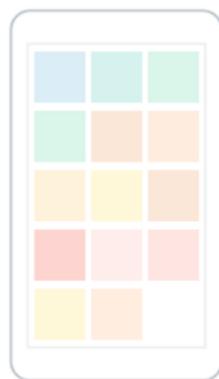
StackLayout



AbsoluteLayout



Grid



FlexLayout

The .NET Multi-platform App UI (.NET MAUI) `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` 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](#) or [CollectionView](#).

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 0.
- `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 0.
- `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.

Rows and columns

By default, a `Grid` contains one row and one column:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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:

By default, a `Grid` contains one row and one column.

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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="GridDemos.Views.XAML.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:

```

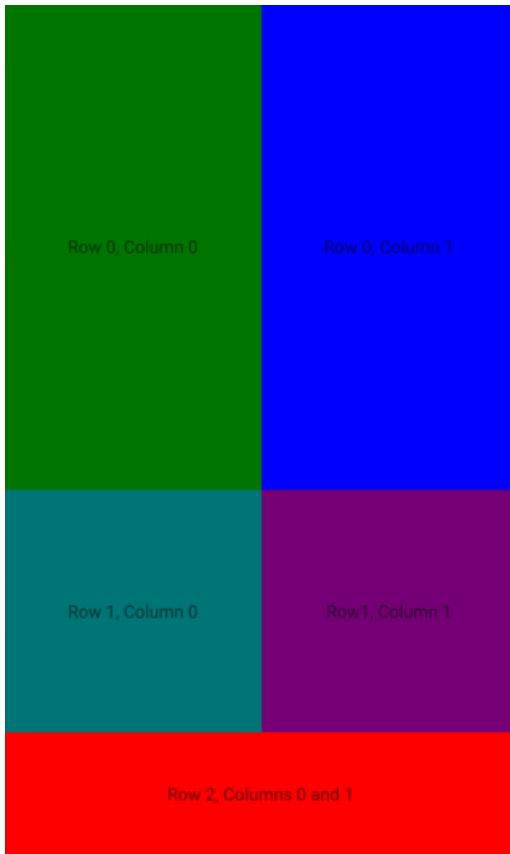
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="GridDemos.Views.XAML.BasicGridPage"
    Title="Basic Grid demo">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition />
            <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 BasicGridPage : ContentPage
{
    public BasicGridPage()
    {
        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 need to be added to the
        // Grid to obtain the default row and column settings.
        grid.Add(new BoxView
        {
            Color = Colors.Green
        });
        grid.Add(new Label
        {
            Text = "Row 0, Column 0",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = 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.Add(new BoxView
{
    Color = Colors.Blue
}, 1, 0);
grid.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.Add(new BoxView
{
    Color = Colors.Teal
}, 0, 1);
grid.Add(new Label
{
    Text = "Row 1, Column 0",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center
}, 0, 1);

// This BoxView and Label are in row 1 and column 1, which are specified as arguments
// to the Add method overload.
grid.Add(new BoxView
{
    Color = Colors.Purple
}, 1, 1);
grid.Add(new Label
{
    Text = "Row1, Column 1",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center
}, 1, 1);

// 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 = Colors.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.Add(boxView);
grid.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.

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 .NET MAUI 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 and columns have no space between them. This can be changed by setting the `RowSpacing` and `ColumnSpacing` properties, respectively:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              x:Class="GridDemos.Views.XAML.GridSpacingPage"
              Title="Grid spacing demo">
    <Grid RowSpacing="6"
          ColumnSpacing="6">
        ...
    </Grid>
</ContentPage>
```

This example creates a `Grid` whose rows and columns are separated by 6 device-independent units of space:

**TIP**

The `RowSpacing` and `ColumnSpacing` properties can be set to negative values to make cell contents overlap.

The equivalent C# code is:

```
public class GridSpacingPage : ContentPage
{
    public GridSpacingPage()
    {
        Grid grid = new Grid
        {
            RowSpacing = 6,
            ColumnSpacing = 6,
            ...
        };
        ...

        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`

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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="GridDemos.Views.XAML.GridAlignmentPage"
    Title="Grid alignment demo">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

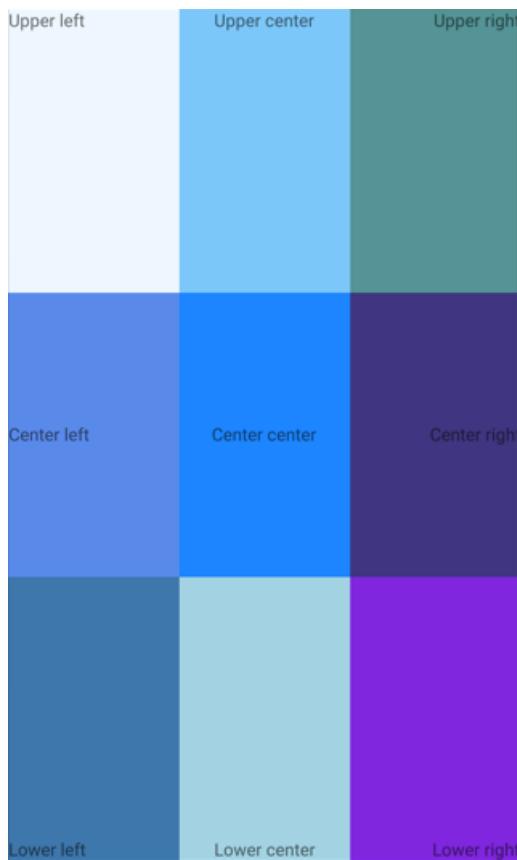
    <BoxView Color="AliceBlue" />
    <Label Text="Upper left"
        HorizontalOptions="Start"
        VerticalOptions="Start" />
    <BoxView Grid.Column="1"
        Color="LightSkyBlue" />
    <Label Grid.Column="1"
        Text="Upper center"
        HorizontalOptions="Center"
        VerticalOptions="Start"/>
    <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="DarkKhaki" />
    <Label Grid.Row="2"
        Grid.Column="2"
        Text="Lower right"
        HorizontalOptions="End"
        VerticalOptions="End" />
</Grid>
```

```

        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 GridAlignmentPage : ContentPage
{
    public GridAlignmentPage()
    {
        Grid grid = new Grid
        {
            RowDefinitions =
            {
                new RowDefinition(),
                new RowDefinition(),
                new RowDefinition()
            },
            ColumnDefinitions =
            {
                new ColumnDefinition(),
                new ColumnDefinition(),
                new ColumnDefinition()
            }
        };
    }
}

```

```
        new ColumnDefinition(),
    }
};

// Row 0
grid.Add(new BoxView
{
    Color = Colors.AliceBlue
});
grid.Add(new Label
{
    Text = "Upper left",
    HorizontalOptions = LayoutOptions.Start,
    VerticalOptions = LayoutOptions.Start
});

grid.Add(new BoxView
{
    Color = Colors.LightSkyBlue
}, 1, 0);
grid.Add(new Label
{
    Text = "Upper center",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Start
}, 1, 0);

grid.Add(new BoxView
{
    Color = Colors.CadetBlue
}, 2, 0);
grid.Add(new Label
{
    Text = "Upper right",
    HorizontalOptions = LayoutOptions.End,
    VerticalOptions = LayoutOptions.Start
}, 2, 0);

// Row 1
grid.Add(new BoxView
{
    Color = Colors.CornflowerBlue
}, 0, 1);
grid.Add(new Label
{
    Text = "Center left",
    HorizontalOptions = LayoutOptions.Start,
    VerticalOptions = LayoutOptions.Center
}, 0, 1);

grid.Add(new BoxView
{
    Color = Colors.DodgerBlue
}, 1, 1);
grid.Add(new Label
{
    Text = "Center center",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center
}, 1, 1);

grid.Add(new BoxView
{
    Color = Colors.DarkSlateBlue
}, 2, 1);
grid.Add(new Label
{
    Text = "Center right",
    HorizontalOptions = LayoutOptions.End,
    VerticalOptions = LayoutOptions.Center
}, 2, 1);
```

```

    VerticalOptions = LayoutOptions.Center
}, 2, 1);

// Row 2
grid.Add(new BoxView
{
    Color = Colors.SteelBlue
}, 0, 2);
grid.Add(new Label
{
    Text = "Lower left",
    HorizontalOptions = LayoutOptions.Start,
    VerticalOptions = LayoutOptions.End
}, 0, 2);

grid.Add(new BoxView
{
    Color = Colors.LightBlue
}, 1, 2);
grid.Add(new Label
{
    Text = "Lower center",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.End
}, 1, 2);

grid.Add(new BoxView
{
    Color = Colors.BlueViolet
}, 2, 2);
grid.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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:converters="clr-namespace:GridDemos.Converters"
    x:Class="GridDemos.Views.XAML.ColorSlidersGridPage"
    Title="Nested Grids demo">

    <ContentPage.Resources>
        <converters:DoubleToIntConverter x:Key="doubleToInt" />

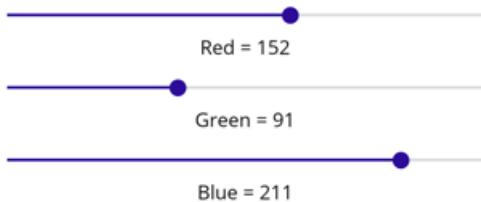
        <Style TargetType="Label">
            <Setter Property="HorizontalTextAlignment"
                Value="Center" />
        </Style>
    </ContentPage.Resources>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="500" />
            <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` 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.

The equivalent C# code is:

```
public class ColorSlidersGridPage : ContentPage
{
    BoxView boxView;
    Slider redSlider;
    Slider greenSlider;
    Slider blueSlider;

    public ColorSlidersGridPage()
    {
        // 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 { HeightRequest = 500 },
                new RowDefinition()
            }
        };
        rootGrid.Children.Add(boxView);
        Content = rootGrid;
    }
}
```

```

        NEW_ROWDEFINITION()
    }

};

boxView = new BoxView { Color = Colors.Black };
rootGrid.Add(boxView);

// Child page layout
Grid childGrid = new Grid
{
    Margin = new Thickness(20),
    RowDefinitions =
    {
        new RowDefinition(),
        new RowDefinition(),
        new RowDefinition(),
        new RowDefinition(),
        new RowDefinition(),
        new RowDefinition()
    }
};

DoubleToIntConverter doubleToInt = new DoubleToIntConverter();

redSlider = new Slider();
redSlider.ValueChanged += OnSliderValueChanged;
childGrid.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.Add(redLabel);

greenSlider = new Slider();
greenSlider.ValueChanged += OnSliderValueChanged;
Grid.SetRow(greenSlider, 2);
childGrid.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.Add(greenLabel);

blueSlider = new Slider();
blueSlider.ValueChanged += OnSliderValueChanged;
Grid.SetRow(blueSlider, 4);
childGrid.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.Add(blueLabel);

// Place the child Grid in the root Grid
rootGrid.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);
}
}

```

HorizontalStackLayout

9/20/2022 • 3 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `HorizontalStackLayout` organizes child views in a one-dimensional horizontal stack, and is a more performant alternative to a `StackLayout`. In addition, a `HorizontalStackLayout` can be used as a parent layout that contains other child layouts.

The `HorizontalStackLayout` defines the following properties:

- `Spacing`, of type `double`, indicates the amount of space between each child view. The default value of this property is 0.

This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings and styled.

The following XAML shows how to create a `HorizontalStackLayout` that contains different child views:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.HorizontalStackLayoutPage">
    <HorizontalStackLayout Margin="20">
        <Rectangle Fill="Red"
            HeightRequest="30"
            WidthRequest="30" />
        <Label Text="Red"
            FontSize="18" />
    </HorizontalStackLayout>
</ContentPage>
```

This example creates a `HorizontalStackLayout` containing a `Rectangle` and a `Label` object. By default, there is no space between the child views:



NOTE

The value of the `Margin` property represents the distance between an element and its adjacent elements. For more information, see [Position controls](#).

Space between child views

The spacing between child views in a `HorizontalStackLayout` can be changed by setting the `Spacing` property to a `double` value:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.HorizontalStackLayoutPage">
    <HorizontalStackLayout Margin="20"
        Spacing="10">
        <Rectangle Fill="Red"
            HeightRequest="30"
            WidthRequest="30" />
        <Label Text="Red"
            FontSize="18" />
    </HorizontalStackLayout>
</ContentPage>

```

This example creates a `HorizontalStackLayout` containing a `Rectangle` and a `Label` object, that have ten device-independent units of space between them:



TIP

The `Spacing` property can be set to negative values to make child views overlap.

Position and size child views

The size and position of child views within a `HorizontalStackLayout` depends upon the values of the child views' `HeightRequest` and `WidthRequest` properties, and the values of their `VerticalOptions` properties. In a `HorizontalStackLayout`, child views expand to fill the available height when their size isn't explicitly set.

The `VerticalOptions` properties of a `HorizontalStackLayout`, and its child views, can be set to fields from the `LayoutOptions` struct, which encapsulates an *alignment* layout preference. This layout preference determines the position and size of a child view within its parent layout.

The following XAML example sets alignment preferences on each child view in the `HorizontalStackLayout`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.HorizontalStackLayoutPage">
    <HorizontalStackLayout Margin="20"
        HeightRequest="200">
        <Label Text="Start"
            BackgroundColor="Gray"
            VerticalOptions="Start" />
        <Label Text="Center"
            BackgroundColor="Gray"
            VerticalOptions="Center" />
        <Label Text="End"
            BackgroundColor="Gray"
            VerticalOptions="End" />
        <Label Text="Fill"
            BackgroundColor="Gray"
            VerticalOptions="Fill" />
    </HorizontalStackLayout>
</ContentPage>

```

In this example, alignment preferences are set on the `Label` objects to control their position within the `HorizontalStackLayout`. The `Start`, `Center`, `End`, and `Fill` fields are used to define the alignment of the `Label` objects within the parent `HorizontalStackLayout`:



A `HorizontalStackLayout` only respects the alignment preferences on child views that are in the opposite direction to the orientation of the layout. Therefore, the `Label` child views within the `HorizontalStackLayout` set their `VerticalOptions` properties to one of the alignment fields:

- `Start`, which positions the `Label` at the start of the `HorizontalStackLayout`.
- `Center`, which vertically centers the `Label` in the `HorizontalStackLayout`.
- `End`, which positions the `Label` at the end of the `HorizontalStackLayout`.
- `Fill`, which ensures that the `Label` fills the height of the `HorizontalStackLayout`.

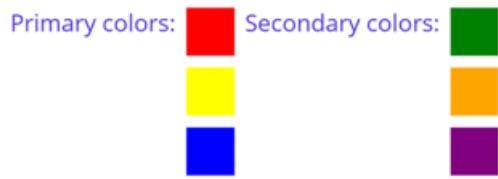
Nest `HorizontalStackLayout` objects

A `HorizontalStackLayout` can be used as a parent layout that contains other nested child layouts.

The following XAML shows an example of nesting `VerticalStackLayout` objects in a `HorizontalStackLayout`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.HorizontalStackLayoutPage">
    <HorizontalStackLayout Margin="20"
        Spacing="6">
        <Label Text="Primary colors:" />
        <VerticalStackLayout Spacing="6">
            <Rectangle Fill="Red"
                WidthRequest="30"
                HeightRequest="30" />
            <Rectangle Fill="Yellow"
                WidthRequest="30"
                HeightRequest="30" />
            <Rectangle Fill="Blue"
                WidthRequest="30"
                HeightRequest="30" />
        </VerticalStackLayout>
        <Label Text="Secondary colors:" />
        <VerticalStackLayout Spacing="6">
            <Rectangle Fill="Green"
                WidthRequest="30"
                HeightRequest="30" />
            <Rectangle Fill="Orange"
                WidthRequest="30"
                HeightRequest="30" />
            <Rectangle Fill="Purple"
                WidthRequest="30"
                HeightRequest="30" />
        </VerticalStackLayout>
    </HorizontalStackLayout>
</ContentPage>
```

In this example, the parent `HorizontalStackLayout` contains two nested `VerticalStackLayout` objects:



IMPORTANT

The deeper you nest layout objects, the more the nested layouts will impact performance.

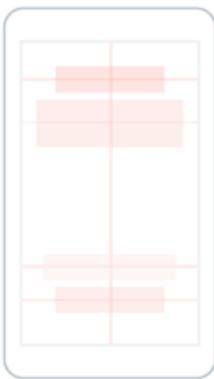
StackLayout

9/20/2022 • 7 minutes to read • [Edit Online](#)

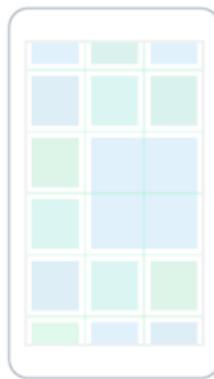
 [Browse the sample](#)



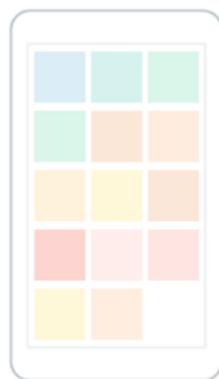
StackLayout



AbsoluteLayout



Grid



FlexLayout

The .NET Multi-platform App UI (.NET MAUI) `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 0.

These properties are backed by `BindableProperty` objects, which means that the properties can be targets of data bindings and styled.

Vertical orientation

The following XAML shows how to create a vertically oriented `StackLayout` that contains different child views:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.XAML.VerticalStackLayoutPage"
    Title="Vertical StackLayout demo">
    <StackLayout Margin="20">
        <Label Text="Primary colors" />
        <BoxView Color="Red"
            HeightRequest="40" />
        <BoxView Color="Yellow"
            HeightRequest="40" />
        <BoxView Color="Blue"
            HeightRequest="40" />
        <Label Text="Secondary colors" />
        <BoxView Color="Green"
            HeightRequest="40" />
        <BoxView Color="Orange"
            HeightRequest="40" />
        <BoxView Color="Purple"
            HeightRequest="40" />
    </StackLayout>
</ContentPage>

```

This example creates a vertical `StackLayout` containing `Label` and `BoxView` objects. By default, there's no space between the child views:



The equivalent C# code is:

```

public class VerticalStackLayoutPage : ContentPage
{
    public VerticalStackLayoutPage()
    {
        Title = "Vertical StackLayout demo";

        StackLayout stackLayout = new StackLayout { Margin = new Thickness(20) };

        stackLayout.Add(new Label { Text = "Primary colors" });
        stackLayout.Add(new BoxView { Color = Colors.Red, HeightRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Yellow, HeightRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Blue, HeightRequest = 40 });
        stackLayout.Add(new Label { Text = "Secondary colors" });
        stackLayout.Add(new BoxView { Color = Colors.Green, HeightRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Orange, HeightRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Purple, HeightRequest = 40 });

        Content = stackLayout;
    }
}

```

NOTE

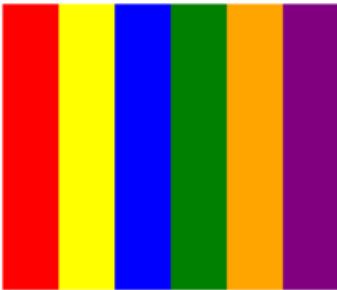
The value of the `Margin` property represents the distance between an element and its adjacent elements. For more information, see [Position controls](#).

Horizontal orientation

The following XAML shows how to create a horizontally oriented `StackLayout` by setting its `Orientation` property to `Horizontal`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.XAML.HorizontalStackLayoutPage"
    Title="Horizontal StackLayout demo">
    <StackLayout Margin="20"
        Orientation="Horizontal"
        HorizontalOptions="Center">
        <BoxView Color="Red"
            WidthRequest="40" />
        <BoxView Color="Yellow"
            WidthRequest="40" />
        <BoxView Color="Blue"
            WidthRequest="40" />
        <BoxView Color="Green"
            WidthRequest="40" />
        <BoxView Color="Orange"
            WidthRequest="40" />
        <BoxView Color="Purple"
            WidthRequest="40" />
    </StackLayout>
</ContentPage>
```

This example creates a horizontal `StackLayout` containing `BoxView` objects, with no space between the child views:



The equivalent C# code is:

```

public class HorizontalStackLayoutPage : ContentPage
{
    public HorizontalStackLayoutPage()
    {
        Title = "Horizontal StackLayout demo";

        StackLayout stackLayout = new StackLayout
        {
            Margin = new Thickness(20),
            Orientation = StackOrientation.Horizontal,
            HorizontalOptions = LayoutOptions.Center
        };

        stackLayout.Add(new BoxView { Color = Colors.Red, WidthRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Yellow, WidthRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Blue, WidthRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Green, WidthRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Orange, WidthRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Purple, WidthRequest = 40 });

        Content = stackLayout;
    }
}

```

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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.XAML.StackLayoutSpacingPage"
    Title="StackLayout Spacing demo">

    <StackLayout Margin="20"
        Spacing="6">
        <Label Text="Primary colors" />
        <BoxView Color="Red"
            HeightRequest="40" />
        <BoxView Color="Yellow"
            HeightRequest="40" />
        <BoxView Color="Blue"
            HeightRequest="40" />
        <Label Text="Secondary colors" />
        <BoxView Color="Green"
            HeightRequest="40" />
        <BoxView Color="Orange"
            HeightRequest="40" />
        <BoxView Color="Purple"
            HeightRequest="40" />
    </StackLayout>
</ContentPage>

```

This example creates a vertical `StackLayout` containing `Label` and `BoxView` objects that have six device-independent units of vertical space between them:

Primary colors



Secondary colors



TIP

The `Spacing` property can be set to negative values to make child views overlap.

The equivalent C# code is:

```
public class StackLayoutSpacingPage : ContentPage
{
    public StackLayoutSpacingPage()
    {
        Title = "StackLayout Spacing demo";

        StackLayout stackLayout = new StackLayout
        {
            Margin = new Thickness(20),
            Spacing = 6
        };

        stackLayout.Add(new Label { Text = "Primary colors" });
        stackLayout.Add(new BoxView { Color = Colors.Red, HeightRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Yellow, HeightRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Blue, HeightRequest = 40 });
        stackLayout.Add(new Label { Text = "Secondary colors" });
        stackLayout.Add(new BoxView { Color = Colors.Green, HeightRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Orange, HeightRequest = 40 });
        stackLayout.Add(new BoxView { Color = Colors.Purple, HeightRequest = 40 });

        Content = stackLayout;
    }
}
```

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 an *alignment* layout preference. This layout preference determines the position and size of a child view within its parent layout.

The following XAML example sets alignment preferences on each child view in the `StackLayout`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.XAML.AlignmentPage"
    Title="Alignment demo">
    <StackLayout Margin="20"
        Spacing="6">
        <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 AlignmentPage : ContentPage
{
    public AlignmentPage()
    {
        Title = "Alignment demo";

        StackLayout stackLayout = new StackLayout
        {
            Margin = new Thickness(20),
            Spacing = 6
        };

        stackLayout.Add(new Label { Text = "Start", BackgroundColor = Colors.Gray, HorizontalOptions =
LayoutOptions.Start });
        stackLayout.Add(new Label { Text = "Center", BackgroundColor = Colors.Gray, HorizontalOptions =
LayoutOptions.Center });
        stackLayout.Add(new Label { Text = "End", BackgroundColor = Colors.Gray, HorizontalOptions =
LayoutOptions.End });
        stackLayout.Add(new Label { Text = "Fill", BackgroundColor = Colors.Gray, HorizontalOptions =
LayoutOptions.Fill });

        Content = stackLayout;
    }
}
```

For more information about alignment, see [Align views in layouts](#).

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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.XAML.CombinedStackLayoutPage"
    Title="Combined StackLayouts demo">
    <StackLayout Margin="20">
        ...
        <Frame BorderColor="Black"
            Padding="5">
            <StackLayout Orientation="Horizontal"
                Spacing="15">
                <BoxView Color="Red"
                    WidthRequest="40" />
                <Label Text="Red"
                    FontSize="18"
                    VerticalOptions="Center" />
            </StackLayout>
        </Frame>
        <Frame BorderColor="Black"
            Padding="5">
            <StackLayout Orientation="Horizontal"
                Spacing="15">
                <BoxView Color="Yellow"
                    WidthRequest="40" />
                <Label Text="Yellow"
                    FontSize="18"
                    VerticalOptions="Center" />
            </StackLayout>
        </Frame>
        <Frame BorderColor="Black"
            Padding="5">
            <StackLayout Orientation="Horizontal"
                Spacing="15">
                <BoxView Color="Blue"
                    WidthRequest="40" />
                <Label Text="Blue"
                    FontSize="18"
                    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.

The equivalent C# code is:

```

public class CombinedStackLayoutPage : ContentPage
{
    public CombinedStackLayoutPage()
    {
        Title = "Combined StackLayouts demo";

        Frame frame1 = new Frame
        {
            BorderColor = Colors.Black,
            Padding = new Thickness(5)
        };
        StackLayout frame1StackLayout = new StackLayout
        {
            Orientation = StackOrientation.Horizontal,
            Spacing = 15
        };
        frame1StackLayout.Add(new BoxView { Color = Colors.Red, WidthRequest = 40 });
        frame1StackLayout.Add(new Label { Text = "Red", FontSize = 22, VerticalOptions =
LayoutOptions.Center });
        frame1.Content = frame1StackLayout;

        Frame frame2 = new Frame
        {
            BorderColor = Colors.Black,
            Padding = new Thickness(5)
        };
        StackLayout frame2StackLayout = new StackLayout
        {
            Orientation = StackOrientation.Horizontal,
            Spacing = 15
        };
        frame2StackLayout.Add(new BoxView { Color = Colors.Yellow, WidthRequest = 40 });
        frame2StackLayout.Add(new Label { Text = "Yellow", FontSize = 22, VerticalOptions =
LayoutOptions.Center });
        frame2.Content = frame2StackLayout;

        Frame frame3 = new Frame
        {
            BorderColor = Colors.Black,
            Padding = new Thickness(5)
        };
        StackLayout frame3StackLayout = new StackLayout
        {
            Orientation = StackOrientation.Horizontal,
            Spacing = 15
        };
        frame3StackLayout.Add(new BoxView { Color = Colors.Blue, WidthRequest = 40 });
        frame3StackLayout.Add(new Label { Text = "Blue", FontSize = 22, VerticalOptions =
LayoutOptions.Center });
        frame3.Content = frame3StackLayout;

        ...

        StackLayout stackLayout = new StackLayout { Margin = new Thickness(20) };
        stackLayout.Add(new Label { Text = "Primary colors" });
        stackLayout.Add(frame1);
        stackLayout.Add(frame2);
        stackLayout.Add(frame3);
        stackLayout.Add(new Label { Text = "Secondary colors" });
        stackLayout.Add(frame4);
        stackLayout.Add(frame5);
        stackLayout.Add(frame6);

        Content = stackLayout;
    }
}

```

VerticalStackLayout

9/20/2022 • 3 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `VerticalStackLayout` organizes child views in a one-dimensional vertical stack, and is a more performant alternative to a `StackLayout`. In addition, a `VerticalStackLayout` can be used as a parent layout that contains other child layouts.

The `VerticalStackLayout` defines the following properties:

- `Spacing`, of type `double`, indicates the amount of space between each child view. The default value of this property is 0.

This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings and styled.

The following XAML shows how to create a `VerticalStackLayout` that contains different child views:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.VerticalStackLayoutPage">
    <VerticalStackLayout Margin="20">
        <Label Text="Primary colors" />
        <Rectangle Fill="Red"
            HeightRequest="30"
            WidthRequest="300" />
        <Rectangle Fill="Yellow"
            HeightRequest="30"
            WidthRequest="300" />
        <Rectangle Fill="Blue"
            HeightRequest="30"
            WidthRequest="300" />
        <Label Text="Secondary colors" />
        <Rectangle Fill="Green"
            HeightRequest="30"
            WidthRequest="300" />
        <Rectangle Fill="Orange"
            HeightRequest="30"
            WidthRequest="300" />
        <Rectangle Fill="Purple"
            HeightRequest="30"
            WidthRequest="300" />
    </VerticalStackLayout>
</ContentPage>
```

This example creates a `VerticalStackLayout` containing `Label` and `Rectangle` objects. By default, there is no space between the child views:

Primary colors



Secondary colors



NOTE

The value of the `Margin` property represents the distance between an element and its adjacent elements. For more information, see [Position controls](#).

Space between child views

The spacing between child views in a `VerticalStackLayout` can be changed by setting the `Spacing` property to a `double` value:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.VerticalStackLayoutPage">
    <VerticalStackLayout Margin="20"
        Spacing="10">
        <Label Text="Primary colors" />
        <Rectangle Fill="Red"
            HeightRequest="30"
            WidthRequest="300" />
        <Rectangle Fill="Yellow"
            HeightRequest="30"
            WidthRequest="300" />
        <Rectangle Fill="Blue"
            HeightRequest="30"
            WidthRequest="300" />
        <Label Text="Secondary colors" />
        <Rectangle Fill="Green"
            HeightRequest="30"
            WidthRequest="300" />
        <Rectangle Fill="Orange"
            HeightRequest="30"
            WidthRequest="300" />
        <Rectangle Fill="Purple"
            HeightRequest="30"
            WidthRequest="300" />
    </VerticalStackLayout>
</ContentPage>
```

This example creates a `VerticalStackLayout` containing `Label` and `Rectangle` objects that have ten device-independent units of space between the child views:

Primary colors



Secondary colors



TIP

The `Spacing` property can be set to negative values to make child views overlap.

Position and size child views

The size and position of child views within a `VerticalStackLayout` depends upon the values of the child views' `HeightRequest` and `WidthRequest` properties, and the values of their `HorizontalOptions` properties. In a `VerticalStackLayout`, child views expand to fill the available width when their size isn't explicitly set.

The `HorizontalOptions` properties of a `VerticalStackLayout`, and its child views, can be set to fields from the `LayoutOptions` struct, which encapsulates an *alignment* layout preference. This layout preference determines the position and size of a child view within its parent layout.

TIP

Don't set the `HorizontalOptions` property of a `VerticalStackLayout` unless you need to. The default value of `LayoutOptions.Fill` allows for the best layout optimization. Changing this property has a cost and consumes memory, even when setting it back to its default value.

The following XAML example sets alignment preferences on each child view in the `VerticalStackLayout`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.VerticalStackLayoutPage">
    <VerticalStackLayout Margin="20"
        Spacing="6">
        <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" />
    </VerticalStackLayout>
</ContentPage>
```

In this example, alignment preferences are set on the `Label` objects to control their position within the `VerticalStackLayout`. The `Start`, `Center`, `End`, and `Fill` fields are used to define the alignment of the `Label` objects within the parent `VerticalStackLayout`:



A `VerticalStackLayout` only respects the alignment preferences on child views that are in the opposite direction to the orientation of the layout. Therefore, the `Label` child views within the `VerticalStackLayout` set their `HorizontalOptions` properties to one of the alignment fields:

- `Start`, which positions the `Label` on the left-hand side of the `VerticalStackLayout`.
- `Center`, which centers the `Label` in the `VerticalStackLayout`.
- `End`, which positions the `Label` on the right-hand side of the `VerticalStackLayout`.
- `Fill`, which ensures that the `Label` fills the width of the `VerticalStackLayout`.

Nest `VerticalStackLayout` objects

A `VerticalStackLayout` can be used as a parent layout that contains other nested child layouts.

The following XAML shows an example of nesting `HorizontalStackLayout` objects in a `VerticalStackLayout`:

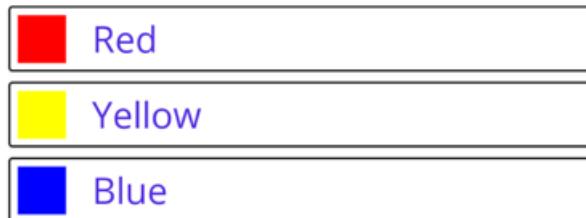
```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="StackLayoutDemos.Views.VerticalStackLayoutPage">
    <VerticalStackLayout Margin="20"
        Spacing="6">
        <Label Text="Primary colors" />
        <Frame BorderColor="Black"
            Padding="5">
            <HorizontalStackLayout Spacing="15">
                <Rectangle Fill="Red"
                    HeightRequest="30"
                    WidthRequest="30" />
                <Label Text="Red"
                    FontSize="18" />
            </HorizontalStackLayout>
        </Frame>
        <Frame BorderColor="Black"
            Padding="5">
            <HorizontalStackLayout Spacing="15">
                <Rectangle Fill="Yellow"
                    HeightRequest="30"
                    WidthRequest="30" />
                <Label Text="Yellow"
                    FontSize="18" />
            </HorizontalStackLayout>
        </Frame>
        <Frame BorderColor="Black"
            Padding="5">
            <HorizontalStackLayout Spacing="15">
                <Rectangle Fill="Blue"
                    HeightRequest="30"
                    WidthRequest="30" />
                <Label Text="Blue"
                    FontSize="18" />
            </HorizontalStackLayout>
        </Frame>
    </VerticalStackLayout>
</ContentPage>

```

In this example, the parent `VerticalStackLayout` contains nested `HorizontalStackLayout` objects inside `Frame` objects:

Primary colors



IMPORTANT

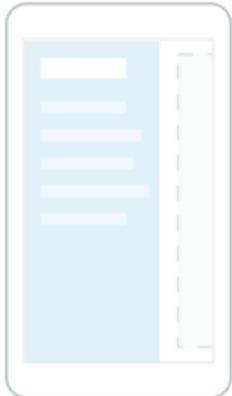
The deeper you nest layout objects, the more the nested layouts will impact performance.

ContentPage

9/20/2022 • 2 minutes to read • [Edit Online](#)



ContentPage



FlyoutPage



NavigationPage



TabbedPage

The .NET Multi-platform App UI (.NET MAUI) `ContentPage` displays a single view, which is often a layout such as `Grid` or `StackLayout`, and is the most common page type.

`ContentPage` defines a `Content` property, of type `View`, which defines the view that represents the page's content. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled. In addition, `ContentPage` inherits `Title`, `IconImageSource`, `BackgroundImageSource`, `IsBusy`, and `Padding` bindable properties from the `Page` class.

NOTE

The `Content` property is the content property of the `ContentPage` class, and therefore does not need to be explicitly set from XAML.

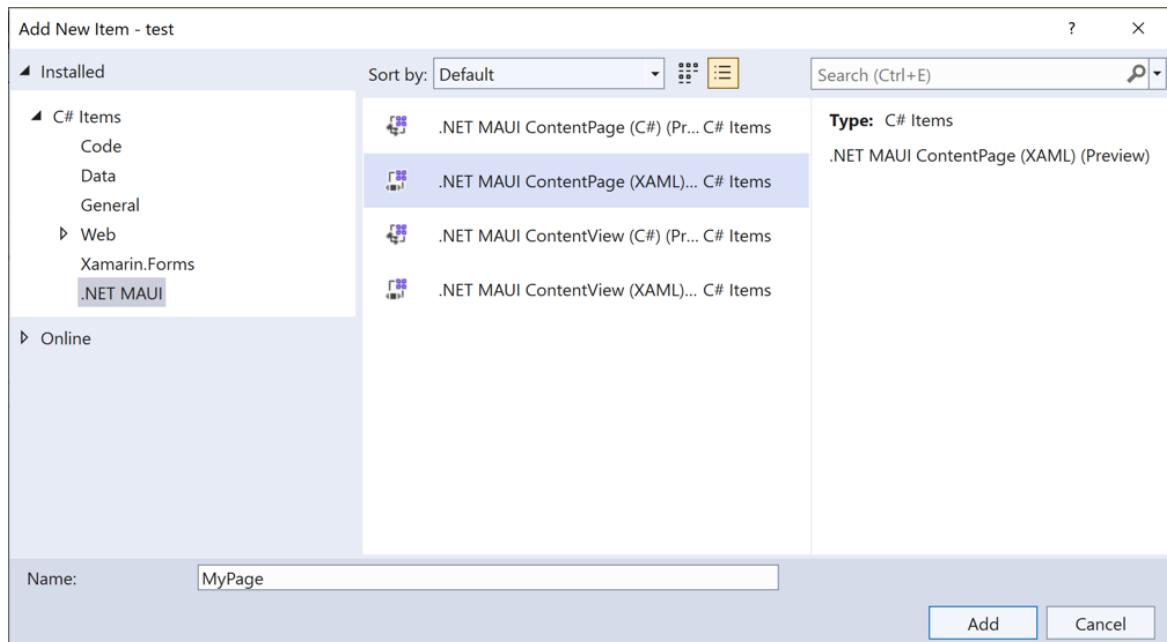
.NET MAUI apps typically contain multiple pages that derive from `ContentPage`, and navigation between these pages can be performed. For more information about page navigation, see [NavigationPage](#).

A `ContentPage` can be templated with a control template. For more information, see [Control templates](#).

Create a ContentPage

To add a `ContentPage` to a .NET MAUI app:

1. In **Solution Explorer** right-click on your project or folder in your project, and select **New Item....**
2. In the **Add New Item** dialog, expand **Installed > C# Items**, select **.NET MAUI**, and select the **.NET MAUI ContentPage (XAML)** item template, enter a suitable page name, and click the **Add** button:



Visual Studio then creates a new `ContentPage`-derived page, which will be similar to the following example:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MyMauiApp.MyPage"
    Title="MyPage"
    BackgroundColor="White">
    <StackLayout>
        <Label Text="Welcome to .NET MAUI!">
            VerticalOptions="Center"
            HorizontalOptions="Center" />
        <!-- Other views go here -->
    </StackLayout>
</ContentPage>
```

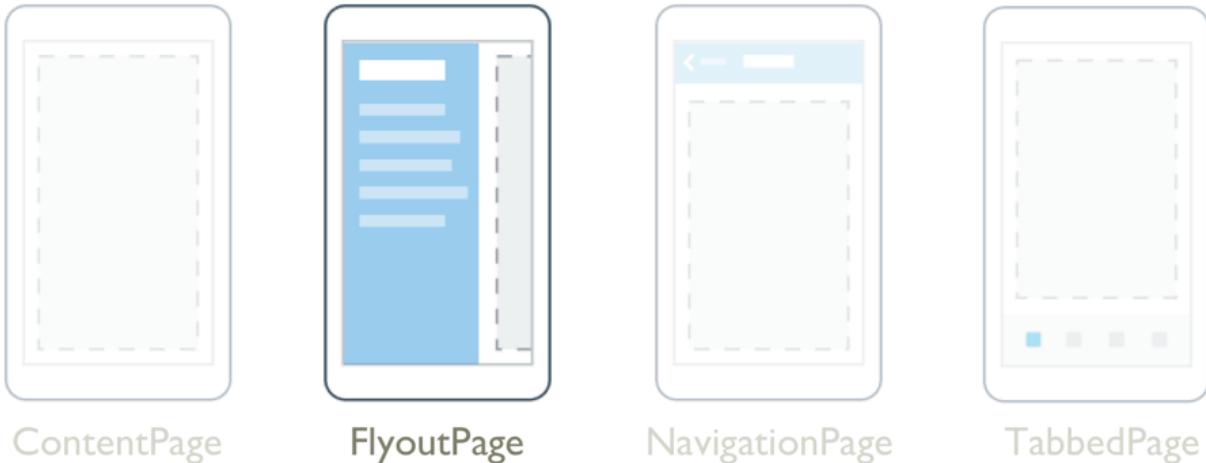
The child of a `ContentPage` is typically a layout, such as `Grid` or `StackLayout`, with the layout typically containing multiple views. However, the child of the `ContentPage` can be a view that displays a collection, such as `CollectionView`.

NOTE

The value of the `Title` property will be shown on the navigation bar, when the app performs navigation using a `NavigationPage`. For more information, see [NavigationPage](#).

FlyoutPage

9/20/2022 • 4 minutes to read • [Edit Online](#)



The .NET Multi-platform App UI (.NET MAUI) `FlyoutPage` is a page that manages two related pages of information – a flyout page that presents items, and a detail page that presents details about items on the flyout page. Selecting an item on the flyout page will navigate to the corresponding detail page.

`FlyoutPage` defines the following properties:

- `Detail`, of type `Page`, defines the detail page displayed for the selected item in the flyout page.
- `Flyout`, of type `Page`, defines the flyout page.
- `FlyoutLayoutBehavior`, of type `FlyoutLayoutBehavior`, indicates the layout behavior of flyout and detail pages.
- `IsGestureEnabled`, of type `bool`, determines whether a swipe gesture will switch between flyout and detail pages. The default value of this property is `true`.
- `IsPresented`, of type `bool`, determines whether the flyout or detail page is displayed. The default value of this property is `false`, which displays the detail page. It should be set to `true` to display the flyout page.

The `IsGestureEnabled`, `IsPresented`, and `FlyoutLayoutBehavior` properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

`FlyoutPage` also defines an `IsPresentedChanged` event, that's raised when the `IsPresented` property changes value.

WARNING

`FlyoutPage` is incompatible with .NET MAUI Shell apps, and an exception will be thrown if you attempt to use `FlyoutPage` in a Shell app. For more information about Shell apps, see [Shell](#).

Create a FlyoutPage

To create a flyout page, create a `FlyoutPage` object and set its `Flyout` and `Detail` properties. The `Flyout` property should be set to `ContentPage` object, and the `Detail` property should be set to a `TabbedPage`, `NavigationPage`, or `ContentPage` object. This will help to ensure a consistent user experience across all platforms.

IMPORTANT

A `FlyoutPage` is designed to be the root page of an app, and using it as a child page in other page types could result in unexpected and inconsistent behavior.

The following example shows a `FlyoutPage` that sets the `Flyout` and `Detail` properties:

```
<FlyoutPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace: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>
```

In this example, the `Flyout` property is set to a `ContentPage` object, and the `Detail` property is set to a `NavigationPage` containing a `ContentPage` object.

The following example shows the definition of the `FlyoutMenuPage` object, which is of type `ContentPage`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:FlyoutPageNavigation"
    x:Class="FlyoutPageNavigation.FlyoutMenuPage"
    Padding="0,40,0,0"
    IconImageSource="hamburger.png"
    Title="Personal Organiser">
    <CollectionView x:Name="collectionView"
        x:FieldModifier="public"
        SelectionMode="Single">
        <CollectionView.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>
        </CollectionView.ItemsSource>
        <CollectionView.ItemTemplate>
            <DataTemplate>
                <Grid Padding="5,10">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="30"/>
                        <ColumnDefinition Width="*" />
                    </Grid.ColumnDefinitions>
                    <Image Source="{Binding IconSource}" />
                    <Label Grid.Column="1"
                        Margin="20,0"
                        Text="{Binding Title}"
                        FontSize="20"
                        FontAttributes="Bold"
                        VerticalOptions="Center" />
                </Grid>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</ContentPage>

```

In this example, the flyout page consists of a `CollectionView` that's populated with data by setting its `ItemsSource` property to an array of `FlyoutPageItem` objects. The following example shows the definition of the `FlyoutPageItem` class:

```

public class FlyoutPageItem
{
    public string Title { get; set; }
    public string IconSource { get; set; }
    public Type TargetType { get; set; }
}

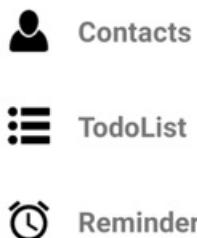
```

A `DataTemplate` is assigned to the `collectionView.ItemTemplate` property, to display each `FlyoutPageItem`. The `DataTemplate` contains a `Grid` 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`. In addition, the flyout 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.

NOTE

The `Flyout` page must have its `Title` property set, or an exception will occur.

The following screenshot shows the resulting flyout:



Create and display the detail page

The `FlyoutMenuPage` object contains a `CollectionView` that's referenced from the `MainPage` class. This allows the `MainPage` class to register a handler for the `SelectionChanged` event. This enables the `MainPage` object to set the `Detail` property to the page that represents the selected `CollectionView` item. The following example shows the event handler:

```
public partial class MainPage : FlyoutPage
{
    public MainPage()
    {
        ...
        flyoutPage.collectionView.SelectionChanged += OnSelectionChanged;
    }

    void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        var item = e.CurrentSelection.FirstOrDefault() as FlyoutPageItem;
        if (item != null)
        {
            Detail = new NavigationPage((Page)Activator.CreateInstance(item.TargetType));
            IsPresented = false;
        }
    }
}
```

In this example, the `OnSelectionChanged` event handler retrieves the `CurrentSelection` from the `CollectionView` object and sets the detail page to an instance of the page type stored in the `TargetType` property of the `FlyoutPageItem`. The detail page is displayed by setting the `FlyoutPage.IsPresented` property to `false`.

Control detail page layout behavior

How the `FlyoutPage` displays the flyout and detail pages depends on the form factor of the device the app is running on, the orientation of the device, and the value of the `FlyoutLayoutBehavior` property. This property should be set to a value of the `FlyoutLayoutBehavior` enumeration, which defines the following members:

- `Default` – 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 example shows how to set the `FlyoutLayoutBehavior` property on a `FlyoutPage`:

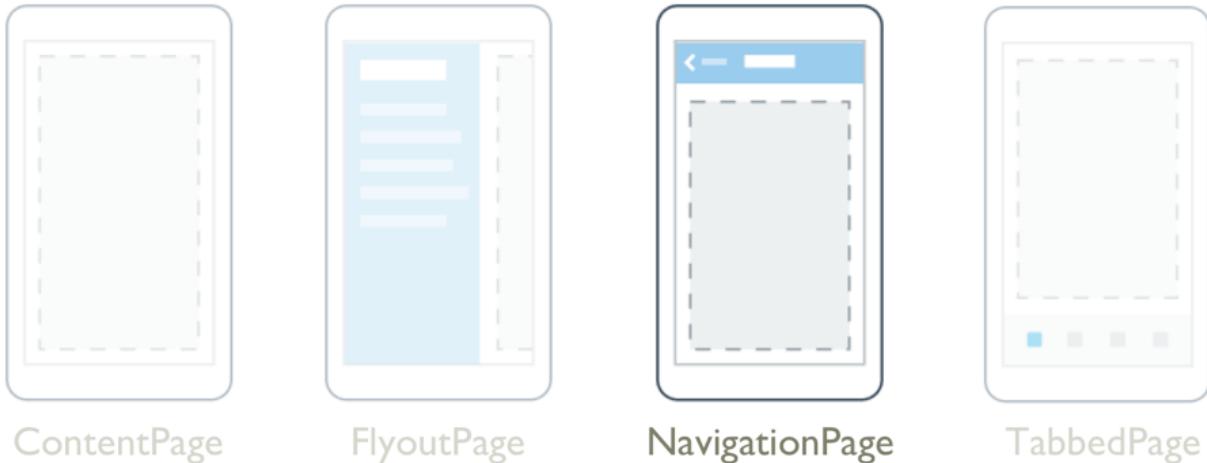
```
<FlyoutPage ...  
    FlyoutLayoutBehavior="Split">  
    ...  
</FlyoutPage>
```

IMPORTANT

The value of the `FlyoutLayoutBehavior` property only affects apps running on tablets or the desktop. Apps running on phones always have the `Popover` behavior.

NavigationPage

9/20/2022 • 10 minutes to read • [Edit Online](#)



The .NET Multi-platform App UI (.NET MAUI) `NavigationPage` provides a hierarchical navigation experience where you're able to navigate through pages, forwards and backwards, as desired. `NavigationPage` provides navigation as a last-in, first-out (LIFO) stack of `Page` objects.

`NavigationPage` defines the following properties:

- `BarBackground`, of type `Brush`, specifies the background of the navigation bar as a `Brush`.
- `BarBackgroundColor`, of type `Color`, specifies the background color of the navigation bar.
- `BackButtonTitle`, of type `string`, represents the text to use for the back button. This is an attached property.
- `BarTextColor`, of type `Color`, specifies the color of the text on the navigation bar.
- `CurrentPage`, of type `Page`, represents the page that's on top of the navigation stack. This is a read-only property.
- `HasNavigationBar`, of type `bool`, represents whether a navigation bar is present on the `NavigationPage`. The default value of this property is `true`. This is an attached property.
- `HasBackButton`, of type `bool`, represents whether the navigation bar includes a back button. The default value of this property is `true`. This is an attached property.
- `IconColor`, of type `Color`, defines the background color of the icon in the navigation bar. This is an attached property.
- `RootPage`, of type `Page`, represents the root page of the navigation stack. This is a read-only property.
- `TitleIconImageSource`, of type `ImageSource`, defines the icon that represents the title on the navigation bar. This is an attached property.
- `TitleView`, of type `View`, defines the view that can be displayed in the navigation bar. This is an attached property.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `NavigationPage` class also defines three events:

- `Pushed` is raised when a page is pushed onto the navigation stack.
- `Popped` is raised when a page is popped from the navigation stack.
- `PoppedToRoot` is raised when the last non-root page is popped from the navigation stack.

All three events receive `NavigationEventArgs` objects that define a read-only `Page` property, which retrieves the page that was popped from the navigation stack, or the newly visible page on the stack.

WARNING

`NavigationPage` is incompatible with .NET MAUI Shell apps, and an exception will be thrown if you attempt to use `NavigationPage` in a Shell app. For more information about Shell apps, see [Shell](#).

Perform modeless navigation

.NET MAUI supports modeless page navigation. A modeless page stays on screen and remains available until you navigate to another page.

A `NavigationPage` is typically used to navigate through a stack of `ContentPage` objects. When one page navigates to another, the new page is pushed on the stack and becomes the active page:



When the second page returns back to the first page, a page is popped from the stack, and the new topmost page then becomes active:



A `NavigationPage` consists of a navigation bar, with the active page being displayed below the navigation bar. The following diagram shows the main components of the navigation bar:



An optional icon can be displayed between the back button and the title.

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 stack, and to manipulate the stack.

TIP

It's recommended that a `NavigationPage` should only be populated with `ContentPage` objects.

Create the root page

An app that is structured around multiple pages always has a *root* page, which is the first page added to the navigation stack. This is accomplished by creating a `NavigationPage` object whose constructor argument is the

root page of the app, and setting the resulting object as the value of the `App.MainPage` property:

```
public partial class App : Application
{
    public App()
    {
        InitializeComponent();
        MainPage = new NavigationPage(new MainPage());
    }
}
```

NOTE

The `RootPage` property of a `NavigationPage` provides access to the first page in the navigation stack.

Push pages to the navigation stack

A page can be navigated to by calling the `PushAsync` method on the `Navigation` property of the current page:

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

In this example, the `DetailsPage` object is pushed onto the navigation stack, where it becomes the active page.

NOTE

The `PushAsync` method has an override that includes a `bool` argument that specifies whether to display a page transition during navigation. The `PushAsync` method that lacks the `bool` argument enables the page transition by default.

Pop pages from the navigation stack

The active page can be popped from the navigation stack by pressing the *Back* button on a device, regardless of whether this is a physical button on the device or an on-screen button.

To programmatically return to the previous page, the `PopAsync` method should be called on the `Navigation` property of the current page:

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

In this example, the current page is removed from the navigation stack, with the new topmost page becoming the active page.

NOTE

The `PopAsync` method has an override that includes a `bool` argument that specifies whether to display a page transition during navigation. The `PopAsync` method that lacks the `bool` argument enables the page transition by default.

In addition, the `Navigation` property of each page also exposes a `PopToRootAsync` method that pops all but the root page off the navigation stack, therefore making the app's root page the active page.

Manipulate the navigation stack

The `Navigation` property of a `Page` exposes a `NavigationStack` property from which the pages in the

navigation stack can be obtained. While .NET MAUI 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:

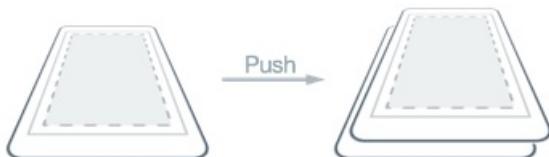


Together, these methods enable a custom navigation experience, such as replacing a login page with a new page following a successful login.

Perform modal navigation

.NET MAUI supports modal page navigation. A modal page encourages users to complete a self-contained task that cannot be navigated away from until the task is completed or cancelled.

A modal page can be any of the page types supported by .NET MAUI. To display a page modally, the app should push it onto the modal stack, where it will become the active page:



To return to the previous page the app should pop the current page from the modal stack, and the new topmost page becomes the active page:



Modal navigation methods are exposed by the `Navigation` property on any `Page` derived types. These methods provide the ability to push pages onto the modal stack, and pop pages from the modal stack. The `Navigation` property also exposes a `ModalStack` property from which 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` object is not required for performing modal page navigation.

Push pages to the modal stack

A page can be modally navigated to by calling the `PushModalAsync` method on the `Navigation` property of the current page:

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

In this example, the `DetailsPage` object is pushed onto the modal stack, where it becomes the active page.

NOTE

The `PushModalAsync` method has an override that includes a `bool` argument that specifies whether to display a page transition during navigation. The `PushModalAsync` method that lacks the `bool` argument enables the page transition by default.

Pop pages from the modal stack

The active page can be popped from the modal stack by pressing the *Back* button on a 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 `PopModalAsync` method should be called on the `Navigation` property of the current page:

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

In this example, the current page is removed from the modal stack, with the new topmost page becoming the active page.

NOTE

The `PopModalAsync` method has an override that includes a `bool` argument that specifies whether to display a page transition during navigation. The `PopModalAsync` method that lacks the `bool` argument enables the page transition by default.

Disable the back button

On Android, you can always return to the previous page by pressing the standard *Back* button on the device. If the modal page requires a self-contained task to be completed before leaving the page, the app must disable the *Back* button. This can be accomplished by overriding the `Page.OnBackPressed` method on the modal page.

Pass data during navigation

Sometimes it's necessary for a page to pass data to another page during navigation. Two standard techniques for accomplishing this are passing data through a page constructor, and by setting the new page's `BindingContext` to the data.

Pass data through a page constructor

The simplest technique for passing data to another page during navigation is through a page constructor argument:

```

Contact contact = new Contact
{
    Name = "Jane Doe",
    Age = 30,
    Occupation = "Developer",
    Country = "USA"
};

...
await Navigation.PushModalAsync(new DetailsPage(contact));

```

In this example, a `Contact` object is passed as a constructor argument to `DetailPage`. The `Contact` object can then be displayed by `DetailsPage`.

Pass 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:

```

Contact contact = new Contact
{
    Name = "Jane Doe",
    Age = 30,
    Occupation = "Developer",
    Country = "USA"
};

await Navigation.PushAsync(new DetailsPage
{
    BindingContext = contact
});

```

The advantage of passing navigation data via a page's `BindingContext` is that the new page can use data binding to display the data:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="MyMauiApp.DetailsPage"
    Title="Details">
    <StackLayout>
        <Label Text="{Binding Name}" />
        <Label Text="{Binding Occupation}" />
    </StackLayout>
</ContentPage>

```

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

Display views in the navigation bar

Any .NET MAUI `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 shows how to set the `NavigationPage.TitleView` attached property:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="NavigationPageTitleView.TitleViewPage">
    <NavigationPage.TitleView>
        <Slider HeightRequest="44"
            WidthRequest="300" />
    </NavigationPage.TitleView>
    ...
</ContentPage>
```

The equivalent C# code is:

```
Slider titleView = new Slider { HeightRequest = 44, WidthRequest = 300 };
NavigationPage.SetTitleView(this, titleView);
```

In this example, a `Slider` is displayed in the navigation bar of the `NavigationPage`, to control zooming.

IMPORTANT

Many views won't appear in the navigation bar unless the size of the view is specified with the `WidthRequest` and `HeightRequest` 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. However, this can result in clipping if the view displayed in the navigation bar is larger than the default size of the navigation bar. However, on Android, the height of the navigation bar can be changed by setting the `NavigationPage.BarHeight` bindable property to a `double` representing the new height.

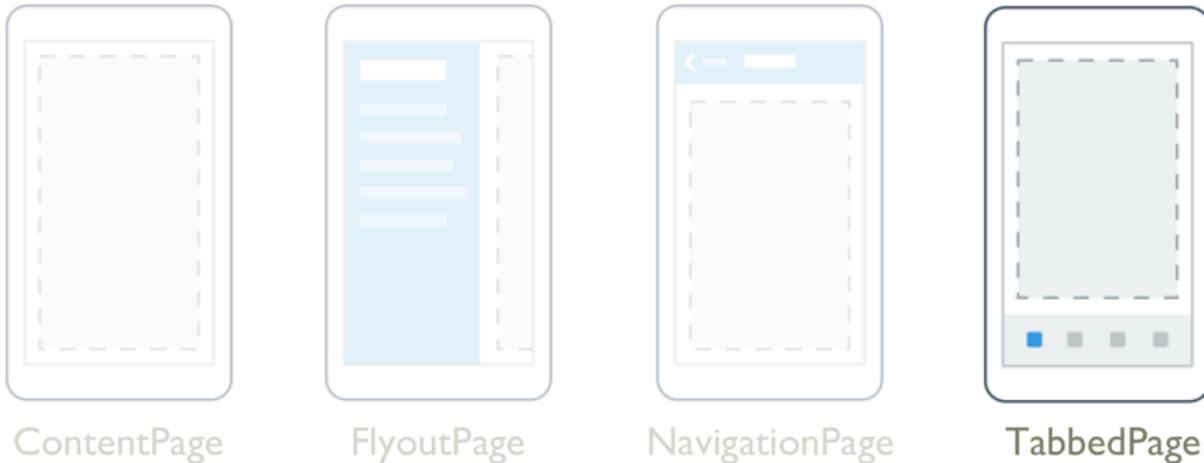
Alternatively, an extended navigation bar can be suggested by placing some of the content in the navigation bar, and some in a view at the top of the page content that you color match to the navigation bar. In addition, on iOS the separator line and shadow that's at the bottom of the navigation bar can be removed by setting the `NavigationPage.HideNavigationBarSeparator` bindable property to `true`.

TIP

The `BackButtonTitle`, `Title`, `TitleIconImageSource`, 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.

TabPage

9/20/2022 • 4 minutes to read • [Edit Online](#)



The .NET Multi-platform App UI (.NET MAUI) `TabPage` maintains a collection of children of type `Page`, only one of which is fully visible at a time. Each child is identified by a series of tabs across the top or bottom of the page. Typically, each child will be a `contentPage` and when its tab is selected the page content is displayed.

`TabPage` defines the following properties:

- `BarBackground`, of type `Brush`, defines the background of the tab bar.
- `BarBackgroundColor`, of type `Color`, defines the background color of the tab bar.
- `BarTextColor`, of type `Color`, represents the color of the text on the tab bar.
- `SelectedTabColor`, of type `Color`, indicates the color of a tab when it's selected.
- `UnselectedTabColor`, of type `Color`, represents the color of a tab when it's unselected.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

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 your app. However, .NET MAUI Shell enables pages accessed through a tab bar to be created on demand, in response to navigation. For more information about Shell apps, see [Shell](#).

WARNING

`TabPage` is incompatible with .NET MAUI Shell apps, and an exception will be thrown if you attempt to use `TabPage` in a Shell app.

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](#)

DataTemplate.

IMPORTANT

A `TabPage` should only be populated with `NavigationPage` and `ContentPage` objects.

Regardless of the approach taken, the location of the tab bar in a `TabPage` is platform-dependent:

- On iOS, the list of tabs appears at the bottom of the screen, and the page content is above. Each tab consists of a title and an icon. 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.
- On Android, the list of tabs appears at the top of the screen, and the page content is below. Each tab consists of a title and an icon. 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 moving the tabs to the bottom of the screen, see [TabPage toolbar placement on Android](#).
- On Windows, the list of tabs appears at the top of the screen, and the page content is below. Each tab consists of a title.

Populate a TabbedPage with a Page collection

A `TabPage` can be populated with a collection of child `Page` objects, which will typically be `ContentPage` objects. This is achieved by adding `ContentPage` objects as children of the `TabPage`:

```
<TabbedPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:TabPageWithNavigationPage"
    x:Class="TabPageWithNavigationPage.MainPage">
    <local:TodayPage />
    <local:SchedulePage />
    <local:SettingsPage />
</TabbedPage>
```

`Page` objects that are added as child elements of `TabPage` are added to the `Children` collection. The `Children` property of the `MultiPage<T>` class, from which `TabPage` 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 following screenshot shows the appearance of the resulting tab bar on the `TabPage`:



The page content for a tab appears when the tab is selected.

Populate a TabbedPage with a DataTemplate

`TabPage` inherits `ItemsSource`, `ItemTemplate`, and `SelectedItem` bindable properties from the `MultiPage<T>` class. These properties enable you to generate `TabPage` children dynamically, by setting the `ItemsSource` property to an `IEnumerable` collection of objects with public properties suitable for data bindings, and by setting the `ItemTemplate` property to a `DataTemplate` with a page type as the root element.

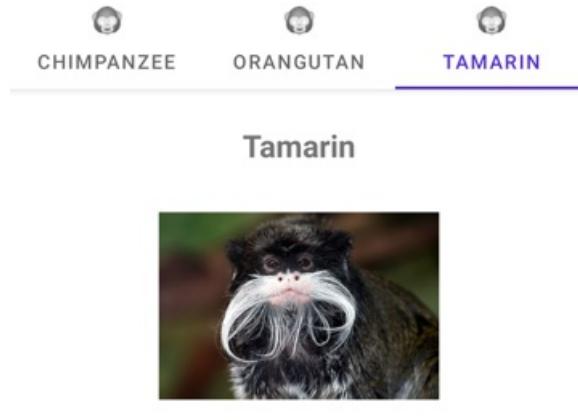
The following example shows generating `TabPage` children dynamically:

```

<TabbedPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:TabPageDemo"
    x:Class="TabPageDemo.MainPage"
    ItemsSource="{x:Static local:MonkeyDataModel.All}">
    <TabbedPage.ItemTemplate>
        <DataTemplate>
            <ContentPage Title="{Binding Name}"
                IconImageSource="monkeyicon.png">
                <StackLayout Padding="5, 25">
                    <Label Text="{Binding Name}"
                        FontAttributes="Bold"
                        FontSize="18"
                        HorizontalOptions="Center" />
                    <Image Source="{Binding PhotoUrl}"
                        HorizontalOptions="Center"
                        WidthRequest="200"
                        HeightRequest="200" />
                    <StackLayout Padding="50, 10">
                        <StackLayout Orientation="Horizontal">
                            <Label Text="Family: "
                                FontAttributes="Bold" />
                            <Label Text="{Binding Family}" />
                        </StackLayout>
                        ...
                    </StackLayout>
                </StackLayout>
            </ContentPage>
        </DataTemplate>
    </TabbedPage.ItemTemplate>
</TabbedPage>

```

In this example, each tab consists of a `ContentPage` object that uses `Image` and `Label` objects to display data for the tab:



Family: Callitrichidae
Genus: Saguinus

Navigate within a tab

Navigation can be performed within a tab, provided that the `ContentPage` object is wrapped in a `NavigationPage` object:

```
<TabbedPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:TabPageWithNavigationPage"
    x:Class="TabPageWithNavigationPage.MainPage">
    <local:TodayPage />
    <NavigationPage Title="Schedule"
        IconImageSource="schedule.png">
        <x:Arguments>
            <local:SchedulePage />
        </x:Arguments>
    </NavigationPage>
</TabbedPage>
```

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` object containing a `ContentPage` object.

When a `ContentPage` is wrapped in a `NavigationPage`, forwards page navigation can be performed by calling the `PushAsync` method on the `Navigation` property of the `ContentPage` object:

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

For more information about performing navigation using the `NavigationPage` class, see [NavigationPage](#).

WARNING

While a `NavigationPage` can be placed in a `TabbedPage`, it's not recommended to place a `TabbedPage` into a `NavigationPage`.

Host a Blazor web app in a .NET MAUI app using BlazorWebView

9/20/2022 • 3 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `BlazorWebView` is a control that enables you to host a Blazor web app in your .NET MAUI app. These apps, known as Blazor Hybrid apps, enable a Blazor web app to be integrated with platform features and UI controls. The `BlazorWebView` control can be added to any page of a .NET MAUI app, and pointed to the root of the Blazor app. The [Razor components](#) run natively in the .NET process and render web UI to an embedded web view control. In .NET MAUI, Blazor Hybrid apps can run on all the platforms supported by .NET MAUI.

`BlazorWebView` defines the following properties:

- `HostPage`, of type `string?`, which defines the root page of the Blazor web app.
- `RootComponents`, of type `RootComponentsCollection`, which specifies the collection of root components that can be added to the control.

The `RootComponent` class defines the following properties:

- `Selector`, of type `string?`, which defines the CSS selector string that specifies where in the document the component should be placed.
- `ComponentType`, of type `Type?`, which defines the type of the root component.
- `Parameters`, of type `IDictionary<string, object?>?`, which represents an optional dictionary of parameters to pass to the root component.

In addition, `BlazorWebView` defines the following events:

- `BlazorWebViewInitializing`, with an accompanying `BlazorWebViewInitializingEventArgs` object, which is raised before the `BlazorWebView` is initialized. This event enables customization of the `BlazorWebView` configuration.
- `BlazorWebViewInitialized`, with an accompanying `BlazorWebViewInitializedEventArgs` object, which is raised after the `BlazorWebView` is initialized but before any component has been rendered. This event enables retrieval of the platform-specific web view instance.
- `UrlLoading`, with an accompanying `UrlLoadingEventArgs` object, is raised when a hyperlink is clicked within a `BlazorWebView`. This event enables customization of whether a hyperlink is opened in the `BlazorWebView`, in an external app, or whether the URL loading attempt is cancelled.

Existing [Razor components](#) can be used in a .NET MAUI Blazor app by moving the code into the app, or by referencing an existing class library or package that contains the component.

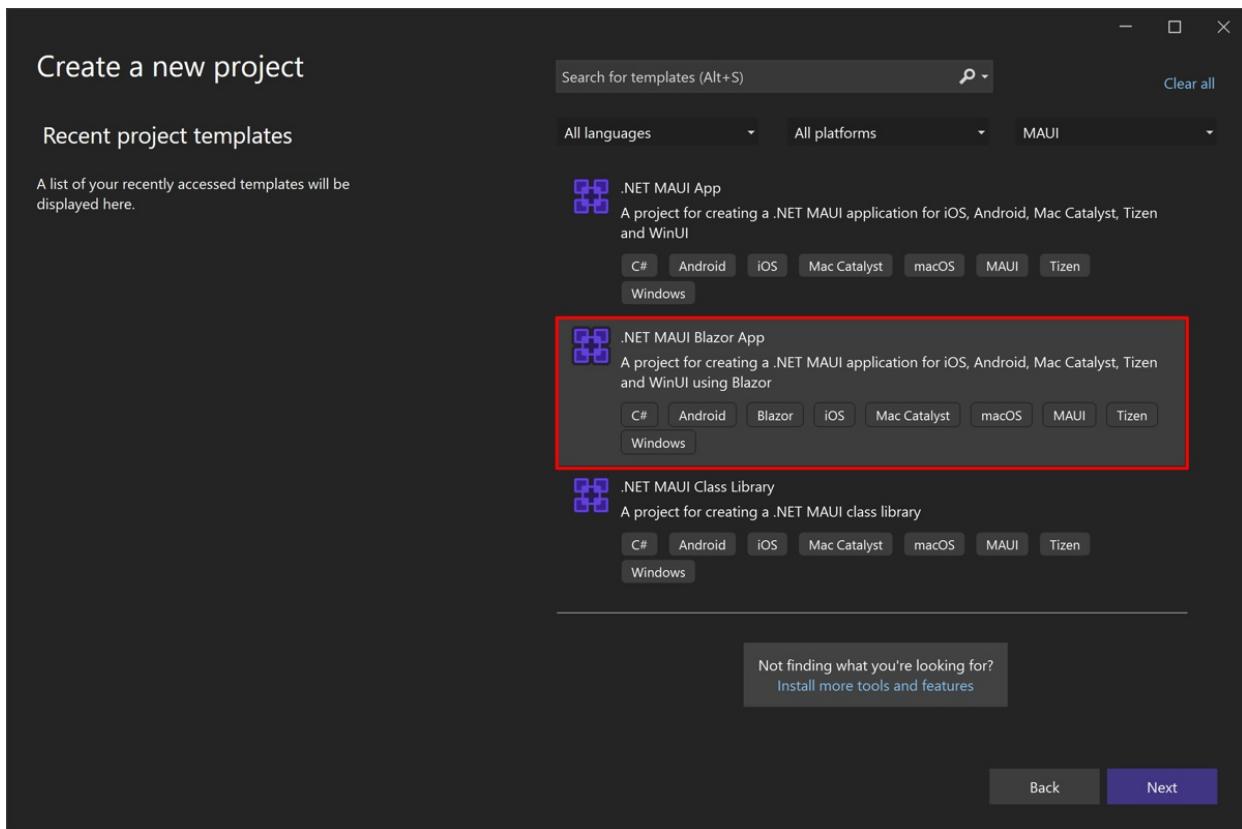
For more information about Blazor Hybrid apps, see [ASP.NET Core Blazor Hybrid](#).

NOTE

While Visual Studio installs all the required tooling to develop .NET MAUI Blazor apps, end users of .NET MAUI Blazor apps on Windows must install the [WebView2 runtime](#).

Create a .NET MAUI Blazor app

A .NET MAUI Blazor app can be created in Visual Studio by the **.NET MAUI Blazor app** template:



This project template creates a multi-targeted .NET MAUI Blazor app that can be deployed to Android, iOS, macOS, and Windows. For step-by-step instructions on creating a .NET MAUI Blazor app, see [Build a .NET MAUI Blazor app](#).

The `BlazorWebView` created by the project template is defined in *MainPage.xaml*, and points to the root of the Blazor app:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:BlazorWebViewDemo"
    x:Class="BlazorWebViewDemo.MainPage"
    BackgroundColor="{DynamicResource PageBackgroundColor}">

    <BlazorWebView HostPage="wwwroot/index.html">
        <BlazorWebView.RootComponents>
            <RootComponent Selector="#app" ComponentType="{x:Type local:Main}" />
        </BlazorWebView.RootComponents>
    </BlazorWebView>

</ContentPage>
```

The root **Razor component** for the app is in *Main.razor*, which Razor compiles into a type named `Main` in the application's root namespace. The rest of the **Razor components** are in the *Pages* and *Shared* project folders, and are identical to the components used in the default Blazor web template. Static web assets for the app are in the *wwwroot* folder.

Add a BlazorWebView to an existing app

The process to add a `BlazorWebView` to an existing .NET MAUI app is as follows:

1. Add the Razor SDK, `Microsoft.NET.Sdk.Razor` to your project by editing its first line of the CSPROJ project file:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
```

The Razor SDK is required to build and package projects containing Razor files for Blazor projects.

2. Add the root [Razor component](#) for the app to the project.
3. Add your [Razor components](#) to project folders named *Pages* and *Shared*.
4. Add your static web assets to a project folder named *wwwroot*.
5. Add any optional *_Imports.razor* files to your project.
6. Add a [BlazorWebView](#) to a page in your .NET MAUI app, and point it to the root of the Blazor app:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              xmlns:local="clr-namespace:MyBlazorApp"
              x:Class="MyBlazorApp.MainPage">

    <BlazorWebView HostPage="wwwroot/index.html">
        <BlazorWebView.RootComponents>
            <RootComponent Selector="app" ComponentType="{x:Type local:Main}" />
        </BlazorWebView.RootComponents>
    </BlazorWebView>

</ContentPage>
```

7. Modify the [createMauiApp](#) method of your [MauiProgram](#) class to register the [BlazorWebView](#) control for use in your app. To do this, on the [IServiceCollection](#) object, call the [AddMauiBlazorWebView](#) method to add component web view services to the services collection:

```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts =>
            {
                fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            });

        builder.Services.AddMauiBlazorWebView();
#if DEBUG
        builder.Services.AddMauiBlazorWebViewDeveloperTools();
#endif
        // Register any app services on the IServiceCollection object
        // e.g. builder.Services.AddSingleton<WeatherForecastService>();

        return builder.Build();
    }
}
```

Border

9/20/2022 • 4 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Border` is a container control that draws a border, background, or both, around another control. A `Border` can only contain one child object. If you want to put a border around multiple objects, wrap them in a container object such as a layout. For more information about layouts, see [Layouts](#).

`Border` defines the following properties:

- `Content`, of type `IView`, represents the content to display in the border. This property is the `ContentProperty` of the `Border` class, and therefore does not need to be explicitly set from XAML.
- `Padding`, of type `Thickness`, represents the distance between the border and its child element.
- `StrokeShape`, of type `IShape`, describes the shape of the border. This property has a type converter applied to it that can convert a string to its equivalent `IShape`.
- `Stroke`, of type `Brush`, indicates the brush used to paint the border.
- `StrokeThickness`, of type `double`, indicates the width of the border. The default value of this property is 1.0.
- `StrokeDashArray`, of type `DoubleCollection`, which represents a collection of `double` values that indicate the pattern of dashes and gaps that make up the border.
- `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 its line. 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 the stroke 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 stroke thickness. The default value of this property is 10.0.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

IMPORTANT

When creating a border using a shape, such as a `Rectangle` or `Polygon`, only closed shapes should be used. Therefore, open shapes such as `Line` are unsupported.

For more information about the properties that control the shape and stroke of the border, see [Shapes](#).

Create a Border

To draw a border, create a `Border` object and set its properties to define its appearance. Then, set its child to the control to which the border should be added.

The following XAML example shows how to draw a border around a `Label`:

```

<Border Stroke="#C49B33"
        StrokeThickness="4"
        StrokeShape="RoundRectangle 40,0,0,40"
        Background="#2B0B98"
        Padding="16,8"
        HorizontalOptions="Center">
    <Label Text=".NET MAUI"
        TextColor="White"
        FontSize="18"
        FontAttributes="Bold" />
</Border>

```

Alternatively, the `StrokeShape` property value can be specified using property tag syntax:

```

<Border Stroke="#C49B33"
        StrokeThickness="4"
        Background="#2B0B98"
        Padding="16,8"
        HorizontalOptions="Center">
    <Border.StrokeShape>
        <RoundRectangle CornerRadius="40,0,0,40" />
    </Border.StrokeShape>
    <Label Text=".NET MAUI"
        TextColor="White"
        FontSize="18"
        FontAttributes="Bold" />
</Border>

```

The equivalent C# code is:

```

using Microsoft.Maui.Controls.Shapes;
using GradientStop = Microsoft.Maui.Controls.GradientStop;
...

Border border = new Border
{
    Stroke = Color.FromArgb("#C49B33"),
    Background = Color.FromArgb("#2B0B98"),
    StrokeThickness = 4,
    Padding = new Thickness(16, 8),
    HorizontalOptions = LayoutOptions.Center,
    StrokeShape = new RoundRectangle
    {
        CornerRadius = new CornerRadius(40, 0, 0, 40)
    },
    Content = new Label
    {
        Text = ".NET MAUI",
        TextColor = Colors.White,
        FontSize = 18,
        FontAttributes = FontAttributes.Bold
    }
};

```

In this example, a border with rounded top-left and bottom-right corners is drawn around a `Label`. The border shape is defined as a `RoundRectangle` object, whose `CornerRadius` property is set to a `Thickness` value that enables independent control of each corner of the rectangle:

.NET MAUI

Because the `Stroke` property is of type `Brush`, borders can also be drawn using gradients:

```
<Border StrokeThickness="4"
        StrokeShape="RoundRectangle 40,0,0,40"
        Background="#2B0B98"
        Padding="16,8"
        HorizontalOptions="Center">
    <Border.Stroke>
        <LinearGradientBrush EndPoint="0,1">
            <GradientStop Color="Orange"
                Offset="0.1" />
            <GradientStop Color="Brown"
                Offset="1.0" />
        </LinearGradientBrush>
    </Border.Stroke>
    <Label Text=".NET MAUI"
        TextColor="White"
        FontSize="18"
        FontAttributes="Bold" />
</Border>
```

The equivalent C# code is:

```
using Microsoft.Maui.Controls.Shapes;
using GradientStop = Microsoft.Maui.Controls.GradientStop;
...

Border gradientBorder = new Border
{
    StrokeThickness = 4,
    Background = Color.FromArgb("#2B0B98"),
    Padding = new Thickness(16, 8),
    HorizontalOptions = LayoutOptions.Center,
    StrokeShape = new RoundRectangle
    {
        CornerRadius = new CornerRadius(40, 0, 0, 40)
    },
    Stroke = new LinearGradientBrush
    {
        EndPoint = new Point(0, 1),
        GradientStops = new GradientStopCollection
        {
            new GradientStop { Color = Colors.Orange, Offset = 0.1f },
            new GradientStop { Color = Colors.Brown, Offset = 1.0f }
        },
    },
    Content = new Label
    {
        Text = ".NET MAUI",
        TextColor = Colors.White,
        FontSize = 18,
        FontAttributes = FontAttributes.Bold
    }
};
```

In this example, a border that uses a linear gradient is drawn around a `Label`:

Define the border shape with a string

In XAML, the value of the `StrokeShape` property can be defined using property-tag syntax, or as a `string`. Valid `string` values for the `StrokeShape` property are:

- `Ellipse`
- `Line`, followed by one or two x- and y-coordinate pairs. For example, `Line 10 20` draws a line from (10,20) to (0,0), and `Line 10 20, 100 120` draws a line from (10,20) to (100,120).
- `Path`, followed by path markup syntax data. For example, `Path M 10,100 L 100,100 100,50Z` will draw a triangular border. For more information about path markup syntax, see [Path markup syntax](#).
- `Polygon`, followed by a collection of x- and y-coordinate pairs. For example, `Polygon 40 10, 70 80, 10 50`.
- `Polyline`, followed by a collection x- and y-coordinate pairs. For example,
`Polyline 0,0 10,30 15,0 18,60 23,30 35,30 40,0 43,60 48,30 100,30`.
- `Rectangle`
- `RoundRectangle`, optionally followed by a corner radius. For example, `RoundRectangle 40` or
`RoundRectangle 40,0,0,40`.

IMPORTANT

While `Line` is a valid `string` value for the `StrokeShape` property, its use is not supported.

`String`-based x- and y-coordinate pairs can be 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. Coordinate pairs will be converted to `Point` objects that define `x` and `y` properties, of type `double`.

BoxView

9/20/2022 • 2 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `BoxView` draws a simple rectangle or square, of a specified width, height, and color.

`BoxView` defines the following properties:

- `Color`, of type `Color`, which defines the color of the `BoxView`.
- `CornerRadius`, of type `CornerRadius`, which defines the corner radius of the `BoxView`. This 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`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

NOTE

Although `BoxView` can mimic simple graphics, a better alternative is to use .NET MAUI Shapes or [.NET MAUI Graphics](#).

Create a BoxView

To draw a rectangle or square, create a `BoxView` object and set its `Color`, `WidthRequest`, and `HeightRequest` properties. Optionally, you can also set its `CornerRadius` property.

The following XAML example shows how to create a `BoxView`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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>
```

In this example, a cornflower blue `BoxView` is displayed in the center of the page:



The `WidthRequest` and `HeightRequest` properties are measured in device-independent units.

NOTE

A `BoxView` can also be a child of an `AbsoluteLayout`. In this case, both the location and size of the `BoxView` are set using the `LayoutBounds` attached bindable property.

A `BoxView` can also be sized to resemble a line of a specific width and thickness.

Frame

9/20/2022 • 2 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Frame` is used to wrap a view or layout with a border that can be configured with color, shadow, and other options. Frames can be used to create borders around controls but can also be used to create more complex UI.

The `Frame` class defines the following properties:

- `BorderColor`, of type `color`, determines the color of the `Frame` border.
- `CornerRadius`, of type `float`, determines the rounded radius of the corner.
- `HasShadow`, of type `bool`, determines whether the frame has a drop shadow.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `Frame` class inherits from `ContentView`, which provides a `Content` bindable property. The `Content` property is the `ContentProperty` of the `Frame` class, and therefore does not need to be explicitly set from XAML.

NOTE

The `Frame` class existed in Xamarin.Forms and is present in .NET MAUI for users who are migrating their apps from Xamarin.Forms to .NET MAUI. If you're building a new .NET MAUI app it's recommended to use `Border` instead, and to set shadows using the `Shadow` bindable property on `VisualElement`. For more information, see [Border](#) and [Shadow](#).

Create a Frame

A `Frame` object typically wraps another control, such as a `Label`:

```
<Frame>
    <Label Text="Frame wrapped around a Label" />
</Frame>
```

The appearance of `Frame` objects can be customized by setting properties:

```
<Frame BorderColor="Gray"
       CornerRadius="10">
    <Label Text="Frame wrapped around a Label" />
</Frame>
```

The equivalent C# code is:

```
Frame frame = new Frame
{
    BorderColor = Colors.Gray
    CornerRadius = 10,
    Content = new Label { Text = "Frame wrapped around a Label" }
};
```

The following screenshot shows the example `Frame`:

Frame wrapped around a Label

Create a card with a Frame

Combining a `Frame` object with a layout such as a `StackLayout` enables the creation of more complex UI.

The following XAML shows how to create a card with a `Frame`:

```
<Frame BorderColor="Gray"
       CornerRadius="5"
       Padding="8">
    <StackLayout>
        <Label Text="Card Example"
              FontSize="14"
              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>
```

The following screenshot shows the example card:

Card Example

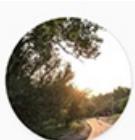
Frames can wrap more complex layouts to create more complex UI components, such as this card!

Round elements

The `CornerRadius` property of the `Frame` control is one approach to creating a circle image. The following XAML shows how to create a circle image with a `Frame`:

```
<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>
```

The following screenshot shows the example circle image:



GraphicsView

9/20/2022 • 3 minutes to read • [Edit Online](#)

{ } [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `GraphicsView` is a graphics canvas on which 2D graphics can be drawn using types from the `Microsoft.Maui.Graphics` namespace. For more information about `Microsoft.Maui.Graphics`, see [Graphics](#).

`GraphicsView` defines the `Drawable` property, of type `IDrawable`, which specifies the content that will be drawn. This property is backed by a `BindableProperty`, which means it can be the target of data binding, and styled.

`GraphicsView` defines the following events:

- `StartHoverInteraction`, with `TouchEventArgs`, which is raised when a pointer enters the hit test area of the `GraphicsView`.
- `MoveHoverInteraction`, with `TouchEventArgs`, which is raised when a pointer moves while the pointer remains within the hit test area of the `GraphicsView`.
- `EndHoverInteraction`, which is raised when a pointer leaves the hit test area of the `GraphicsView`.
- `StartInteraction`, with `TouchEventArgs`, which is raised when the `GraphicsView` is pressed.
- `DragInteraction`, with `TouchEventArgs`, which is raised when the `GraphicsView` is dragged.
- `EndInteraction`, with `TouchEventArgs`, which is raised when the press that raised the `StartInteraction` event is released.
- `CancelInteraction`, which is raised when the press that made contact with the `GraphicsView` loses contact.

Create a GraphicsView

A `GraphicsView` must define an `IDrawable` object that specifies the content that will be drawn on the control. This can be achieved by creating an object that derives from `IDrawable`, and by implementing its `Draw` method:

```
namespace MyMauiApp
{
    public class GraphicsDrawable : IDrawable
    {
        public void Draw(ICanvas canvas, RectF dirtyRect)
        {
            // Drawing code goes here
        }
    }
}
```

The `Draw` method has `ICanvas` and `RectF` arguments. The `ICanvas` argument is the drawing canvas on which you draw graphical objects. The `RectF` argument is a `struct` that contains data about the size and location of the drawing canvas. For more information about drawing on an `ICanvas`, see [Draw graphical objects](#).

In XAML, the `IDrawable` object should be declared as a resource, and then consumed by a `GraphicsView` by specifying its key:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:drawable="clr-namespace:MyMauiApp"
    x:Class="MyMauiApp.MainPage">
    <ContentPage.Resources>
        <drawable:GraphicsDrawable x:Key="drawable" />
    </ContentPage.Resources>
    <VerticalStackLayout>
        <GraphicsView Drawable="{StaticResource drawable}"
            HeightRequest="300"
            WidthRequest="400" />
    </VerticalStackLayout>
</ContentPage>

```

Position and size graphical objects

The location and size of the `ICanvas` on a page can be determined by examining properties of the `RectF` argument in the `Draw` method.

The `RectF` struct defines the following properties:

- `Bottom`, of type `float`, which represents the y-coordinate of the bottom edge of the canvas.
- `Center`, of type `PointF`, which specifies the coordinates of the center of the canvas.
- `Height`, of type `float`, which defines the height of the canvas.
- `IsEmpty`, of type `bool`, which indicates whether the canvas has a zero size and location.
- `Left`, of type `float`, which represents the x-coordinate of the left edge of the canvas.
- `Location`, of type `PointF`, which defines the coordinates of the upper-left corner of the canvas.
- `Right`, of type `float`, which represents the x-coordinate of the right edge of the canvas.
- `Size`, of type `SizeF`, which defines the width and height of the canvas.
- `Top`, of type `float`, which represents the y-coordinate of the top edge of the canvas.
- `Width`, of type `float`, which defines the width of the canvas.
- `X`, of type `float`, which defines the x-coordinate of the upper-left corner of the canvas.
- `Y`, of type `float`, which defines the y-coordinate of the upper-left corner of the canvas.

These properties can be used to position and size graphical objects on the `ICanvas`. For example, graphical objects can be placed at the center of the `Canvas` by using the `Center.X` and `Center.Y` values as arguments to a drawing method. For information about drawing on an `ICanvas`, see [Draw graphical objects](#).

Invalidate the canvas

`GraphicsView` has an `Invalidate` method that informs the canvas that it needs to redraw itself. This method must be invoked on a `GraphicsView` instance:

```
graphicsView.Invalidate();
```

.NET MAUI automatically invalidates the `GraphicsView` as needed by the UI. For example, when the element is first shown, comes into view, or is revealed by moving an element from on top of it, it's redrawn. The only time you need to call `Invalidate` is when you want to force the `GraphicsView` to redraw itself, such as if you have changed its content while it's still visible.

Image

9/20/2022 • 7 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Image` displays an image that can be loaded from a local file, a URI, an embedded resource, or a stream. The standard platform image formats are supported, including animated GIFs, and local Scalable Vector Graphics (SVG) files are also supported. For more information about adding images to a .NET MAUI app project, see [Add images to a .NET MAUI app project](#).

`Image` defines the following properties:

- `Aspect`, of type `Aspect`, defines the scaling mode of the image.
- `IsAnimationPlaying`, of type `bool`, determines whether an animated GIF is playing or stopped. The default value of this property is `false`.
- `IsLoading`, of type `bool`, indicates the loading status of the image. The default value of this property is `false`.
- `IsOpaque`, of type `bool`, indicates whether the rendering engine may treat the image as opaque while rendering it. The default value of this property is `false`.
- `Source`, of type `ImageSource`, specifies the source of the image.

These properties are backed by `BindableProperty` objects, which means that they can be styled, and be the target of data bindings.

NOTE

Font icons can be displayed by an `Image` by specifying the font icon data as a `FontImageSource` object. For more information, see [Display font icons](#).

The `ImageSource` class defines the following methods that can be used to load an image from different sources:

- `FromFile` returns a `FileImageSource` that reads an image from a local file.
- `FromUri` returns an `UriImageSource` that downloads and reads an image from a specified URI.
- `FromResource` returns a `StreamImageSource` that reads an image file embedded in an assembly.
- `FromStream` returns a `StreamImageSource` that reads an image from a stream that supplies image data.

In XAML, images can be loaded from files and URIs by specifying the filename or URI as a string value for the `Source` property. Images can also be loaded from resources or streams in XAML through custom markup extensions.

IMPORTANT

Images will be displayed at their full resolution unless the size of the `Image` is constrained by its layout, or the `HeightRequest` or `WidthRequest` property of the `Image` is specified.

For information about adding app icons and a splash screen to your app, see [App icons](#) and [Splash screen](#).

Load a local image

Images can be added to your app project by dragging them to the `Resources/Images` folder of your project, where its build action will automatically be set to `MauiImage`. At build time, vector images are resized to the

correct resolutions for the target platform and device, and added to your app package. This is necessary because different platforms support different image resolutions, and the operating system chooses the appropriate image resolution at runtime based on the device's capabilities.

To comply with Android resource naming rules, all local image filenames must be lowercase, start and end with a letter character, and contain only alphanumeric characters or underscores. For more information, see [App resources overview](#) on developer.android.com.

IMPORTANT

.NET MAUI converts SVG files to PNG files. Therefore, when adding an SVG file to your .NET MAUI app project, it should be referenced from XAML or C# with a .png extension.

Adhering to these rules for file naming and placement enables the following XAML to load and display an image:

```
<Image Source="dotnet_bot.png" />
```

The equivalent C# code is:

```
Image image = new Image
{
    Source = ImageSource.FromFile("dotnet_bot.png")
};
```

The `ImageSourceFromFile` method requires a `string` argument, and returns a new `FileImageSource` object that reads the image from the file. There's also an implicit conversion operator that enables the filename to be specified as a `string` argument to the `Image.Source` property:

```
Image image = new Image { Source = "dotnet_bot.png" };
```

Load a remote image

Remote images can be downloaded and displayed by specifying a URI as the value of the `Source` property:

```
<Image Source="https://aka.ms/campus.jpg" />
```

The equivalent C# code is:

```
Image image = new Image
{
    Source = ImageSource.FromUri(new Uri("https://aka.ms/campus.jpg"))
};
```

The `ImageSource.FromUri` method requires a `Uri` argument, and returns a new `UriImageSource` object that reads the image from the `Uri`. There's also an implicit conversion for string-based URIs:

```
Image image = new Image { Source = "https://aka.ms/campus.jpg" };
```

Image caching

Caching of downloaded images is enabled by default, with cached images being stored for 1 day. This behavior

can be changed by setting properties of the `UriImageSource` class.

The `UriImageSource` class defines the following properties:

- `Uri`, of type `Uri`, represents the URI of the image to be downloaded for display.
- `CacheValidity`, of type `TimeSpan`, specifies how long the image will be stored locally for. The default value of this property is 1 day.
- `CachingEnabled`, of type `bool`, defines whether image caching is enabled. The default value of this property is `true`.

These properties are backed by `BindableProperty` objects, which means that they can be styled, and be the target of data bindings.

To set a specific cache period, set the `Source` property to an `UriImageSource` object that sets its `CacheValidity` property:

```
<Image>
    <Image.Source>
        <UriImageSource Uri="https://aka.ms/campus.jpg"
                        CacheValidity="10:00:00.0" />
    </Image.Source>
</Image>
```

The equivalent C# code is:

```
Image image = new Image();
image.Source = new UriImageSource
{
    Uri = new Uri("https://aka.ms/campus.jpg"),
    CacheValidity = new TimeSpan(10,0,0,0)
};
```

In this example, the caching period is set to 10 days.

Load an embedded image

Embedded images can be added to an assembly as a resource by dragging them into your project, and ensuring their build action is set to **Embedded resource** in the **Properties window**.

Embedded images are loaded based on their resource ID, which is comprised of the name of the project and its location in the project. For example, placing `dotnet_bot.png` in the root folder of a project named `MyProject` will result in a resource ID of `MyProject.dotnet_bot.png`. Similarly, placing `dotnet_bot.png` in the `Assets` folder of a project named `MyProject` will result in a resource ID of `MyProject.Assets.dotnet_bot.png`.

The `ImageSource.FromResource` method can be used to load an image that's embedded into an assembly as a resource:

```
Image image = new Image
{
    Source = ImageSource.FromResource("MyProject.Assets.dotnet_bot.png")
};
```

By default, the `ImageSource.FromResource` method only looks for images in the same assembly as the calling code. However, the assembly containing the embedded image can be specified as the second argument to the `ImageSource.FromResource` method:

```

Image image = new Image
{
    Source = ImageSource.FromResource("MyLibrary.MyFolder.myimage.png",
typeof(MyLibrary.MyClass).GetTypeInfo().Assembly)
};

```

Load an embedded image in XAML

Embedded images can be loaded in XAML with a custom XAML markup extension:

```

using System.Reflection;
using System.Xml;

namespace ImageDemos
{
    [ContentProperty("Source")]
    public 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);
        }
    }
}

```

The `ImageResourceExtension` class is used to access an embedded image in XAML, and uses the `Source` property to call the `ImageSource.FromResource` method. The argument to the `ProvideValue` method is an `IServiceProvider` object that can be used to obtain an `IXmlLineInfoProvider` object that can provide line and character information indicating where an error has been detected. This object is used to raise an exception when the `Image.Source` property hasn't been set.

The markup extension can be consumed in XAML to load an embedded image:

```

<ContentPage ...
    xmlns:local="clr-namespace:ImageDemos">
    <StackLayout>
        <Image Source="{local:ImageResource monkey.png}"
            HeightRequest="100" />
    </StackLayout>
</ContentPage>

```

For more information about XAML markup extensions, see [Create XAML markup extensions](#).

Load an image from a stream

Images can be loaded from streams with the `ImageSource.FromStream` method:

```
Image image = new Image
{
    Source = ImageSource.FromStream(() => stream)
};
```

Load animated GIFs

.NET MAUI includes support for displaying small, animated GIFs. This is accomplished by setting the `Source` property to an animated GIF file:

```
<Image Source="demo.gif" />
```

IMPORTANT

While the animated GIF support in .NET MAUI 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`. Therefore, when an animated GIF is loaded it will not be played until the `IsAnimationPlaying` property is set to `true`. Playback can be stopped by resetting the `IsAnimationPlaying` property to `false`. Note that this property has no effect when displaying a non-GIF image source.

Control image scaling

The `Aspect` property determines how the image will be scaled to fit the display area, and should be set to one of the members of the `Aspect` enumeration:

- `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.
- `AspectFill` - clips the image so that it fills the display area while preserving the aspect ratio.
- `Fill` - stretches the image to completely and exactly fill the display area. This may result in the image being distorted.
- `Center` - centers the image in the display area while preserving the aspect ratio.

Label

9/20/2022 • 11 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `Label` displays single-line and multi-line text. Text displayed by a `Label` can be colored, spaced, and can have text decorations.

`Label` defines the following properties:

- `CharacterSpacing`, of type `double`, sets the spacing between characters in the displayed text.
- `FontAttributes`, of type `FontAttributes`, determines text style.
- `FontAutoScalingEnabled`, of type `bool`, defines whether the text will reflect scaling preferences set in the operating system. The default value of this property is `true`.
- `FontFamily`, of type `string`, defines the font family.
- `FontSize`, of type `double`, defines the font size.
- `FormattedText`, of type `FormattedString`, specifies the presentation of text with multiple presentation options such as fonts and colors.
- `HorizontalTextAlignment`, of type `TextAlignment`, defines the horizontal alignment of the displayed text.
- `LineBreakMode`, of type `LineBreakMode`, determines how text should be handled when it can't fit on one line.
- `LineHeight`, of type `double`, specifies the multiplier to apply to the default line height when displaying text.
- `MaxLines`, of type `int`, indicates the maximum number of lines allowed in the `Label`.
- `Padding`, of type `Thickness`, determines the label's padding.
- `Text`, of type `string`, defines the text displayed as the content of the label.
- `TextColor`, of type `Color`, defines the color of the displayed text.
- `TextDecorations`, of type `TextDecorations`, specifies the text decorations (underline and strikethrough) that can be applied.
- `TextTransform`, of type `TextTransform`, specifies the casing of the displayed text.
- `TextType`, of type `TextType`, determines whether the `Label` should display plain text or HTML text.
- `VerticalTextAlignment`, of type `TextAlignment`, defines the vertical alignment of the displayed text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

For information about specifying fonts on a `Label`, see [Fonts](#).

Create a Label

The following example shows how to create a `Label`:

```
<Label Text="Hello world" />
```

The equivalent C# code is:

```
Label label = new Label { Text = "Hello world" };
```

Set colors

Labels can be set to use a specific text color via the `TextColor` property.

The following example sets the text color of a `Label`:

```
<Label TextColor="#77d065"  
      Text="This is a green label." />
```

For more information about colors, see [Colors](#).

Set character spacing

Character spacing can be applied to `Label` objects by setting the `CharacterSpacing` property to a `double` value:

```
<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.

Add 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" };
```

Control text truncation and wrapping

Text wrapping and truncation can be controlled by setting the `LineBreakMode` property to a value of the `LineBreakMode` enumeration:

- `NoWrap` — does not wrap text, displaying only as much text as can fit on one line. This is the default value of the `LineBreakMode` property.
- `WordWrap` — wraps text at the word boundary.
- `CharacterWrap` — wraps text onto a new line at a character boundary.

- `HeadTruncation` — truncates the head of the text, showing the end.
- `MiddleTruncation` — displays the beginning and end of the text, with the middle replaced by an ellipsis.
- `TailTruncation` — shows the beginning of the text, truncating the end.

Display a specific number of lines

The number of lines displayed by a `Label` can be specified by setting the `MaxLines` property to an `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" />
```

Set line height

The vertical height of a `Label` can be customized by setting the `Label.LineHeight` property to a `double` value.

NOTE

- On iOS, the `Label.LineHeight` property changes the line height of text that fits on a single line, and text that wraps onto multiple lines.
- On Android, the `Label.LineHeight` property only changes the line height of text that wraps onto multiple lines.
- On Windows, the `Label.LineHeight` property changes the line height of text that wraps onto multiple lines.

The following 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 following screenshot shows the result of setting the `Label.LineHeight` property to 1.8:

 Lorem ipsum dolor sit amet, consectetur
 adipiscing elit. In facilisis nulla eu felis fringilla
 vulputate. Nullam porta eleifend lacinia. Donec
 at iaculis tellus.

Display HTML

The `Label` class has a `TextType` property, which determines whether the `Label` object 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 `TextType` property.
- `Html` indicates that the `Label` will display HTML text.

Therefore, `Label` objects can display HTML by setting the `TextType` property to `Html`, and the `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 `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.

IMPORTANT

Displaying HTML in a `Label` is limited to the HTML tags that are supported by the underlying platform.

Decorate text

Underline and strikethrough text decorations can be applied to `Label` objects by setting the `TextDecorations` property to one or more `TextDecorations` enumeration members:

- `None`
- `Underline`
- `Strikethrough`

The following example demonstrates setting the `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:

```
Label underlineLabel = new Label { Text = "This is underlined text.", TextDecorations = TextDecorations.Underline };
Label strikethroughLabel = new Label { Text = "This is text with strikethrough.", TextDecorations = TextDecorations.Strikethrough };
Label bothLabel = new Label { Text = "This is underlined text with strikethrough.", TextDecorations = TextDecorations.Underline | TextDecorations.Strikethrough };
```

The following screenshot shows the `TextDecorations` enumeration members applied to `Label` instances:

This is underlined text.

~~This is text with strikethrough.~~

~~This is underlined text with strikethrough.~~

NOTE

Text decorations can also be applied to `Span` instances. For more information about the `Span` class, see [Use 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" />
```

Use formatted text

`Label` exposes 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.

NOTE

It's not possible to display HTML in a `Span`.

`Span` defines the following properties:

- `BackgroundColor`, of type `Color`, which represents the color of the span background.
- `CharacterSpacing`, of type `double`, sets the spacing between characters in the displayed text.
- `FontAttributes`, of type `FontAttributes`, determines text style.

- `FontAutoScalingEnabled`, of type `bool`, defines whether the text will reflect scaling preferences set in the operating system. The default value of this property is `true`.
- `FontFamily`, of type `string`, defines the font family.
- `FontSize`, of type `double`, defines the font size.
- `LineHeight`, of type `double`, specifies the multiplier to apply to the default line height when displaying text.
- `Style`, of type `Style`, which is the style to apply to the span.
- `Text`, of type `string`, defines the text displayed as the content of the `Span`.
- `TextColor`, of type `Color`, defines the color of the displayed text.
- `TextDecorations`, of type `TextDecorations`, specifies the text decorations (underline and strikethrough) that can be applied.
- `TextTransform`, of type `TextTransform`, specifies the casing of the displayed text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

NOTE

The `Span.LineHeight` property has no effect on Windows.

In addition, the `GestureRecognizers` property can be used to define a collection of gesture recognizers that will respond to gestures on the `Span`.

The following XAML example demonstrates a `FormattedText` property that consists of three `Span` instances:

```
<Label LineBreakMode="WordWrap">
    <Label.FormattedText>
        <FormattedString>
            <Span Text="Red Bold, " TextColor="Red" FontAttributes="Bold" />
            <Span Text="default, " FontSize="14">
                <Span.GestureRecognizers>
                    <TapGestureRecognizer Command="{Binding TapCommand}" />
                </Span.GestureRecognizers>
            </Span>
            <Span Text="italic small." FontAttributes="Italic" FontSize="12" />
        </FormattedString>
    </Label.FormattedText>
</Label>
```

The equivalent C# code is:

```
FormattedString formattedString = new FormattedString ();
formattedString.Spans.Add (new Span { Text = "Red bold, ", TextColor = Colors.Red, FontAttributes =
FontAttributes.Bold });

Span 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 = 14 });

Label label = new Label { FormattedText = formattedString };
```

The following screenshot shows the resulting `Label` that contains three `Span` objects:

Red Bold, default, italic small.

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 examples. Therefore, when this `Span` is tapped the `TapGestureRecognizer` will respond by executing the `ICommand` defined by the `Command` property. For more information about tap gesture recognition, see [Recognize a tap gesture](#).

Create a hyperlink

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 example, shows a `Label` whose content is set from multiple `Span` objects:

```
<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/dotnet/maui/" />
                </Span.GestureRecognizers>
            </Span>
            <Span Text=" to view .NET MAUI documentation." />
        </FormattedString>
    </Label.FormattedText>
</Label>
```

In this example, the first and third `Span` instances contain 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 screenshot:

Alternatively, click [here](#) to view .NET MAUI documentation.

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:

```

using System.Windows.Input;

public partial class MainPage : ContentPage
{
    // Launcher.OpenAsync is provided by 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 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.

Create a reusable hyperlink class

The previous approach to creating a hyperlink requires writing repetitive code every time you require a hyperlink in your app. 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 example 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 = Colors.Blue;
        GestureRecognizers.Add(new TapGestureRecognizer
        {
            // Launcher.OpenAsync is provided by 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://schemas.microsoft.com/dotnet/2021/maui"
    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/dotnet/" />
                    <Span Text=" to view .NET documentation." />
                </FormattedString>
            </Label.FormattedText>
        </Label>
    </StackLayout>
</ContentPage>
```

ScrollView

9/20/2022 • 9 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `ScrollView` is a view that's capable of scrolling its content. By default, `ScrollView` scrolls its content vertically. A `ScrollView` can only have a single child, although this can be other layouts.

`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.

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 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 [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 .NET MAUI 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:ScrollViewDemos"
    x:Class="ScrollViewDemos.Views.XAML.ColorListPage"
    Title="ScrollView demo">
    <ScrollView Margin="20">
        <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 .NET MAUI. By default, a `ScrollView` scrolls vertically, which reveals more content:

Alice Blue
Antique White
Aqua
Aquamarine
Azure
Beige
Bisque
Black
Blanched Almond
Blue
Blue Violet
Brown
Burly Wood
Cadet Blue
Chartreuse
Chocolate
Coral
Cornflower Blue
Cornsilk
Crimson

The equivalent C# code is:

```

public class ColorListPage : ContentPage
{
    public ColorListPage()
    {
        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
            };
            horizontalStackLayout.Add(boxView);
            horizontalStackLayout.Add(label);

            return horizontalStackLayout;
        });
    }

    StackLayout stackLayout = new StackLayout();
    BindableLayout.SetItemsSource(stackLayout, NamedColor.All);
    BindableLayout.SetItemTemplate(stackLayout, dataTemplate);

    ScrollView scrollView = new ScrollView
    {
        Margin = new Thickness(20),
        Content = stackLayout
    };

    Title = "ScrollView demo";
    Content = scrollView;
}
}

```

For more information about bindable layouts, see [BindableLayout](#).

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 `Grid`. 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 `Grid`, it doesn't receive a specific height. The `Grid` 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 `RowDefinition` of the `Grid` row that contains the `ScrollView` should be set to `*`. This will cause the `Grid` 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 `Grid`:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ScrollViewDemos.Views.XAML.BlackCatPage"
    Title="ScrollView as a child layout demo">
    <Grid Margin="20"
        RowDefinitions="Auto,*,Auto">
        <Label Text="THE BLACK CAT by Edgar Allan Poe"
            FontSize="14"
            FontAttributes="Bold"
            HorizontalOptions="Center" />
        <ScrollView x:Name="scrollView"
            Grid.Row="1"
            VerticalOptions="FillAndExpand"
            Scrolled="OnScrollViewScrolled">
            <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>
        <Button Grid.Row="2"
            Text="Scroll to end"
            Clicked="OnButtonClicked" />
    </Grid>
</ContentPage>

```

In this example, the root layout is a `Grid` that has a `Label`, `ScrollView`, and `Button` 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 BLACK CAT by Edgar Allan Poe

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.

From my infancy I was noted for the docility and humanity of my disposition. My tenderness of heart was even so conspicuous as to make me the jest of my companions. I was especially fond of animals, and was indulged by my parents with a great variety of pets. With these I spent most of my time, and never was so happy as when feeding and caressing them. This peculiarity of character grew with my growth, and, in my manhood, I derived from it one of my principal sources of pleasure. To those who have cherished an affection for a faithful and sagacious dog, I need hardly be at the trouble of explaining the nature or the intensity of the

The equivalent C# code is:

```

public class BlackCatPage : ContentPage
{
    public BlackCatPage()
    {
        Label titleLabel = new Label
        {
            Text = "THE BLACK CAT by Edgar Allan Poe",
            // More properties set here to define the Label appearance
        };

        StackLayout stackLayout = new StackLayout();
        stackLayout.Add(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

        ScrollView scrollView = new ScrollView();
        scrollView.Content = stackLayout;
        // ...

        Title = "ScrollView as a child layout demo";
        Grid grid = new Grid
        {
            Margin = new Thickness(20),
            RowDefinitions =
            {
                new RowDefinition { Height = new GridLength(0, GridUnitType.Auto) },
                new RowDefinition { Height = new GridLength(1, GridUnitType.Star) },
                new RowDefinition { Height = new GridLength(0, GridUnitType.Auto) }
            }
        };
        grid.Add(titleLabel);
        grid.Add(scrollView, 0, 1);
        grid.Add(button, 0, 2);

        Content = grid;
    }
}

```

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's raised 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 raised 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.

Shapes

9/20/2022 • 5 minutes to read • [Edit Online](#)

 [Browse the sample](#)

A .NET Multi-platform App UI (.NET MAUI) `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. .NET MAUI Shapes is available in the `Microsoft.Maui.Controls.Shapes` namespace.

`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.

.NET MAUI defines a number of objects that derive from the `Shape` class. These are `Ellipse`, `Line`, `Path`, `Polygon`, `Polyline`, `Rectangle`, and `RoundRectangle`.

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 [Brushes](#). For more information about valid `color` values, see [Colors](#).

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 .NET MAUI layout system, because of either an explicit `WidthRequest` and `HeightRequest` setting or because of its `HorizontalOptions` and `VerticalOptions` settings.

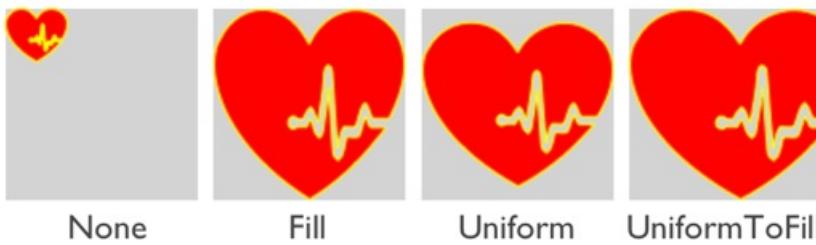
The `Stretch` enumeration defines the following members:

- `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

Ellipse

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [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 [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" />
```

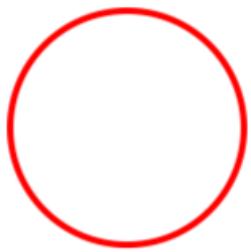
In this example, a red filled ellipse with dimensions 150x50 (device-independent units) is drawn:



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](#).

Fill rules

9/20/2022 • 4 minutes to read • [Edit Online](#)

 [Browse the sample](#)

Several .NET Multi-platform App UI (.NET MAUI) 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.

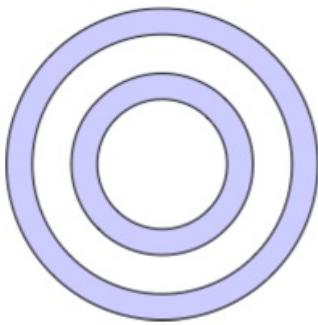
EvenOdd

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.

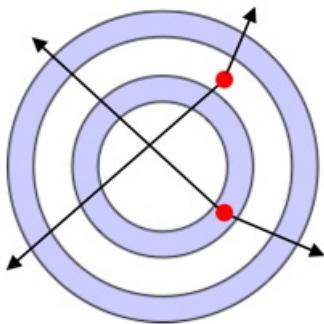
The following XAML example creates and renders a composite shape, with the `FillRule` defaulting to `EvenOdd`:

```
<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.

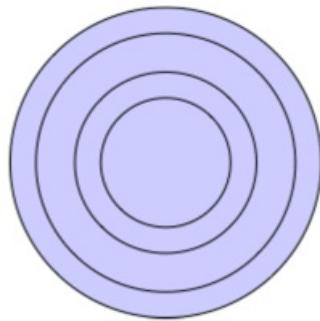
Nonzero

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.

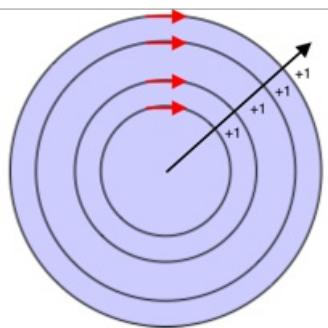
The following XAML example creates and renders a composite shape, with the `FillRule` set to `Nonzero`:

```
<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>

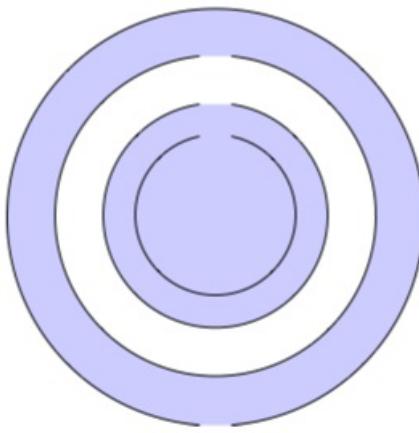
          <!-- Second ring -->
          <PathFigure StartPoint="120,100">
            <PathFigure.Segments>
              <PathSegmentCollection>
                <ArcSegment Size="70,70"
                            IsLargeArc="True"
                            SweepDirection="CounterClockwise"
                            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>

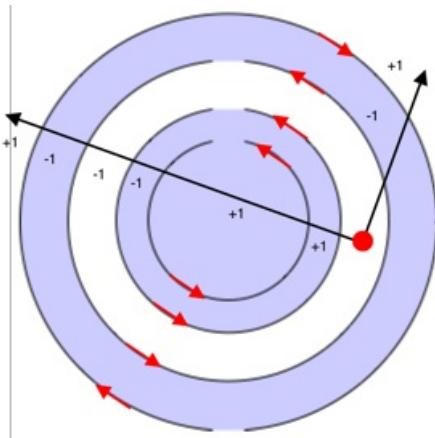
          <!-- Outer ring -->
          <PathFigure StartPoint="120,300">
            <PathFigure.Segments>
              <ArcSegment Size="130,130"
                            IsLargeArc="True"
                            SweepDirection="Clockwise"
                            Point="140,300" />
            </PathFigure.Segments>
          </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.

Geometries

9/20/2022 • 17 minutes to read • [Edit Online](#)

Browse the sample

The .NET Multi-platform App UI (.NET MAUI) `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 `RoundRectangleGeometry` 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 [.NET MAUI 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

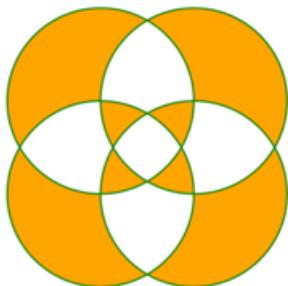
The `Children` property is the `ContentProperty` of the `GeometryGroup` class, and so does not need to be explicitly set from XAML.

For more information about the `FillRule` enumeration, see [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 [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.

Line

9/20/2022 • 2 minutes to read • [Edit Online](#)

{ } [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [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 [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](#).

Path

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [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 [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 [Path markup syntax](#).

Create a Path with Geometry objects

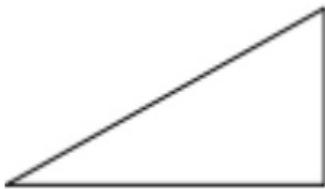
Curves and shapes can be described using `Geometry` objects, which are used to set the `Path` object's `Data` property. There are a variety of `Geometry` objects to choose from. The `EllipseGeometry`, `LineGeometry`, and `RectangleGeometry` classes describe relatively simple shapes. To create more complex 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 [Geometries](#).

Path markup syntax

9/20/2022 • 7 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) path markup syntax enables you to compactly specify path geometries in XAML.

Path markup 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 `FillRule` that specifies whether the geometry should use the `EvenOdd` or `Nonzero` fill rule. `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 [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.

Path transforms

9/20/2022 • 12 minutes to read • [Edit Online](#)

 [Browse the sample](#)

A .NET Multi-platform App UI (.NET MAUI) `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. .NET MAUI 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.

.NET MAUI 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

.NET MAUI 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 .NET MAUI 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 .NET MAUI:

```
<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" />
```

Polygon

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [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 [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 [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:

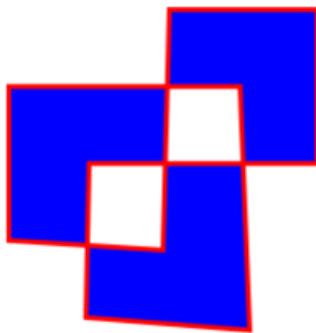


For more information about drawing a dashed polygon, see [Draw dashed shapes](#).

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.

Polyline

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [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 [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 [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 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:

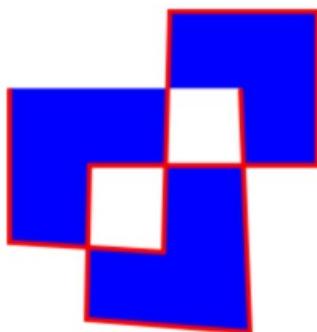


For more information about drawing a dashed polyline, see [Draw dashed shapes](#).

The following XAML example shows a polyline that uses the default fill rule:

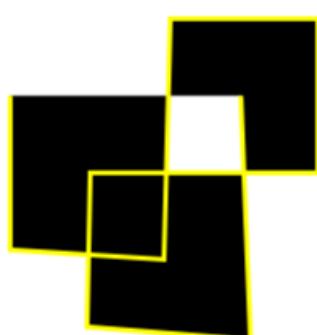
```
<Polyline Points="0 48, 0 144, 96 150, 100 0, 192 0, 192 96, 50 96, 48 192, 150 200 144 48"
    Fill="Blue"
    Stroke="Red"
    StrokeThickness="3" />
```

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.

Rectangle

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [.NET MAUI 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 [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.

NOTE

There's also a `RoundRectangle` class, that has a `CornerRadius` `BindableProperty`, which can be used to draw rectangles with rounded corners.

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](#).

WebView

9/20/2022 • 6 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `WebView` displays remote web pages, local HTML files, and HTML strings, in an app. The content displayed a `WebView` includes support for Cascading Style Sheets (CSS), and JavaScript. By default, .NET MAUI projects include the platform permissions required for a `WebView` to display a remote web page.

`WebView` defines the following properties:

- `Cookies`, of type `CookieContainer`, provides storage for a collection of cookies.
- `CanGoBack`, of type `bool`, indicates whether the user can navigate to previous pages. This is a read-only property.
- `CanGoForward`, of type `bool`, indicates whether the user can navigate forward. This is a read-only property.
- `Source`, of type `WebViewSource`, represents the location that the `WebView` displays.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `Source` property can be set to an `UrlWebViewSource` object or a `HtmlWebViewSource` object, which both derive from `WebViewSource`. A `UrlWebViewSource` is used for loading a web page specified with a URL, while a `HtmlWebViewSource` object is used for loading a local HTML file, or local HTML.

`WebView` defines a `Navigating` event that's raised when page navigation starts, and a `Navigated` event that's raised when page navigation completes. The `WebNavigatingEventArgs` object that accompanies the `Navigating` event defines a `Cancel` property, of type `bool`, that can be used to cancel navigation. The `WebNavigatedEventArgs` object that accompanies the `Navigated` event defines a `Result` property, of type `WebNavigationResult`, that indicates the navigation result.

IMPORTANT

A `WebView` must specify its `HeightRequest` and `WidthRequest` properties when contained in a `HorizontalStackLayout`, `StackLayout`, or `VerticalStackLayout`. If you fail to specify these properties, the `WebView` will not render.

Display a web page

To display a remote web page, set the `Source` property to a `string` that specifies the URI:

```
<WebView Source="https://docs.microsoft.com/dotnet/maui" />
```

The equivalent C# code is:

```
WebView webView = new WebView
{
    Source = "https://docs.microsoft.com/dotnet/maui"
};
```

URIs must be fully formed with the protocol specified.

NOTE

Despite the `Source` property being of type `WebViewSource`, the property can be set to a string-based URI. This is because .NET MAUI includes a type converter, and an implicit conversion operator, that converts the string-based URI to a `UrlWebViewSource` object.

Configure App Transport Security on iOS

Since version 9, iOS will only allow your app to communicate with secure servers. An app has to opt into enabling communication with insecure servers.

The following `Info.plist` configuration shows how to enable a specific domain to bypass Apple Transport Security (ATS) requirements:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSExceptionDomains</key>
    <dict>
        <key>mydomain.com</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>
            <key>NSTemporaryExceptionAllowsInsecureHTTPLoads</key>
            <true/>
            <key>NSTemporaryExceptionMinimumTLSVersion</key>
            <string>TLSv1.1</string>
        </dict>
    </dict>
    ...
</key>
```

It's best practice to only enable specific domains to bypass ATS, allowing you to use trusted sites while benefitting from additional security on untrusted domains.

The following `Info.plist` configuration shows how to disable ATS for an app:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
...
</key>
```

IMPORTANT

If your app requires a connection to an insecure website, you should always enter the domain as an exception using the `NSExceptionDomains` key instead of turning ATS off completely using the `NSAllowsArbitraryLoads` key.

Display local HTML

To display inline HTML, set the `Source` property to a `HtmlWebViewSource` object:

```
<WebView>
    <WebView.Source>
        <HtmlWebViewSource Html="<HTML><BODY><H1>.NET MAUI</H1><P>Welcome to
WebView.</P></BODY></HTML>" />
    </WebView.Source>
</WebView>
```

In XAML, HTML strings can become unreadable due to escaping the `<` and `>` symbols. Therefore, for greater readability the HTML can be inlined in a `CDATA` section:

```
<WebView>
    <WebView.Source>
        <HtmlWebViewSource>
            <HtmlWebViewSource.Html>
                <![CDATA[
                    <HTML>
                    <BODY>
                        <H1>.NET MAUI</H1>
                        <P>Welcome to WebView.</P>
                    </BODY>
                </HTML>
                ]]>
            </HtmlWebViewSource.Html>
        </HtmlWebViewSource>
    </WebView.Source>
</WebView>
```

The equivalent C# code is:

```
WebView webView = new WebView
{
    Source = new HtmlWebViewSource
    {
        Html = @"<HTML><BODY><H1>.NET MAUI</H1><P>Welcome to WebView.</P></BODY></HTML>"
    }
};
```

Display a local HTML file

To display a local HTML file, add the file to the `Resources\Raw` folder of your app project and set its build action to **MauiAsset**. Then, the file can be loaded from inline HTML that's defined in a `HtmlWebViewSource` object that's set as the value of the `Source` property:

```
<WebView>
    <WebView.Source>
        <HtmlWebViewSource>
            <HtmlWebViewSource.Html>
                <![CDATA[
                    <html>
                        <head>
                            </head>
                            <body>
                                <h1>.NET MAUI</h1>
                                <p>The CSS and image are loaded from local files!</p>
                                <p><a href="localfile.html">next page</a></p>
                            </body>
                        </html>
                ]]>
            </HtmlWebViewSource.Html>
        </HtmlWebViewSource>
    </WebView.Source>
</WebView>
```

The local HTML file can load Cascading Style Sheets (CSS), JavaScript, and images, provided that they've also been added to your app project with the **MauiAsset** build action.

For more information about raw assets, see [Raw assets](#).

Reload content

`WebView` has a `Reload` method that can be called to reload its source:

```
WebView webView = new WebView();
...
webView.Reload();
```

Perform navigation

`WebView` supports programmatic navigation with the `GoBack` and `GoForward` methods. These methods enable navigation through the `WebView` page stack, and should only be called after inspecting the values of the `CanGoBack` and `CanGoForward` properties:

```
WebView webView = new WebView();
...
// Go backwards, if allowed.
if (webView.CanGoBack)
{
    webView.GoBack();
}

// Go forwards, if allowed.
if (webView.CanGoForward)
{
    webView.GoForward();
}
```

When page navigation occurs in a `WebView`, either initiated programmatically or by the user, the following events occur:

- `Navigating`, which is raised when page navigation starts. The `WebNavigatingEventArgs` object that accompanies the `Navigating` event defines a `Cancel` property, of type `bool`, that can be used to cancel

navigation.

- `Navigated`, which is raised when page navigation completes. The `WebNavigatedEventArgs` object that accompanies the `Navigated` event defines a `Result` property, of type `WebNavigationResult`, that indicates the navigation result.

Set 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;

CookieContainer cookieContainer = new CookieContainer();
Uri uri = new Uri("https://docs.microsoft.com/dotnet/maui", UriKind.RelativeOrAbsolute);

Cookie cookie = new Cookie
{
    Name = "DotNetMAUICookie",
    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.

Invoke 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 `EvaluateJavaScriptAsync` method, which is shown in the following example:

```
Entry numberEntry = new Entry { Text = "5" };
Label resultLabel = new Label();
WebView 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>
```

Launch the system browser

It's possible to open a URI in the system web browser with the `Launcher` class, that's provided by `Microsoft.Maui.Essentials`. This is achieved by calling it's `OpenAsync` method, passing in a `string` or `Uri` argument that represents the URI to open:

```
await Launcher.OpenAsync("https://docs.microsoft.com/dotnet/maui");
```

For more information, see [Launcher](#).

Button

9/20/2022 • 9 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Button` displays text and responds to a tap or click that directs the app to carry out a task. A `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. When the `Button` is pressed with a finger or clicked with a mouse it initiates that command.

`Button` defines the following properties:

- `BorderColor`, of type `Color`, describes the border color of the button.
- `BorderWidth`, of type `double`, defines the width of the button's border.
- `CharacterSpacing`, of type `double`, defines the spacing between characters of the button's text.
- `Command`, of type `ICommand`, defines the command that's executed when the button is tapped.
- `CommandParameter`, of type `object`, is the parameter that's passed to `Command`.
- `ContentLayout`, of type `ButtonContentLayout`, defines the object that controls the position of the button image and the spacing between the button's image and text.
- `CornerRadius`, of type `int`, describes the corner radius of the button's border.
- `FontAttributes`, of type `FontAttributes`, determines text style.
- `FontAutoScalingEnabled`, of type `bool`, defines whether the button text will reflect scaling preferences set in the operating system. The default value of this property is `true`.
- `FontFamily`, of type `string`, defines the font family.
- `FontSize`, of type `double`, defines the font size.
- `ImageSource`, of type `ImageSource`, specifies a bitmap image to display as the content of the button.
- `LineBreakMode`, of type `LineBreakMode`, determines how text should be handled when it can't fit on one line.
- `Padding`, of type `Thickness`, determines the button's padding.
- `Text`, of type `string`, defines the text displayed as the content of the button.
- `TextColor`, of type `Color`, describes the color of the button's text.
- `TextTransform`, of type `TextTransform`, defines the casing of the button's text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

NOTE

While `Button` defines an `ImageSource` 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.

In addition, `Button` defines `Clicked`, `Pressed`, and `Released` events. The `Clicked` event is raised when a `Button` tap with a finger or mouse pointer is released from the button's surface. The `Pressed` event is raised when a finger presses on a `Button`, or a mouse button is pressed with the pointer positioned over the `Button`. The `Released` event is raised when the finger or mouse button is released. Generally, a `Clicked` event is also raised 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.

IMPORTANT

A `Button` must have its `IsEnabled` property set to `true` for it to respond to taps.

Create a Button

To create a button, create a `Button` object and handle its `Clicked` event.

The following XAML example show how to create a `Button`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ButtonDemos.BasicButtonClickPage"
    Title="Basic Button Click">
    <StackLayout>
        <Button Text="Click to Rotate Text!"
            VerticalOptions="Center"
            HorizontalOptions="Center"
            Clicked="OnButtonClicked" />
        <Label x:Name="label"
            Text="Click the Button above"
            FontSize="18"
            VerticalOptions="Center"
            HorizontalOptions="Center" />
    </StackLayout>
</ContentPage>
```

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:

```
public partial class BasicButtonClickPage : ContentPage
{
    public BasicButtonClickPage ()
    {
        InitializeComponent ();
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        await label.RelRotateTo(360, 1000);
    }
}
```

In this example, 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. The `Clicked` handler calls an animation function that rotates the `Label` 360 degrees in 1000 milliseconds:



The equivalent C# code to create a `Button` is:

```

Button button = new Button
{
    Text = "Click to Rotate Text!",
    VerticalOptions = LayoutOptions.Center,
    HorizontalOptions = LayoutOptions.Center
};
button.Clicked += async (sender, args) => await label.RelRotateTo(360, 1000);

```

Use the command interface

An app can 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) pattern. In an MVVM application, the viewmodel defines properties of type `ICommand` that are then connected to `Button` objects with data bindings. .NET MAUI also defines `Command` and `Command<T>` classes that implement the `ICommand` interface and assist the viewmodel in defining properties of type `ICommand`. For more information about commanding, see [Commanding](#).

The following example shows a very simple viewmodel class that defines a property of type `double` named `Number`, and two properties of type `ICommand` named `MultiplyBy2Command` and `DivideBy2Command`:

```

public class CommandDemoViewModel : INotifyPropertyChanged
{
    double number = 1;

    public event PropertyChangedEventHandler PropertyChanged;

    public ICommand MultiplyBy2Command { get; private set; }
    public ICommand DivideBy2Command { get; private set; }

    public CommandDemoViewModel()
    {
        MultiplyBy2Command = new Command(() => Number *= 2);
        DivideBy2Command = new Command(() => Number /= 2);
    }

    public double Number
    {
        get
        {
            return number;
        }
        set
        {
            if (number != value)
            {
                number = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Number"));
            }
        }
    }
}

```

In this example, 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 value of the `Number` property.

The following XAML example consumes the `CommandDemoViewModel` class:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="18"
            VerticalOptions="Center"
            HorizontalOptions="Center" />
        <Button Text="Multiply by 2"
            VerticalOptions="Center"
            HorizontalOptions="Center"
            Command="{Binding MultiplyBy2Command}" />
        <Button Text="Divide by 2"
            VerticalOptions="Center"
            HorizontalOptions="Center"
            Command="{Binding DivideBy2Command}" />
    </StackLayout>
</ContentPage>
```

In this example, the `Label` element and two `Button` objects contain bindings to the three properties in the `CommandDemoViewModel` class. As the two `Button` objects 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's also possible for the `Command` objects to control the enabling and disabling of the `Button` objects. 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:

```

public 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));
    }
    ...
}

```

In this example, the calls to the `ChangeCanExecute` method of `Command` are required 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's also 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. For more information, see [Commanding](#).

Press and release the button

The `Pressed` event is raised when a finger presses on a `Button`, or a mouse button is pressed with the pointer positioned over the `Button`. The `Released` event is raised when the finger or mouse button is released.

Generally, a `Clicked` event is also raised 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 following XAML example shows a `Label` and a `Button` with handlers attached for the `Pressed` and `Released` events:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ButtonDemos.PressAndReleaseButtonPage"
    Title="Press and Release Button">
    <StackLayout>
        <Button Text="Press to Rotate Text!"
            VerticalOptions="Center"
            HorizontalOptions="Center"
            Pressed="OnButtonPressed"
            Released="OnButtonReleased" />
        <Label x:Name="label"
            Text="Press and hold the Button above"
            FontSize="18"
            VerticalOptions="Center"
            HorizontalOptions="Center" />
    </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
{
    IDispatcherTimer timer;
    Stopwatch stopwatch = new Stopwatch();

    public PressAndReleaseButtonPage()
    {
        InitializeComponent();

        timer = Dispatcher.CreateTimer();
        timer.Interval = TimeSpan.FromMilliseconds(16);
        timer.Tick += (s, e) =>
        {
            label.Rotation = 360 * (stopwatch.Elapsed.TotalSeconds % 1);
        };
    }

    void OnButtonPressed(object sender, EventArgs args)
    {
        stopwatch.Start();
        timer.Start();
    }

    void OnButtonReleased(object sender, EventArgs args)
    {
        stopwatch.Stop();
        timer.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.

Button visual states

`Button` has a `Pressed` `VisualState` that can be used to initiate a visual change to the `Button` when pressed, 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>

```

In this example, 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 [Visual states](#).

Use bitmaps with buttons

The `Button` class defines an `ImageSource` property that allows you to display a small 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.

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`, and its 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.

In XAML, you can create a `Button` and set the `ContentLayout` property by specifying only the enumeration member, or the spacing, or both in any order separated by commas:

```

<Button Text="Button text"
       ImageSource="button.png"
       ContentLayout="Right, 20" />

```

The equivalent C# code is:

```
Button button = new Button
{
    Text = "Button text",
    ImageSource = new FileImageSource
    {
        File = "button.png"
    },
    ContentLayout = new Button.ButtonContentLayout(Button.ButtonContentLayout.ImageLayout.Right, 20)
};
```

Disable a Button

Sometimes an app enters a state where a `Button` click is not a valid operation. In such cases, the `Button` can be disabled by setting its `IsEnabled` property to `false`.

ImageButton

9/20/2022 • 5 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `ImageButton` view combines the `Button` view and `Image` view to create a button whose content is an image. When you press the `ImageButton` with a finger or click it with a mouse, it directs the app to carry out a task. However, unlike the `Button`, the `ImageButton` view has no concept of text and text appearance.

`ImageButton` defines the following properties:

- `Aspect`, of type `Aspect`, determines how the image will be scaled to fit the display area.
- `BorderColor`, of type `Color`, describes the border color of the button.
- `BorderWidth`, of type `double`, defines the width of the button's border.
- `Command`, of type `ICommand`, defines the command that's executed when the button is tapped.
- `CommandParameter`, of type `object`, is the parameter that's passed to `Command`.
- `CornerRadius`, of type `int`, describes the corner radius of the button's border.
- `IsLoading`, of type `bool`, represents the loading status of the image. The default value of this property is `false`.
- `IsOpaque`, of type `bool`, determines whether .NET MAUI should treat the image as opaque when rendering it. The default value of this property is `false`.
- `IsPressed`, of type `bool`, represents whether the button is being pressed. The default value of this property is `false`.
- `Padding`, of type `Thickness`, determines the button's padding.
- `Source`, of type `ImageSource`, specifies an image to display as the content of the button.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

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.
- `Center` - centers the image in the `ImageButton` while preserving the aspect ratio.

In addition, `ImageButton` defines `Clicked`, `Pressed`, and `Released` events. The `Clicked` event is raised when an `ImageButton` tap with a finger or mouse pointer is released from the button's surface. The `Pressed` event is raised when a finger presses on an `ImageButton`, or a mouse button is pressed with the pointer positioned over the `ImageButton`. The `Released` event is raised when the finger or mouse button is released. Generally, a `Clicked` event is also raised 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.

IMPORTANT

An `ImageButton` must have its `.IsEnabled` property set to `true` for it to respond to taps.

Create an ImageButton

To create an image button, create an `ImageButton` object, set its `Source` property and handle its `Clicked` event.

The following XAML example show how to create an `ImageButton`:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ControlGallery.Views.XAML.ImageButtonDemoPage"
    Title="ImageButton Demo">
    <StackLayout>
        <ImageButton Source="image.png"
            Clicked="OnImageButtonClicked"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
    </StackLayout>
</ContentPage>
```

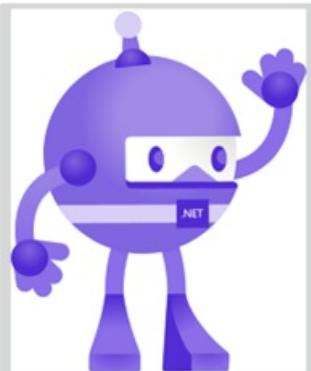
The `Source` property specifies the image that appears in the `ImageButton`. The `Clicked` event is set to an event handler named `OnImageButtonClicked`. This handler is located in the code-behind file:

```
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")}";
    }
}
```

In this example, 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. The `Clicked` handler increments a counter and displays the counter value in a `Label`:



5 ImageButton clicks

The equivalent C# code to create an `ImageButton` is:

```
Label label;
int clickTotal = 0;
...

ImageButton(imageButton = new ImageButton
{
    Source = "XamarinLogo.png",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.CenterAndExpand
});
imageButton.Clicked += (s, e) =>
{
    clickTotal += 1;
    label.Text = $"{clickTotal} ImageButton click{{(clickTotal == 1 ? "" : "s")}}";
};
```

Use the command interface

An app can 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) pattern. For more information about commanding, see [Use the command interface](#) in the [Button](#) article.

Press and release an ImageButton

The `Pressed` event is raised when a finger presses on a `ImageButton`, or a mouse button is pressed with the pointer positioned over the `ImageButton`. The `Released` event is raised when the finger or mouse button is released. Generally, the `Clicked` event is also raised 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 [Press and release the button](#) in the [Button](#) article.

ImageButton visual states

`ImageButton` has a `Pressed` `VisualState` that can be used to initiate a visual change to the `ImageButton` when pressed, provided that it's enabled.

The following XAML example shows how to define a visual state for the `Pressed` state:

```
<ImageButton Source="image.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>
```

In this example, 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 [Visual states](#).

Disable an ImageButton

Sometimes an app enters a state where an `ImageButton` click is not a valid operation. In those cases, the `ImageButton` should be disabled by setting its `.IsEnabled` property to `false`.

RadioButton

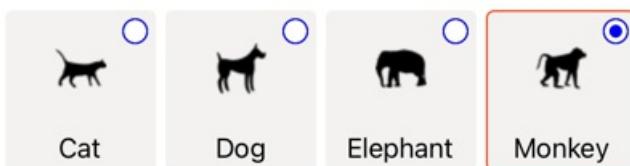
9/20/2022 • 10 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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`:



`RadioButton` 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.
- `FontAutoScalingEnabled`, of type `bool`, which defines whether an app's UI reflects text scaling preferences set in the operating system. The default value of this property is `true`.
- `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.

`RadioButton` also defines a `CheckedChanged` event that's raised 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 raised, 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 an `ILayout`.
- `SelectedValue`, of type `object`, which represents the value of the checked `RadioButton` object within an `ILayout` 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, Windows), 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 screenshot:

- What's your favorite animal?
- Cat
 - Dog
 - Elephant
 - Monkey

Display arbitrary content

On iOS and Windows, 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 screenshot:

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.

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 an `ILayout` 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 an `ILayout` 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 [Visual states](#).

Redefine RadioButton appearance

By default, `RadioButton` objects use handlers 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:

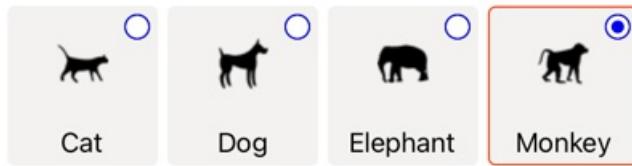
```

<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="Center" />
                    <Label Text="Cat"
                        HorizontalOptions="Center"
                        VerticalOptions="End" />
                </StackLayout>
            </RadioButton.Content>
        </RadioButton>
        <RadioButton Value="Dog">
            <RadioButton.Content>
                <StackLayout>
                    <Image Source="dog.png"
                        HorizontalOptions="Center"
                        VerticalOptions="Center" />
                    <Label Text="Dog"
                        HorizontalOptions="Center"
                        VerticalOptions="End" />
                </StackLayout>
            </RadioButton.Content>
        </RadioButton>
        <RadioButton Value="Elephant">
            <RadioButton.Content>
                <StackLayout>
                    <Image Source="elephant.png"
                        HorizontalOptions="Center"
                        VerticalOptions="Center" />
                    <Label Text="Elephant"
                        HorizontalOptions="Center"
                        VerticalOptions="End" />
                </StackLayout>
            </RadioButton.Content>
        </RadioButton>
        <RadioButton Value="Monkey">
            <RadioButton.Content>
                <StackLayout>
                    <Image Source="monkey.png"
                        HorizontalOptions="Center"
                        VerticalOptions="Center" />
                    <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 [Control templates](#).

Disable a RadioButton

Sometimes an app 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`.

RefreshView

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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.

Create a RefreshView

To add a `RefreshView` to a page, create a `RefreshView` object and set its `IsRefreshing` and `Command` properties. Then set its child to a scrollable control.

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:

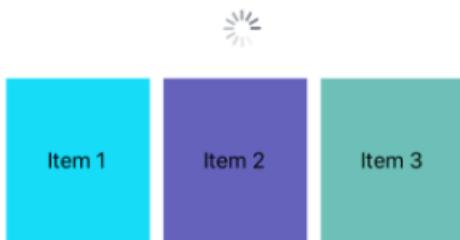
```
RefreshView refreshView = new RefreshView();
ICommand refreshCommand = new Command(() =>
{
    // IsRefreshing is true
    // Refresh data here
    refreshView.IsRefreshing = false;
});
refreshView.Command = refreshCommand;

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 layout](#).

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. The following screenshot shows the progress circle on iOS:



NOTE

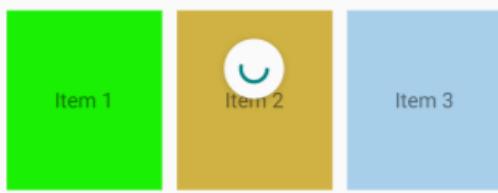
Manually setting the `IsRefreshing` property to `true` will trigger the refresh visualization, and will execute the `ICommand` defined by the `Command` property.

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 Android screenshot shows a `RefreshView` with the `RefreshColor` property:



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 app may enter a state where pull to refresh is not a valid operation. In such cases, the `RefreshView` can be

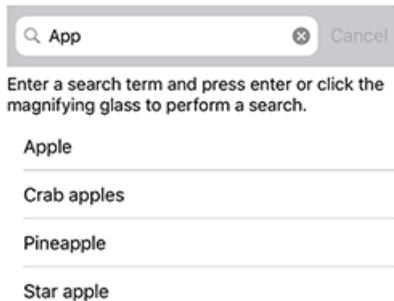
disabled by setting its `IsEnabled` property to `false`. This will prevent users from being able to trigger pull to refresh.

Alternatively, when defining the `Command` property, the `CanExecute` delegate of the `ICommand` can be specified to enable or disable the command.

SearchBar

9/20/2022 • 3 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `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 iOS screenshot shows a `SearchBar` query with results displayed in a `ListView`:



`SearchBar` defines the following properties:

- `CancelButtonColor` is a `Color` that defines the color of the cancel button.
- `CharacterSpacing`, is a `double` that's the spacing between characters of the `SearchBar` text.
- `CursorPosition` is an `int` that determines the position at which the next character will be inserted into the string stored in the `Text` property.
- `FontAttributes` is a `FontAttributes` enum value that determines whether the `SearchBar` font is bold, italic, or neither.
- `FontAutoScalingEnabled` is a `bool` which defines whether an app's UI reflects text scaling preferences set in the operating system.
- `FontFamily` is a `string` that determines the font family used by the `SearchBar`.
- `FontSize` is 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.
- `IsTextPredictionEnabled` is a `bool` that determines whether text prediction and automatic text correction is enabled.
- `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`.
- `SelectionLength` is an `int` that can be used to return or set the length of text selection within the `SearchBar`.
- `Text` is a `string` containing the query text in the `SearchBar`.
- `TextColor` is a `Color` that defines the query text color.
- `VerticalTextAlignment` is a `TextAlignment` enum value that defines the vertical alignment of the query text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

In addition, `SearchBar` defines a `SearchButtonPressed` event, which is raised when the search button is clicked,

or the enter key is pressed.

NOTE

`SearchBar` derives from the `InputView` class, from which it inherits additional properties and events.

Create a SearchBar

To create a search bar, create a `SearchBar` object and set its `Placeholder` property to text that instructs the user to enter a search term.

The following XAML example shows how to create a `SearchBar`:

```
<SearchBar Placeholder="Search items..." />
```

The equivalent C# code is:

```
SearchBar searchBar = new SearchBar { Placeholder = "Search items..." };
```

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`, which is called when the user either clicks the search button or presses the enter key.
- `TextChanged`, which is called anytime the text in the query box is changed. This event is inherited from the `InputView` class.

The following XAML example shows an event handler attached to the `TextChanged` event and uses a `ListView` to display search results:

```
<SearchBar TextChanged="OnTextChanged" />
<ListView x:Name="searchResults" >
```

In this example, the `TextChanged` event is set to an event handler named `OnTextChanged`. This handler is located in the code-behind file:

```
void OnTextChanged(object sender, EventArgs e)
{
    SearchBar searchBar = (SearchBar)sender;
    searchResults.ItemsSource = DataService.GetSearchResults(searchBar.Text);
}
```

In this example, a `DataService` class with a `GetSearchResults` method is used to return 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`.

Perform a search using a viewmodel

A search can be executed without event handlers by binding the `SearchCommand` property to an `ICommand` implementation. For more information about commanding, see [Commanding](#).

The following example shows a.viewmodel class that contains an `ICommand` property named `PerformSearch`:

```
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 following XAML example consumes the `SearchViewModel` class:

```
<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>
```

In this example, the `BindingContext` is set to an instance of the `SearchViewModel` class. The `SearchBar.SearchCommand` property binds to `PerformSearch` viewmodel property, and the `SearchCommandParameter` property binds to the `SearchBar.Text` property. Similarly, the `ListView.ItemsSource` property is bound to the `SearchResults` property of the viewmodel.

SwipeView

9/20/2022 • 10 minutes to read • [Edit Online](#)

{ } [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 raised when a swipe starts. The `SwipeStartedEventArgs` object that accompanies this event has a `SwipeDirection` property, of type `SwipeDirection`.
- `SwipeChanging` is raised 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 raised 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.

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 = Colors.LightGreen
};
favoriteSwipeItem.Invoked += OnFavoriteSwipeItemInvoked;

SwipeItem deleteSwipeItem = new SwipeItem
{
    Text = "Delete",
    IconImageSource = "delete.png",
    BackgroundColor = Colors.LightPink
};
deleteSwipeItem.Invoked += OnDeleteSwipeItemInvoked;

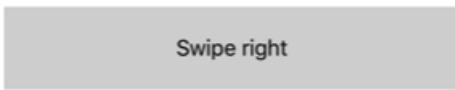
List<SwipeItem> swipeItems = new List<SwipeItem>() { favoriteSwipeItem, deleteSwipeItem };

// SwipeView content
Grid grid = new Grid
{
    HeightRequest = 60,
    WidthRequest = 300,
    BackgroundColor = Colors.LightGray
};
grid.Add(new Label
{
    Text = "Swipe right",
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.Center
});

SwipeView swipeView = new SwipeView
{
    LeftItems = new SwipeItems(swipeItems),
    Content = grid
};

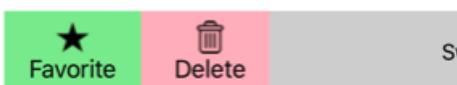
```

In this example, the `SwipeView` content is a `Grid` that contains a `Label`:



Swipe right

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 raised 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.

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.

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 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 raised 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 app 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.

CheckBox

9/20/2022 • 3 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 the following properties:

- `IsChecked`, of type `bool`, which indicates whether the `CheckBox` is checked. This property has a default binding mode of `TwoWay`.
- `Color`, of type `Color`, which indicates the color of the `CheckBox`.

These properties are backed by `BindableProperty` objects, which means that they can be styled, and be the target of data bindings.

`CheckBox` defines a `CheckedChanged` event that's raised 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 raised, 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 screenshot:



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 screenshot:



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
};
```

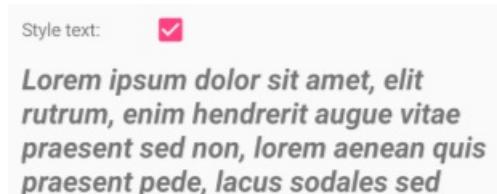
Data bind a CheckBox

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="18" />
        </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.

The following screenshot shows the `Label` formatting when the `checkBox` is checked:



For more information about triggers, see [Triggers](#).

Disable a Checkbox

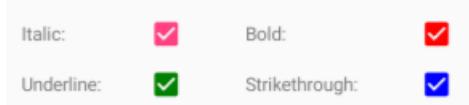
Sometimes an application enters a state where a `CheckBox` being checked is not a valid operation. In such cases, the `CheckBox` can be disabled by setting its `IsEnabled` property to `false`.

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 screenshot shows 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 [Visual states](#).

DatePicker

9/20/2022 • 2 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `DatePicker` invokes the platform's date-picker control and allows you 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.
- `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.

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) pattern.

WARNING

When setting `MinimumDate` and `MaximumDate`, make sure that `MinimumDate` is always less than or equal to `MaximumDate`. Otherwise, `DatePicker` will raise an exception.

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`.

The `DatePicker` fires a `DateSelected` event when the user selects a date.

Create a DatePicker

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="01/01/2022"  
           MaximumDate="12/31/2022"  
           Date="06/21/2022" />
```

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, your app should attach a handler to the `Selected` event to be informed when the user selects a new date.

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)  
};
```

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.

Slider

9/20/2022 • 6 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Slider` is a horizontal bar that you can manipulate to select a `double` value from a continuous range.

`Slider` defines the following properties:

- `Minimum`, of type `double`, is the minimum of the range, with a default value of 0.
- `Maximum`, of type `double`, is the maximum of the range, with a default value of 1.
- `Value`, of type `double`, is the slider's value, which can range between `Minimum` and `Maximum` and has a default value of 0.
- `MinimumTrackColor`, of type `Color`, is the bar color on the left side of the thumb.
- `MaximumTrackColor`, of type `Color`, is the bar color on the right side of the thumb.
- `ThumbColor` of type `Color`, is the thumb color.
- `ThumbImageSource`, of type `ImageSource`, is the image to use for the thumb, of type `ImageSource`.
- `DragStartedCommand`, of type `ICommand`, which is executed at the beginning of a drag action.
- `DragCompletedCommand`, of type `ICommand`, which is executed at the end of a drag action.

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) pattern.

NOTE

The `ThumbColor` and `ThumbImageSource` properties are mutually exclusive. If both properties are set, the `ThumbImageSource` property will take precedence.

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`. 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. For more information on setting the `Minimum` and `Maximum` properties, see [Precautions](#).

`Slider` defines a `ValueChanged` event that's raised 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 raised when the `Value` property is coerced as described in the previous paragraph. The `ValueChangedEventArgs` object that accompanies the `ValueChanged` event has `OldValue` and `NewValue` properties, of type `double`. At the time the event is raised, the value of `NewValue` is the same as the `Value` property of the `Slider` object.

`Slider` also defines `DragStarted` and `DragCompleted` events, that are raised at the beginning and end of the drag action. Unlike the `ValueChanged` event, the `DragStarted` and `DragCompleted` events are only raised 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`. Keep the default `HorizontalOptions` setting of `Fill`, and don't use a width of `Auto` when putting `Slider` in a `Grid` layout.

Create a Slider

The following example shows how to create a `Slider`, with two `Label` objects:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="18"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <Slider Maximum="360"
            ValueChanged="OnSliderValueChanged" />
        <Label x:Name="displayLabel"
            Text="(uninitialized)"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
    </StackLayout>
</ContentPage>
```

In this example, the `Slider` is initialized to have a `Maximum` property of 360. The second `Label` displays the text "(uninitialized)" until the `Slider` is manipulated, which causes the first `ValueChanged` event to be raised.

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);
    }
}
```

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`:



The Slider value is 304

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;
```

The equivalent C# code for creating a `Slider` is:

```
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);
};
```

Data bind a Slider

The `ValueChanged` event handler can be eliminated by using data binding to respond to the `Slider` value changing:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="18"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <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="Center" />
    </StackLayout>
</ContentPage>

```

In this example, 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. When the page first appears, the second `Label` displays the text string with the value. 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 example 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
};

```

In this example, 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. The properties must be set 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, the following example won't raise an exception:

```
Slider slider = new Slider  
{  
    Value = 10  
};
```

Instead, the `Value` property is coerced to the `Maximum` value of 1.

A previous example set `Maximum` to 20, and `Minimum` to 10:

```
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 raised:

```
<Slider ValueChanged="OnSliderValueChanged"  
        Maximum="20"  
        Minimum="10" />
```

When `Minimum` is set to 10, `Value` is also set to 10, and the `ValueChanged` event is raised. 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.

Stepper

9/20/2022 • 6 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `stepper` enables a numeric value to be selected from a range of values. It 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) pattern.

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`. 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](#).

`Stepper` defines a `ValueChanged` event that's raised 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 raised when the `Value` property is coerced as previously described. The `ValueChangedEventArgs` object that accompanies the `ValueChanged` event has `oldValue` and `NewValue`, of type `double`. At the time the event is raised, the value of `NewValue` is the same as the `Value` property of the `Stepper` object.

Create a Stepper

The following example shows how to create a `Stepper`, with two `Label` objects:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="18"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <Stepper Maximum="360"
            Increment="30"
            HorizontalOptions="Center"
            ValueChanged="OnStepperValueChanged" />
        <Label x:Name="_displayLabel"
            Text="(uninitialized)"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
    </StackLayout>
</ContentPage>

```

In this example, 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 second `Label` displays the text "(uninitialized)" until the `Stepper` is manipulated, which causes the first `ValueChanged` event to be raised.

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);
    }
}

```

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`:

ROTATING TEXT



The Stepper value is 30

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;
```

The equivalent C# code for creating a `Stepper` is:

```
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);
};
```

Data bind a Stepper

The `ValueChanged` event handler can be eliminated by using data binding to respond to the `Stepper` value changing:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="18"
            HorizontalOptions="Center"
            VerticalOptions="Center" />
        <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="Center" />
    </StackLayout>
</ContentPage>

```

In this example, 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. When the page first appears, the second `Label` displays the text string with the value. 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 example 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
};

```

In this example, 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 won't raise an exception:

```
Stepper stepper = new Stepper  
{  
    Value = 180  
};
```

Instead, the `Value` property is coerced to the `Maximum` value of 100.

A previous example set `Maximum` to 360 and `Minimum` to 180:

```
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 raised:

```
<Stepper ValueChanged="OnStepperValueChanged"  
        Maximum="360"  
        Minimum="180" />
```

When `Minimum` is set to 180, `Value` is also set to 180, and the `ValueChanged` event is raised. 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.

Switch

9/20/2022 • 4 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 following screenshot shows a `switch` control in its on and off toggle states:



The `switch` control defines the following properties:

- `IsToggled` is a `boolean` value that indicates whether the `switch` is on. The default value of this property is `false`.
- `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 `BindableProperty` objects, which means they can be styled and be the target of data bindings.

The `switch` control defines a `Toggled` event that's raised 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 raised, 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 = Colors.Orange, ThumbColor = Colors.Green };
```

The following screenshot shows the `Switch` in its on and off toggle states, with the `OnColor` and `ThumbColor` properties set:



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="18" />
        </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 [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.

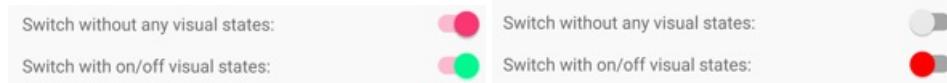
The following XAML example shows how to define visual states for the `On` and `Off` states:

```

<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:



For more information about visual states, see [Visual states](#).

Disable a Switch

An app 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`.

TimePicker

9/20/2022 • 2 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `TimePicker` invokes the platform's time-picker control and allows you 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.
- `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) pattern.

NOTE

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.

Create a TimePicker

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.

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"
};
```

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.

Editor

9/20/2022 • 7 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Editor` allows you to enter and edit multiple lines of text.]

`Editor` defines the following properties:

- `AutoSize`, of type `EditorAutoSizeOption`, defines whether the editor will change size to accommodate user input. By default, the editor doesn't auto size.
- `CharacterSpacing`, of type `double`, sets the spacing between characters in the entered text.
- `CursorPosition`, of type `int`, defines the position of the cursor within the editor.
- `FontAttributes`, of type `FontAttributes`, determines text style.
- `FontAutoScalingEnabled`, of type `bool`, defines whether the text will reflect scaling preferences set in the operating system. The default value of this property is `true`.
- `FontFamily`, of type `string`, defines the font family.
- `FontSize`, of type `double`, defines the font size.
- `HorizontalTextAlignment`, of type `TextAlignment`, defines the horizontal alignment of the text.
- `IsTextPredictionEnabled`, of type `bool`, controls whether text prediction and automatic text correction is enabled.
- `Placeholder`, of type `string`, defines the text that's displayed when the control is empty.
- `PlaceholderColor`, of type `Color`, defines the color of the placeholder text.
- `SelectionLength`, of type `int`, represents the length of selected text within the editor.
- `Text`, of type `string`, defines the text entered into the editor.
- `TextColor`, of type `Color`, defines the color of the entered text.
- `VerticalTextAlignment`, of type `TextAlignment`, defines the vertical alignment of the text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

In addition, `Editor` defines a `Completed` event, which is raised when the user finalizes text in the `Editor` with the return key.

`Editor` derives from the `InputView` class, from which it inherits the following properties:

- `IsReadOnly`, of type `bool`, defines whether the user should be prevented from modifying text. The default value of this property is `false`.
- `IsSpellCheckEnabled`, of type `bool`, controls whether spell checking is enabled.
- `Keyboard`, of type `Keyboard`, specifies the virtual keyboard that's displayed when entering text.
- `MaxLength`, of type `int`, defines the maximum input length.
- `TextTransform`, of type `TextTransform`, specifies the casing of the entered text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

In addition, `InputView` defines a `TextChanged` event, which is raised when the text in the `Editor` changes. The `TextChangedEventArgs` object that accompanies the `TextChanged` event has `NewTextValue` and `OldTextValue` properties, which specify the new and old text, respectively.

For information about specifying fonts on an `Editor`, see [Fonts](#).

Create an Editor

The following example shows how to create an `Editor`:

```
<Editor x:Name="editor"
    Placeholder="Enter your response here"
    HeightRequest="250"
    TextChanged="OnEditorTextChanged"
    Completed="OnEditorCompleted" />
```

The equivalent C# code is:

```
Editor editor = new Editor { Placeholder = "Enter text", HeightRequest = 250 };
editor.TextChanged += OnEditorTextChanged;
editor.Completed += OnEditorCompleted;
```

The following screenshot shows the resulting `Editor` on Android:



Entered text can be accessed by reading the `Text` property, and the `TextChanged` and `Completed` events signal that the text has changed or been completed.

The `TextChanged` event is raised when the text in the `Editor` changes, and the `TextChangedEventArgs` provide the text before and after the change via the `OldTextValue` and `NewTextValue` properties:

```
void OnEditorTextChanged(object sender, TextChangedEventArgs e)
{
    string oldText = e.OldTextValue;
    string newText = e.NewTextValue;
    string myText = editor.Text;
}
```

The `Completed` event is raised when the user has ended input by pressing the return key on the keyboard, or by pressing the Tab key on Windows. The handler for the event is a generic event handler:

```
void OnEditorCompleted(object sender, EventArgs e)
{
    string text = ((Editor)sender).Text;
}
```

Set character spacing

Character spacing can be applied to an `Editor` by setting the `CharacterSpacing` property to a `double` value:

```
<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.

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" />
```

A `MaxLength` property value of 0 indicates that no input will be allowed, and a value of `int.MaxValue`, which is the default value for an `Editor`, indicates that there is no effective limit on the number of characters that may be entered.

Auto-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 as follows:

```
<Editor Text="Enter text here"
        AutoSize="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.

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" />
```

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.

The following example shows setting the `Keyboard` property:

```
<Editor Keyboard="Chat" />
```

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:

```
Editor 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" />
```

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" />
```

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.

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" />
```

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.

Entry

9/20/2022 • 9 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Entry` allows you to enter and edit a single line of text. In addition, the `Entry` can be used as a password field.

`Entry` defines the following properties:

- `CharacterSpacing`, of type `double`, sets the spacing between characters in the entered text.
- `ClearButtonVisibility`, of type `clearButtonVisibility`, controls whether a clear button is displayed, which enables the user to clear the text. The default value of this property ensures that a clear button isn't displayed.
- `CursorPosition`, of type `int`, defines the position of the cursor within the entry.
- `FontAttributes`, of type `FontAttributes`, determines text style.
- `FontAutoScalingEnabled`, of type `bool`, defines whether the text will reflect scaling preferences set in the operating system. The default value of this property is `true`.
- `FontFamily`, of type `string`, defines the font family.
- `FontSize`, of type `double`, defines the font size.
- `Keyboard`, of type `Keyboard`, specifies the virtual keyboard that's displayed when entering text.
- `HorizontalTextAlignment`, of type `TextAlignment`, defines the horizontal alignment of the text.
- `IsPassword`, of type `bool`, specifies whether the entry should visually obscure typed text.
- `IsTextPredictionEnabled`, of type `bool`, controls whether text prediction and automatic text correction is enabled.
- `Placeholder`, of type `string`, defines the text that's displayed when the control is empty.
- `PlaceholderColor`, of type `Color`, defines the color of the placeholder text.
- `ReturnCommand`, of type `ICommand`, defines the command to be executed when the return key is pressed.
- `ReturnCommandParameter`, of type `object`, specifies the parameter for the `ReturnCommand`.
- `ReturnType`, of type `ReturnType`, specifies the appearance of the return button.
- `SelectionLength`, of type `int`, represents the length of selected text within the entry.
- `Text`, of type `string`, defines the text entered into the entry.
- `TextColor`, of type `Color`, defines the color of the entered text.
- `VerticalTextAlignment`, of type `TextAlignment`, defines the vertical alignment of the text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

In addition, `Entry` defines a `Completed` event, which is raised when the user finalizes text in the `Entry` with the return key.

`Entry` derives from the `InputView` class, from which it inherits the following properties:

- `IsReadOnly`, of type `bool`, defines whether the user should be prevented from modifying text. The default value of this property is `false`.
- `IsSpellCheckEnabled`, of type `bool`, controls whether spell checking is enabled.
- `MaxLength`, of type `int`, defines the maximum input length.
- `TextTransform`, of type `TextTransform`, specifies the casing of the entered text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data

bindings, and styled.

In addition, `InputView` defines a `TextChanged` event, which is raised when the text in the `Entry` changes. The `TextChangedEventArgs` object that accompanies the `TextChanged` event has `NewTextValue` and `OldTextValue` properties, which specify the new and old text, respectively.

For information about specifying fonts on an `Entry`, see [Fonts](#).

Create an Entry

The following example shows how to create an `Entry`:

```
<Entry x:Name="entry"
       Placeholder="Enter text"
       TextChanged="OnEntryTextChanged"
       Completed="OnEntryCompleted" />
```

The equivalent C# code is:

```
Entry entry = new Entry { Placeholder = "Enter text" };
entry.TextChanged += OnEntryTextChanged;
entry.Completed += OnEntryCompleted;
```

The following screenshot shows the resulting `Entry` on Android:



Entered text can be accessed by reading the `Text` property, and the `TextChanged` and `Completed` events signal that the text has changed or been completed.

The `TextChanged` event is raised when the text in the `Entry` changes, and the `TextChangedEventArgs` provide the text before and after the change via the `OldTextValue` and `NewTextValue` properties:

```
void OnEntryTextChanged(object sender, TextChangedEventArgs e)
{
    string oldText = e.OldTextValue;
    string newText = e.NewTextValue;
    string myText = entry.Text;
}
```

The `Completed` event is raised when the user has ended input by pressing the return key on the keyboard, or by pressing the Tab key on Windows. The handler for the event is a generic event handler:

```
void OnEntryCompleted(object sender, EventArgs e)
{
    string text = ((Entry)sender).Text;
}
```

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 `ReturnCommand`.

NOTE

The `VisualElement` class, which is in the `Entry` inheritance hierarchy, also has `Focused` and `Unfocused` events.

Set character spacing

Character spacing can be applied to an `Entry` by setting the `CharacterSpacing` property to a `double` value:

```
<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.

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" />
```

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.

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" />
```

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" />
```

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 `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:

```
<Entry Text=".NET MAUI"  
      ClearButtonVisibility="WhileEditing" />
```

The following screenshot shows an `Entry` on Android with the clear button enabled:



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" />
```

Obscure text entry

`Entry` provides the `IsPassword` property which visually obscures entered text when it's set to `true`:

```
<Entry IsPassword="true" />
```

The following screenshot shows an `Entry` whose input has been obscured:



Customize the keyboard

The virtual 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.

The following example shows setting the `Keyboard` property:

```
<Entry Keyboard="Chat" />
```

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:

```
Entry 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 virtual 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" />
```

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 Android and Windows, 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 [Commanding](#).

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" />
```

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" />
```

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.

Prevent text entry

Users can be prevented from modifying the text in an `Entry` by setting the `IsReadOnly` property to `true`:

```
<Entry Text="User input won't be accepted."  
      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.

ActivityIndicator

9/20/2022 • 2 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `ActivityIndicator` displays an animation to show that the application is engaged in a lengthy activity. Unlike `ProgressBar`, `ActivityIndicator` gives no indication of progress.

The appearance of an `ActivityIndicator` is platform-dependent, and the following screenshot shows an `ActivityIndicator` on iOS and Android:



iOS



Android

`ActivityIndicator` defines the following properties:

- `Color` is a `Color` value that defines the color of the `ActivityIndicator`.
- `IsRunning` is a `bool` value that indicates whether the `ActivityIndicator` should be visible and animating, or hidden. The default value of this property is `false`, which indicates that the `ActivityIndicator` isn't visible.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

Create an ActivityIndicator

To indicate a lengthy activity, create an `ActivityIndicator` object and sets its properties to define its appearance.

The following XAML example shows how to display an `ActivityIndicator`:

```
<ActivityIndicator IsRunning="true" />
```

The equivalent C# code is:

```
ActivityIndicator activityIndicator = new ActivityIndicator { IsRunning = true };
```

The following XAML example shows how to change the color of an `ActivityIndicator`:

```
<ActivityIndicator IsRunning="true"
                  Color="Orange" />
```

The equivalent C# code is:

```
ActivityIndicator activityIndicator = new ActivityIndicator
{
    IsRunning = true,
    Color = Colors.Orange
};
```

ProgressBar

9/20/2022 • 2 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `ProgressBar` indicates to users that the app is progressing through a lengthy activity. The progress bar is a horizontal bar that is filled to a percentage represented by a `double` value.

The appearance of a `ProgressBar` is platform-dependent, and the following screenshot shows a `ProgressBar` on iOS and Android:

iOS



Android



`ProgressBar` defines two properties:

- `Progress` is a `double` 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. The default value of this property is 0.
- `ProgressColor` is a `Color` values that defines the color of the `ProgressBar`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

`ProgressBar` also defines a `ProgressTo` method that animates the bar from its current value to a specified value. For more information, see [Animate a ProgressBar](#).

Create a ProgressBar

To indicate progress through a lengthy activity, create a `ProgressBar` object and set its properties to define its appearance.

The following XAML example shows how to display a `ProgressBar`:

```
<ProgressBar Progress="0.5" />
```

The equivalent C# code is:

```
ProgressBar progressBar = new ProgressBar { Progress = 0.5 };
```

WARNING

Do not use unconstrained horizontal layout options such as `Center`, `Start`, or `End` with `ProgressBar`. Keep the default `HorizontalOptions` value of `Fill`.

The following XAML example shows how to change the color of a `ProgressBar`:

```
<ProgressBar Progress="0.5"
             ProgressColor="Orange" />
```

The equivalent C# code is:

```
ProgressBar progressBar = new ProgressBar
{
    Progress = 0.5,
    ProgressColor = Colors.Orange
};
```

Animate a ProgressBar

The `ProgressTo` method animates the `ProgressBar` from its current `Progress` value to a provided value over time. The method accepts a `double` progress value, a `uint` duration in milliseconds, an `Easing` enum value and returns a `Task<bool>`. The following example 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](#).

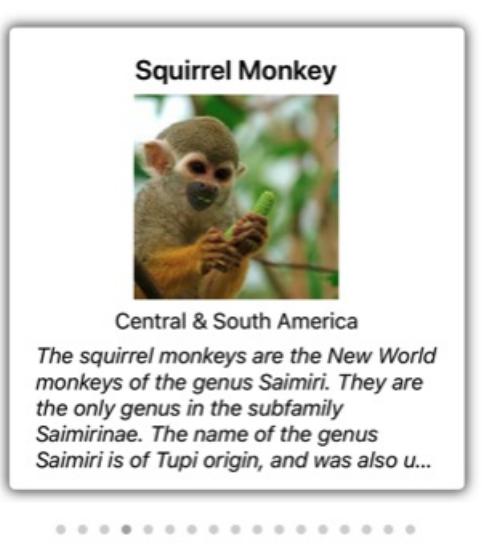
CarouselView

9/20/2022 • 2 minutes to read • [Edit Online](#)

{ } [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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. For more information about `CollectionView`, see [CollectionView](#).

Populate a CarouselView with data

9/20/2022 • 9 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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.

`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 .NET MAUI applications. For more information, see [Compiled bindings](#).

For information on how to change the `CarouselView` orientation, see [Specify 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 [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="18"
                            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();
    stackLayout.Add(nameLabel);
    stackLayout.Add(image);
    stackLayout.Add(locationLabel);
    stackLayout.Add(detailsLabel);

    Frame frame = new Frame { ... };
    StackLayout rootStackLayout = new StackLayout();
    rootStackLayout.Add(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 screenshot shows an example of templating each item:



For more information about data templates, see [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 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:

Squirrel Monkey



Central & South America

The squirrel monkeys are the New World monkeys of the genus Saimiri. They are the only genus in the subfamily Saimirinae. The name of the genus Saimiri is of Tupi origin, and was also u...

• • • • • • • • • • • • • • •

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 [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 = Colors.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 = Colors.LightPink
    };
    deleteSwipeItem.SetBinding(MenuItem.CommandProperty, new Binding("BindingContext.DeleteCommand", source: carouselView));
    deleteSwipeItem.SetBinding(MenuItem.CommandParameterProperty, ".");

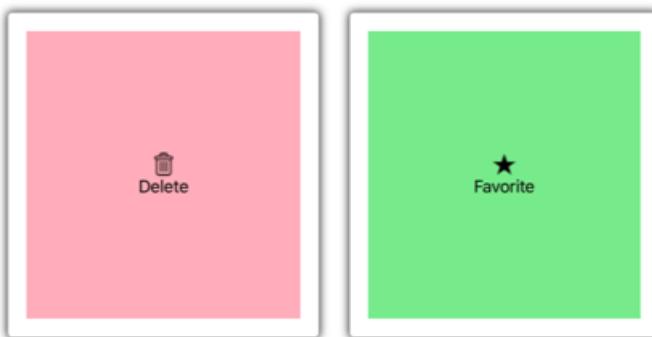
    swipeView.TopItems = new SwipeItems { favoriteSwipeItem };
    swipeView.BottomItems = new SwipeItems { deleteSwipeItem };

    StackLayout swipeViewStackLayout = new StackLayout { ... };
    swipeView.Content = swipeViewStackLayout;
    frame.Content = swipeView;
    stackLayout.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 bottom and from the top:



`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 [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 [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.

Specify CarouselView layout

9/20/2022 • 4 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [Control scrolling in a CarouselView 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="18"
                            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.

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="18"
                            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

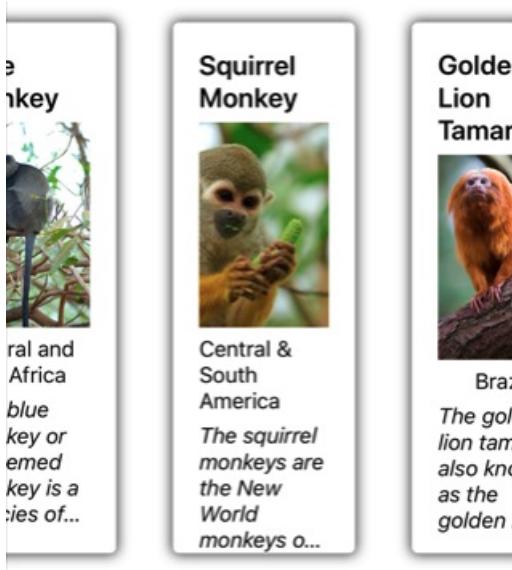
By default, `CarouselView` displays full items at once. However, this behavior can be changed by setting the `PeekAreaInsets` property to a `Thickness` value that specifies how much to make adjacent items partially visible by. This can be useful to indicate to users that there are additional items to 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://schemas.microsoft.com/dotnet/2021/maui"
    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`.

Configure CarouselView interaction

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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;
```

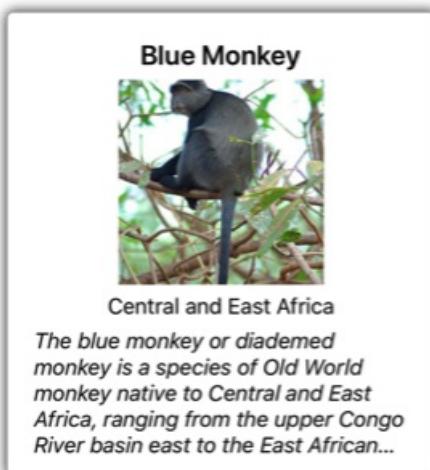
In this example, the `OnCurrentItemChanged` event handler is executed when the `CurrentItemChanged` event fires:

```
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:

Previous item: Capuchin Monkey

Current item: Blue Monkey



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;
}
```

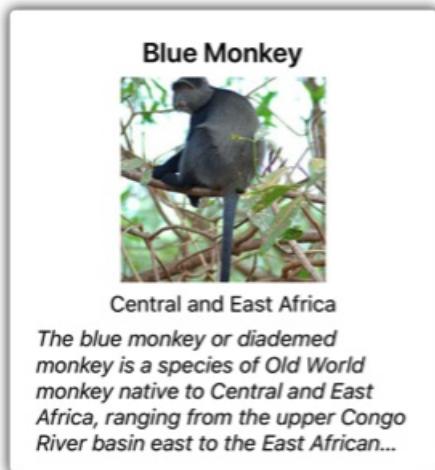
In this example, the `OnCurrentItemChanged` event handler exposes the previous and current positions:

Previous item: Capuchin Monkey

Current item: Blue Monkey

Previous item position: 1

Current item position: 2



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}"
...
</CarouselView>
```

The equivalent C# code is:

```
CarouselView carouselView = new CarouselView();
carouselView.SetBinding(ItemsView.ItemsSourceProperty, "Monkeys");
carouselView.SetBinding(CarouselView.PositionChangedCommandProperty, "PositionChangedCommand");
carouselView.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:

Squirrel Monkey



Central & South America

The squirrel monkeys are the New World monkeys of the genus Saimiri. They are the only genus in the subfamily Saimirinae. The name of the genus Saimiri is of Tupi origin, a...

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:

Current item: Squirrel Monkey

Position: 3



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 screenshot shows the `CurrentItem`, `PreviousItem`, and `NextItem` visual states:



Squirrel
Monkey

Central &
South America

Squirrel
monkeys are
World
keys o...



Blue
Monkey



Squirrel
Monkey

Central
South America

The squirrel
monkey or
diademed
monkey is a
species of...

For more information about visual states, see [Visual states](#).

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`.

Define an EmptyView for a CarouselView

9/20/2022 • 6 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [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 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 .NET MAUI 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:

```
StackLayout stackLayout = new StackLayout();
stackLayout.Add(new Label { Text = "No results matched your filter.", ... } );
stackLayout.Add(new Label { Text = "Try a broader filter?", ... } );

SearchBar searchBar = new SearchBar { ... };
CarouselView carouselView = new CarouselView
{
    EmptyView = new ContentView
    {
        Content = stackLayout
    }
};
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://schemas.microsoft.com/dotnet/2021/maui"
    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.IsChecked` 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 [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 ...
    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}"
        EmptyView="{Binding Source={x:Reference searchBar}, Path=Text}"
        EmptyViewTemplate="{StaticResource SearchSelector}">
        <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 DataTemplateSelector](#).

Control scrolling in a CarouselView

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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

One `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

Another `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 screenshot shows 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.



CollectionView

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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`.

The following screenshot shows a `collectionView` that uses a two-column vertical grid and allows multiple selections:



`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 [BindableLayout](#).

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 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 Configure CollectionView item 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.

CONCEPT	LISTVIEW API	COLLECTIONVIEW
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 Display grouped data in a 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, which scroll items into view. For more information, see Control scrolling in a CollectionView .

Populate a CollectionView with data

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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.

`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 .NET MAUI applications. For more information, see [Compiled bindings](#).

For information on how to change the `CollectionView` layout, see [Specify 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 [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.Add(image);
    grid.Add(nameLabel, 1, 0);
    grid.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 screenshot shows the result of templating each item in the list:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

*Southern Cameroon, Gabon,
Equatorial Guinea, and Congo*

For more information about data templates, see [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:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

Southern Cameroon, Gabon, Equatorial Guinea, and Congo

For more information about data template selectors, see [Create a 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 = Colors.White };
    // ...

    SwipeView swipeView = new SwipeView();
    SwipeItem favoriteSwipeItem = new SwipeItem
    {
        Text = "Favorite",
        IconImageSource = "favorite.png",
        BackgroundColor = Colors.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 = Colors.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:



Baboon

Africa & Asia

Favorite

Delete



Capuchin

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

Southern Cameroon, Gabon,
Equatorial Guinea, and Congo

`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 [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:

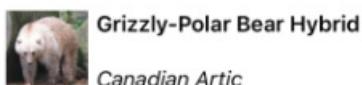
```

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 [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.

Specify CollectionView layout

9/20/2022 • 11 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [Control scrolling in a CollectionView](#).

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:

-  **Baboon**
Africa & Asia
-  **Capuchin Monkey**
Central & South America
-  **Blue Monkey**
Central and East Africa
-  **Squirrel Monkey**
Central & South America
-  **Golden Lion Tamarin**
Brazil
-  **Howler Monkey**
South America
-  **Japanese Macaque**
Japan
-  **Mandrill**
*Southern Cameroon, Gabon,
Equatorial Guinea, and Congo*

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:

	Baboon		Capuchin...
	Africa &...		Central...
	Blue Mo...		Squirrel...
	Central...		Central...
	Golden...		Howler...
	Brazil		South A...
	Japan...		Mandrill
	Japan		Souther...
	Probos...		Red-sh...
	Borneo		Vietnam...
	Gray-sh...		Golden...
	Vietnam		China
	Black S...		Tonkin...
	China		Vietnam

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:



Baboon

Africa & Asia



Go

Bra



Capuchin Monk...

Central & South...



Ho

Sol



Blue Monkey

Central and East...



Ja

Jap



Squirrel Monkey

Central & South...



Ma

Sol

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>
```

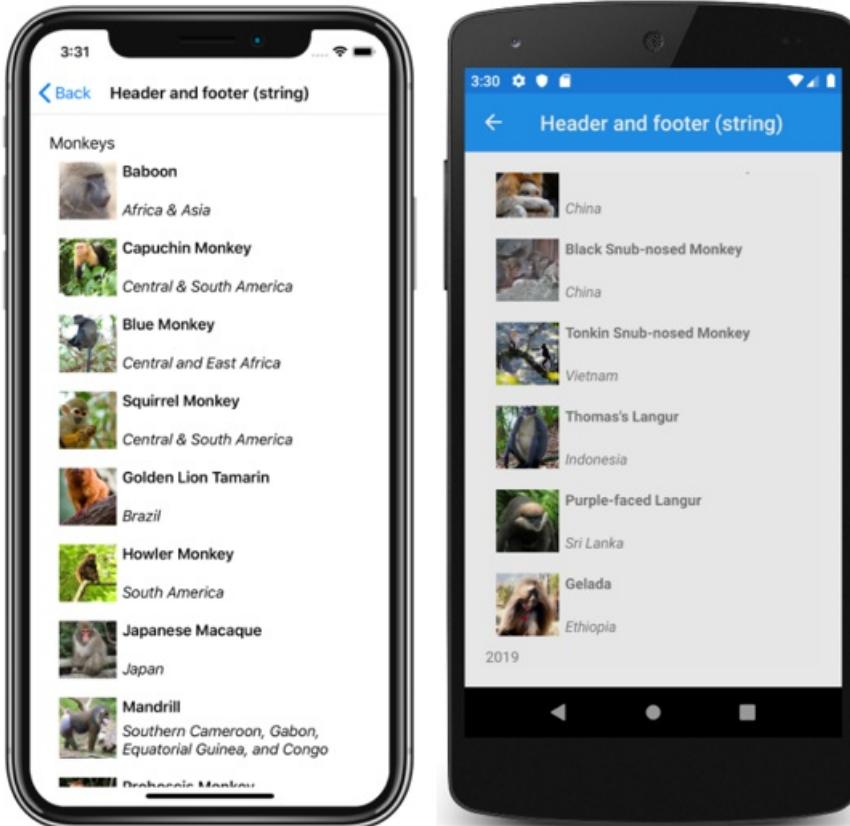
The equivalent C# code is:

```

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="12"
                  FontAttributes="Bold" />
        </StackLayout>
    </CollectionView.Header>
    <CollectionView.Footer>
        <StackLayout BackgroundColor="LightGray">
            <Label Margin="10,0,0,0"
                  Text="Friends of Xamarin Monkey"
                  FontSize="12"
                  FontAttributes="Bold" />
        </StackLayout>
    </CollectionView.Footer>
    ...
</CollectionView>

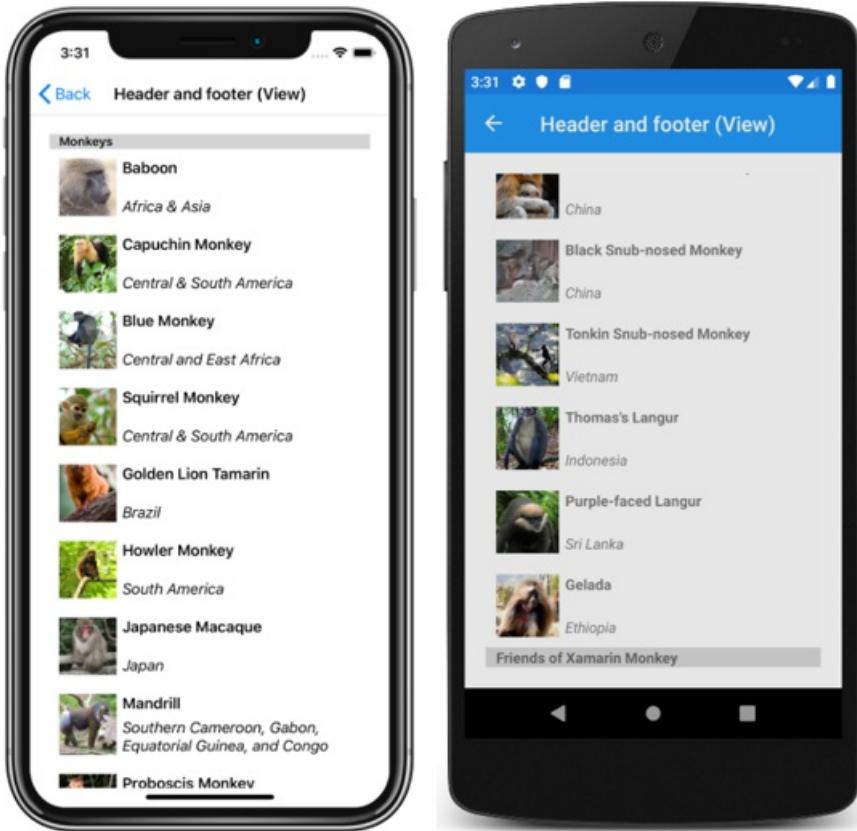
```

The equivalent C# code is:

```
StackLayout headerStackLayout = new StackLayout();
headerStackLayout.Add(new Label { Text = "Monkeys", ... } );
StackLayout footerStackLayout = new StackLayout();
footerStackLayout.Add(new Label { Text = "Friends of Xamarin Monkey", ... } );

CollectionView collectionView = new CollectionView
{
    Header = headerStackLayout,
    Footer = footerStackLayout
};
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="12"
                FontAttributes="Bold" />
        </StackLayout>
    </DataTemplate>
</CollectionView.HeaderTemplate>
<CollectionView.FooterTemplate>
    <DataTemplate>
        <StackLayout BackgroundColor="LightGray">
            <Label Margin="10,0,0,0"
                Text="Friends of Xamarin Monkey"
                FontSize="12"
                FontAttributes="Bold" />
        </StackLayout>
    </DataTemplate>
</CollectionView.FooterTemplate>
...
</CollectionView>

```

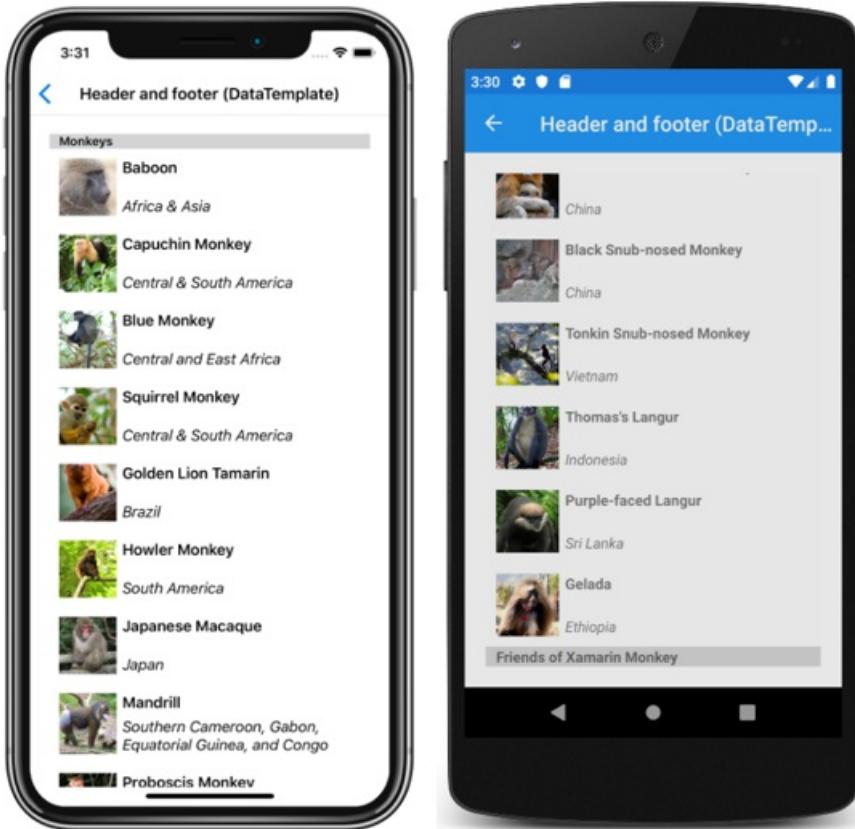
The equivalent C# code is:

```

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:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan

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 ensures 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:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan

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://schemas.microsoft.com/dotnet/2021/maui"
    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 screenshot:



Configure CollectionView item selection

9/20/2022 • 6 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 screenshot shows single item selection in a `CollectionView`:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

*Southern Cameroon, Gabon,
Equatorial Guinea, and Congo*

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:

```

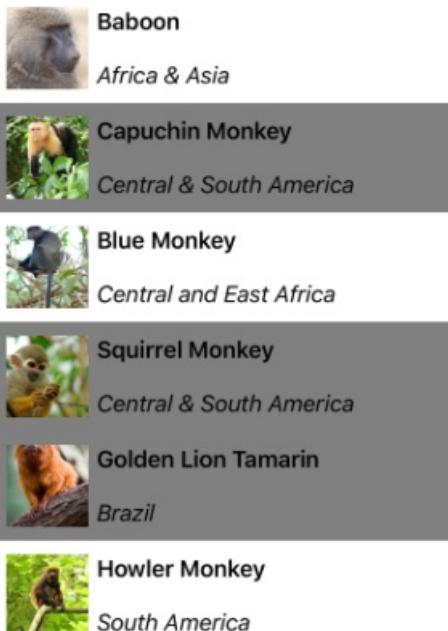
void OnCollectionViewSelectionChanged(object sender, SelectionChangedEventArgs e)
{
    var previous = e.PreviousSelection;
    var current = e.CurrentSelection;
    ...
}

```

IMPORTANT

The `SelectionChanged` event can be fired by changes that occur as a result of changing the `SelectionMode` property.

The following screenshot shows 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:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

*Southern Cameroon, Gabon,
Equatorial Guinea, and Congo*

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:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

Southern Cameroon, Gabon, Equatorial Guinea, and Congo

Clear selections

The `SelectedItem` and `SelectedItems` properties can be cleared by setting them, or the objects they bind to, to `null`. When either of these properties are cleared, the `SelectionChanged` event will be raised with an empty `CurrentSelection` property, and the `SelectionChangedCommand` will be executed.

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`:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

Southern Cameroon, Gabon,
Equatorial Guinea, and Congo

For more information about visual states, see [Visual states](#).

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.

Define an EmptyView for a CollectionView

9/20/2022 • 6 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 [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:

No items to display

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 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 .NET MAUI 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:

```

StackLayout stackLayout = new StackLayout();
stackLayout.Add(new Label { Text = "No results matched your filter.", ... } );
stackLayout.Add(new Label { Text = "Try a broader filter?", ... } );

SearchBar searchBar = new SearchBar { ... };
CollectionView collectionView = new CollectionView
{
    EmptyView = new ContentView
    {
        Content = stackLayout
    }
};
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:



No results matched your filter.

Try a broader filter?

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:



Your filter term of Xamarin did not match any records.

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://schemas.microsoft.com/dotnet/2021/maui"
    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>
                </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 [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:



No results matched your filter.

Try a broader filter?

For more information about data template selectors, see [Create a DataTemplateSelector](#).

Control scrolling in a CollectionView

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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

One `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

Another `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:



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

Southern Cameroon, Gabon, Equatorial Guinea, and Congo



Proboscis Monkey

Borneo

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:



Proboscis Monkey

Borneo



Red-shanked Douc

Vietnam, Laos



Gray-shanked Douc

Vietnam



Golden Snub-nosed Monkey

China



Black Snub-nosed Monkey

China



Tonkin Snub-nosed Monkey

Vietnam



Thomas's Langur

Indonesia

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:



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

*Southern Cameroon, Gabon,
Equatorial Guinea, and Congo*



Proboscis Monkey

Borneo



Red-shanked Douc

Vietnam, Laos



Gray-shanked Douc

Vietnam



Golden Snub-nosed Monkey

China

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:



Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

*Southern Cameroon, Gabon,
Equatorial Guinea, and Congo*



Proboscis Monkey

Borneo

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

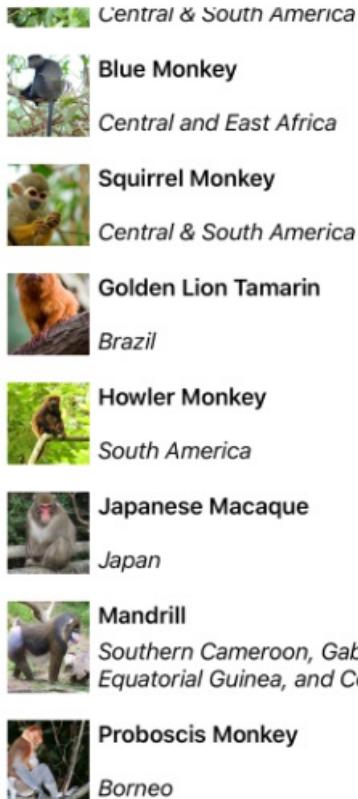
When snapping occurs, it will occur in the direction that produces the least amount of motion.

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 screenshot:



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>

```

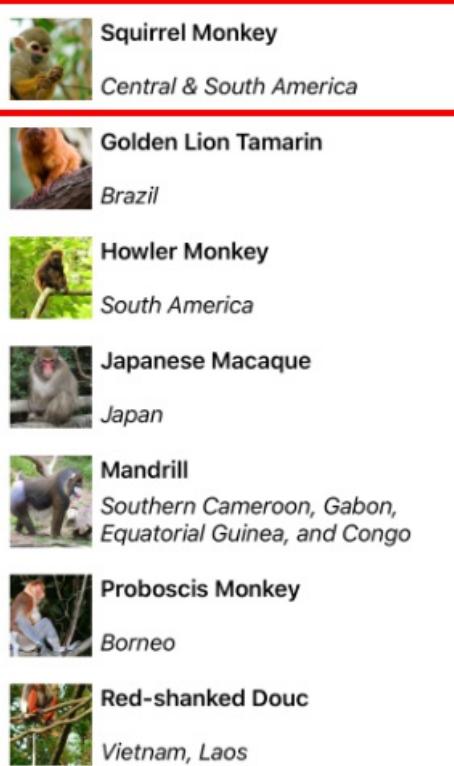
The equivalent C# code is:

```

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:

 **Central & South America**



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil



Howler Monkey

South America



Japanese Macaque

Japan



Mandrill

*Southern Cameroon, Gabon,
Equatorial Guinea, and Congo*



Proboscis Monkey

Borneo

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:



South America



Japanese Macaque

Japan



Mandrill

*Southern Cameroon, Gabon,
Equatorial Guinea, and Congo*



Proboscis Monkey

Borneo



Red-shanked Douc

Vietnam, Laos



Gray-shanked Douc

Vietnam



Golden Snub-nosed Monkey

China



Black Snub-nosed Monkey

China

Display grouped data in a CollectionView

9/20/2022 • 5 minutes to read • [Edit Online](#)

 [Browse 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.

The .NET Multi-platform App UI (.NET MAUI) `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 screenshot shows a `collectionView` displaying grouped data:



For more information about data templates, see [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 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

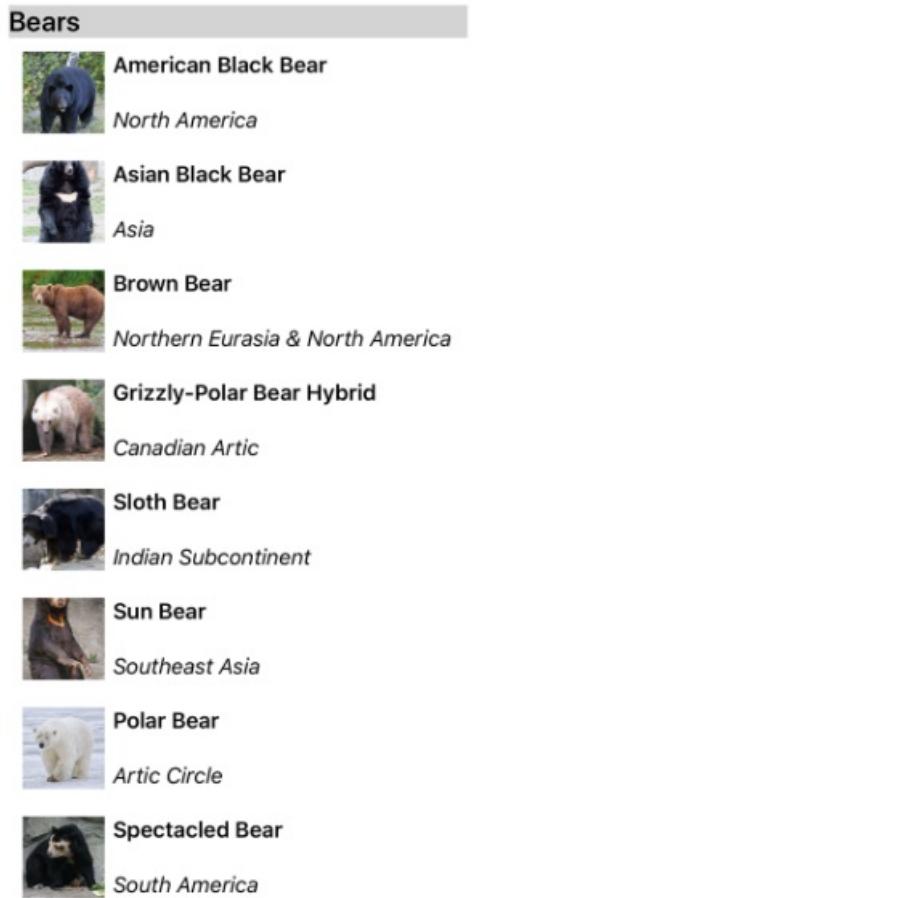
The appearance of each group header can be customized by setting the `CollectionView.GroupHeaderTemplate` property to a `DataTemplate`:

```

<CollectionView ItemsSource="{Binding Animals}"
    IsGrouped="true">
    ...
    <CollectionView.GroupHeaderTemplate>
        <DataTemplate>
            <Label Text="{Binding Name}"
                BackgroundColor="LightGray"
                FontSize="18"
                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 screenshot shows the customized group header:

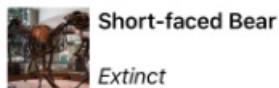


Customize the group footer

The appearance of each group footer can be customized by setting the `CollectionView.GroupFooterTemplate` property to a `DataTemplate`:

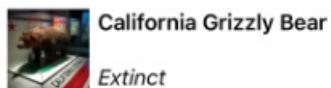
```
<CollectionView ItemsSource="{Binding Animals}"
    IsGrouped="true">
    ...
    <CollectionView.GroupFooterTemplate>
        <DataTemplate>
            <Label Text="{Binding Count, StringFormat='Total animals: {0:D}'}"
                Margin="0,0,0,10" />
        </DataTemplate>
    </CollectionView.GroupFooterTemplate>
</CollectionView>
```

In this example, each group footer is set to a `Label` that displays the number of items in the group. The following screenshot shows the customized group footer:



Short-faced Bear

Extinct

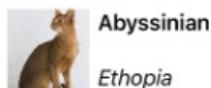


California Grizzly Bear

Extinct

Total animals: 10

Cats



Abyssinian

Ethopia



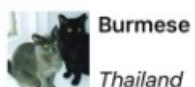
Arabian Mau

Arabian Peninsula



Bengal

Asia



Burmese

Thailand



Cyprus

Cyprus



German Rex

Germany

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 screenshot shows an empty group:

Aardvarks

Total animals: 0

Bears



American Black Bear

North America



Asian Black Bear

Asia



Brown Bear

Northern Eurasia & North America



Grizzly-Polar Bear Hybrid

Canadian Arctic



Sloth Bear

Indian Subcontinent



Sun Bear

Southeast Asia



Polar Bear

Arctic Circle

NOTE

On iOS 10, 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.

IndicatorView

9/20/2022 • 3 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `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 .NET MAUI, 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

To add indicators to a page, create an `IndicatorView` object and set its `IndicatorColor` and `SelectedIndicatorColor` properties. In addition, set the `CarouselView.IndicatorView` property to the name of the

`IndicatorView` object.

The following example shows how to create 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="ionicons"
                    FontSize="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 screenshot shows 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 `VisualState` 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="&#xf30c;">
                        <FontFamily>ionicons</FontFamily>
                        <FontSize>12</FontSize>
                        <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 [Visual states](#).

ListView

9/20/2022 • 27 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `ListView` displays a scrollable vertical list of selectable data items. While `ListView` manages the appearance of the list, the appearance of each item in the list is defined by a `DataTemplate` that uses a `cell` to display items. .NET MAUI includes cell types to display combinations of text and images, and you can also define custom cells that display any content you want. `ListView` also includes support for displaying headers and footers, grouped data, pull-to-refresh, and context menu items.

The `ListView` class derives from the `ItemsView<Cell>` class, from which it inherits the following properties:

- `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.

`ListView` defines the following properties:

- `Footer`, of type `object`, specifies the string 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`.
- `GroupHeaderTemplate`, of type `DataTemplate`, defines the `DataTemplate` used to define the appearance of the header of each group.
- `HasUnevenRows`, of type `bool`, indicates whether items in the list can have rows of different heights. The default value of this property is `false`.
- `Header`, of type `object`, specifies the string 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`.
- `HorizontalScrollBarVisibility`, of type `ScrollBarVisibility`, indicates when the horizontal scroll bar will be visible.
- `IsGroupedEnabled`, of type `bool`, indicates whether the underlying data should be displayed in groups. The default value of this property is `false`.
- `IsPullToRefreshEnabled`, of type `bool`, indicates whether the user can swipe down to cause the `ListView` to refresh its data. The default value of this property is `false`.
- `IsRefreshing`, of type `bool`, indicates whether the `ListView` is currently refreshing. The default value of this property is `false`.
- `RefreshCommand`, of type `ICommand`, represents the command that will be executed when a refresh is triggered.
- `RefreshControlColor`, of type `Color`, determines the color of the refresh visualization that's shown while a refresh occurs.
- `RowHeight`, of type `int`, determines the height of each row when `HasUnevenRows` is `false`.
- `SelectedItem`, of type `object`, represents the currently selected item in the `ListView`.
- `SelectionMode`, of type `ListViewSelectionMode`, indicates whether items can be selected in the `ListView` or not. The default value of this property is `Single`.
- `SeparatorColor`, of type `Color`, defines the color of the bar that separates items in the list.
- `SeparatorVisibility`, of type `SeparatorVisibility`, defines whether separators are visible between items.
- `VerticalScrollBarVisibility`, of type `ScrollBarVisibility`, indicates when the vertical scroll bar will be

visible.

All of these properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

In addition, `ListView` defines the following properties that aren't backed by `BindableProperty` objects:

- `GroupDisplayBinding`, of type `BindingBase`, the binding to use for displaying the group header.
- `GroupShortNameBinding`, of type `BindingBase`, the binding for the name to display in grouped jump lists.
- `CachingStrategy`, of type `ListViewCachingStrategy`, defines the cell reuse strategy of the `ListView`. This is a read-only property.

`ListView` defines the following events:

- `ItemAppearing`, which is raised when the visual representation of an item is being added to the visual layout of the `ListView`. The `ItemVisibilityEventArgs` object that accompanies this event defines `Item` and `Index` properties.
- `ItemDisappearing`, which is raised when the visual representation of an item is being removed from the visual layout of the `ListView`. The `ItemVisibilityEventArgs` object that accompanies this event defines `Item` and `Index` properties.
- `ItemSelected`, which is raised when a new item in the list is selected. The `SelectedItemChangedEventArgs` object that accompanies this event defines `SelectedItem` and `SelectedItemIndex` properties.
- `ItemTapped`, which is raised when an item in the `ListView` is tapped. The `ItemTappedEventArgs` object that accompanies this event defines `Group`, `Item`, and `ItemIndex` properties.
- `Refreshing`, which is raised when a pull to refresh operation is triggered on the `ListView`.
- `Scrolled`, . The `ScrolledEventArgs` object that accompanies this event defines `ScrollX` and `ScrollY` properties.
- `ScrollToRequested` . The `ScrollToRequestedEventArgs` object that accompanies this event defines `Element`, `Mode`, `Position`, `ScrollX`, `ScrollY`, and `ShouldAnimate` properties.

Populate a ListView with data

A `ListView` is populated with data by setting its `ItemsSource` property to any collection that implements `IEnumerable`.

IMPORTANT

If the `ListView` 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`.

`ListView` 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:

```
<ListView ItemsSource="{Binding Monkeys}" />
```

The equivalent C# code is:

```
ListView listView = new ListView();
listView.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 .NET MAUI applications. For more information, see [Compiled bindings](#).

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

Define item appearance

The appearance of each item in the `ListView` can be defined by setting the `ItemTemplate` property to a `DataTemplate`:

```
<ListView ItemsSource="{Binding Monkeys}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <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>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

The elements specified in the `DataTemplate` define the appearance of each item in the list, and the child of the `DataTemplate` must be a `cell` object. 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 screenshot shows the result of templating each item in the list:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil

For more information about data templates, see [Data templates](#).

Cells

The appearance of each item in a `ListView` is defined by a `DataTemplate`, and the `DataTemplate` must reference a `Cell` class to display items. Each cell represents an item of data in the `ListView`. .NET MAUI includes the following built-in cells:

- `TextCell`, which displays primary and secondary text on separate lines.
- `ImageCell`, which displays an image with primary and secondary text on separate lines.
- `SwitchCell`, which displays text and a switch that can be switched on or off.
- `EntryCell`, which displays a label and text that's editable.
- `ViewCell`, which is a custom cell whose appearance is defined by a `View`. This cell type should be used when you want to fully define the appearance of each item in a `ListView`.

Typically, `SwitchCell` and `EntryCell` will only be used in a `TableView` and won't be used in a `ListView`. For more information about `switchCell` and `EntryCell`, see [TableView](#).

Text cell

A `TextCell` displays primary and secondary text on separate lines. `TextCell` defines the following properties:

- `Text`, of type `string`, defines the primary text to be displayed.
- `TextColor`, of type `Color`, represents the color of the primary text.
- `Detail`, of type `string`, defines the secondary text to be displayed.
- `DetailColor`, of type `color`, indicates the color of the secondary text.
- `Command`, of type `ICommand`, defines the command that's executed when the cell is tapped.
- `CommandParameter`, of type `object`, represents 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 following example shows using a `TextCell` to define the appearance of items in a `ListView`:

```

<ListView ItemsSource="{Binding Food}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextCell Text="{Binding Name}"
                      Detail="{Binding Description}" />
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>

```

The following screenshot shows the resulting cell appearance:



Image cell

An `ImageCell` displays an image with primary and secondary text on separate lines. `ImageCell` inherits the properties from `TextCell`, and defines the `ImageSource` property, of type `ImageSource`, which specifies the image to be displayed in the cell. This property is backed by a `BindableProperty` object, which means it can be the target of data bindings, and be styled.

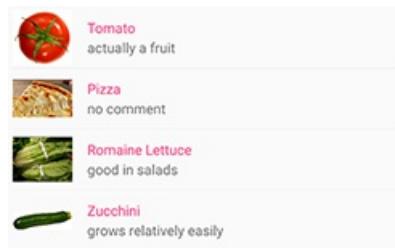
The following example shows using an `ImageCell` to define the appearance of items in a `ListView`:

```

<ListView ItemsSource="{Binding Food}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ImageCell ImageSource="{Binding Image}"
                       Text="{Binding Name}"
                       Detail="{Binding Description}" />
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>

```

The following screenshot shows the resulting cell appearance:



View cell

A `ViewCell` is a custom cell whose appearance is defined by a `View`. `ViewCell` defines a `View` property, of type `View`, which defines the view that represents the content of the cell. This property is backed by a `BindableProperty` object, which means it can be the target of data bindings, and be styled.

NOTE

The `View` property is the content property of the `ViewCell` class, and therefore does not need to be explicitly set from XAML.

The following example shows using a `ViewCell` to define the appearance of items in a `ListView`:

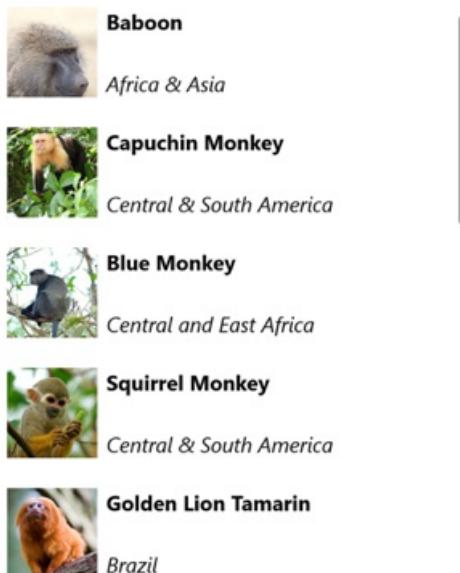
```

<ListView ItemsSource="{Binding Monkeys}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <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>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>

```

Inside the `ViewCell`, layout can be managed by any .NET MAUI layout. In this example, layout is managed by a `Grid`. The `Grid` contains an `Image` object, and two `Label` objects, that all bind to properties of the `Monkey` class.

The following screenshot shows the result of templating each item in the list:



Choose item appearance at runtime

The appearance of each item in the `ListView` can be chosen at runtime, based on the item value, by setting the `ItemTemplate` property to a `DataTemplateSelector` object:

```
<ContentPage ...>
    xmlns:templates="clr-namespace:ListViewDemos.Templates">
<ContentPage.Resources>
    <DataTemplate x:Key="AmericanMonkeyTemplate">
        <ViewCell>
            ...
        </ViewCell>
    </DataTemplate>

    <DataTemplate x:Key="OtherMonkeyTemplate">
        <ViewCell>
            ...
        </ViewCell>
    </DataTemplate>

    <templates:MonkeyDataTemplateSelector x:Key="MonkeySelector"
        AmericanMonkey="{StaticResource AmericanMonkeyTemplate}"
        OtherMonkey="{StaticResource OtherMonkeyTemplate}" />
</ContentPage.Resources>

<StackLayout Margin="20">
    <ListView ItemsSource="{Binding Monkeys}"
        ItemTemplate="{StaticResource MonkeySelector}" />
</StackLayout>
</ContentPage>
```

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:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America



Golden Lion Tamarin

Brazil

For more information about data template selectors, see [Create a DataTemplateSelector](#).

Respond to item selection

By default, `ListView` selection is enabled. However, this behavior can be changed by setting the `SelectionMode` property. The `ListViewSelectionMode` enumeration defines the following members:

- `None` – indicates that items cannot be selected.
- `Single` – indicates that a single item can be selected, with the selected item being highlighted. This is the default value.

`ListView` defines an `ItemSelected` event that's raised when the `SelectedItem` property changes, either due to the user selecting an item from the list, or when an app sets the property. The `SelectedItemChangedEventArgs` object that accompanies this event has `SelectedItem` and `SelectedItemIndex` properties.

When the `SelectionMode` property is set to `Single`, a single item in the `ListView` 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 `ItemSelected` event is raised.

The following example shows a `ListView` that can respond to single item selection:

```
<ListView ItemsSource="{Binding Monkeys}"
          ItemSelected="OnItemSelected"
          ...
</ListView>
```

In this example, the `OnItemSelected` event handler is executed when the `ItemSelected` event fires, with the event handler retrieving the selected item:

```
void OnItemSelected(object sender, SelectedItemChangedEventArgs args)
{
    Monkey item = args.SelectedItem as Monkey;
}
```

The following screenshot shows single item selection in a `ListView`:



Baboon

Africa & Asia



Capuchin Monkey

Central & South America



Blue Monkey

Central and East Africa



Squirrel Monkey

Central & South America

Clear the selection

The `SelectedItem` property can be cleared by setting it, or the object it binds to, to `null`.

Disable selection

`ListView` selection is enabled by default. However, it can be disabled by setting the `SelectionMode` property to `None`:

```
<ListView ...  
    SelectionMode="None" />
```

When the `SelectionMode` property is set to `None`, items in the `ListView` cannot be selected, the `SelectedItem` property will remain `null`, and the `ItemSelected` event will not be fired.

Cache data

`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.

A `ListView` is 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 app 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 app from instantiating thousands of objects, saving time and memory.

.NET MAUI permits `ListView` cell reuse through the `ListViewCachingStrategy` enumeration, which defines the following members:

- `RetainElement`, specifies that the `ListView` will generate a cell for each item in the list.
- `RecycleElement`, specifies that the `ListView` will attempt to minimize its memory footprint and execution speed by recycling list cells.
- `RecycleElementAndDataTemplate`, as `RecycleElement` while also ensuring that when a `ListView` uses a `DataTemplateSelector`, `DataTemplate` objects are cached by the type of item in the list.

Retain elements

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` objects, become performance bottlenecks when they're set up and destroyed in real time as the user scrolls.

Recycle elements

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 app 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 shown in the following 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;
        }
    }
}
```

Recycle elements with a `DataTemplateSelector`

When a `ListView` uses a `DataTemplateSelector` to select a `DataTemplate`, the `RecycleElement` caching strategy does not cache `DataTemplate` objects. Instead, a `DataTemplate` is selected for each item of data in the list.

NOTE

The `RecycleElement` caching strategy requires 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.

Recycle elements with DataTemplates

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` objects are cached by the type of item in the list. Therefore, `DataTemplate` objects are selected once per item type, instead of once per item instance.

NOTE

The `RecycleElementAndDataTemplate` caching strategy requires that `DataTemplate` objects returned by the `DataTemplateSelector` must use the `DataTemplate` constructor that takes a `Type`.

Set the caching strategy

The `ListView` caching strategy can be defined by in XAML by setting the `CachingStrategy` attribute:

```
<ListView CachingStrategy="RecycleElement">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                ...
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
```

In C#, the caching strategy is set via a constructor overload:

```
ListView listView = new ListView(ListViewCachingStrategy.RecycleElement);
```

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`. The solution to this issue is to specify a constructor on the subclassed `ListView` that accepts a `ListViewCachingStrategy` parameter and passes it to 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` attribute:

```
<local:CustomListView>
    <x:Arguments>
        <ListViewCachingStrategy>RecycleElement</ListViewCachingStrategy>
    </x:Arguments>
</local:CustomListView>
```

Headers and footers

`ListView` 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.

`ListView` 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.

Display strings in the header and footer

The `Header` and `Footer` properties can be set to `string` values, as shown in the following example:

```
<ListView ItemsSource="{Binding Monkeys}"
    Header="Monkeys"
    Footer="2022">
    ...
</ListView>
```

The following screenshot shows the resulting header:



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:

```

<ListView ItemsSource="{Binding Monkeys}">
    <ListView.Header>
        <StackLayout BackgroundColor="LightGray">
            <Label Margin="10,0,0,0"
                  Text="Monkeys"
                  FontSize="12"
                  FontAttributes="Bold" />
        </StackLayout>
    </ListView.Header>
    <ListView.Footer>
        <StackLayout BackgroundColor="LightGray">
            <Label Margin="10,0,0,0"
                  Text="Friends of Monkey"
                  FontSize="12"
                  FontAttributes="Bold" />
        </StackLayout>
    </ListView.Footer>
    ...
</ListView>

```

The following screenshot shows the resulting header:



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:

```

<ListView ItemsSource="{Binding Monkeys}"
          Header="{Binding .}"
          Footer="{Binding .}">
    <ListView.HeaderTemplate>
        <DataTemplate>
            <StackLayout BackgroundColor="LightGray">
                <Label Margin="10,0,0,0"
                      Text="Monkeys"
                      FontSize="12"
                      FontAttributes="Bold" />
            </StackLayout>
        </DataTemplate>
    </ListView.HeaderTemplate>
    <ListView.FooterTemplate>
        <DataTemplate>
            <StackLayout BackgroundColor="LightGray">
                <Label Margin="10,0,0,0"
                      Text="Friends of Monkey"
                      FontSize="12"
                      FontAttributes="Bold" />
            </StackLayout>
        </DataTemplate>
    </ListView.FooterTemplate>
    ...
</ListView>

```

Control item separators

By default, separators are displayed between `ListView` items on iOS and Android. This behavior can be changed

by setting the `SeparatorVisibility` property, of type `SeparatorVisibility`, to `None`:

```
<ListView ...  
    SeparatorVisibility="None" />
```

In addition, when the separator is enabled, its color can be set with the `SeparatorColor` property:

```
<ListView ...  
    SeparatorColor="Blue" />
```

Size items

By default, all items in a `ListView` have the same height, which is derived from the contents of the `DataTemplate` that defines the appearance of each item. However, this behavior can be changed with the `HasUnevenRows` and `RowHeight` properties. By default, the `HasUnevenRows` property is `false`.

The `RowHeight` property can be set to an `int` that represents the height of each item in the `ListView`, provided that `HasUnevenRows` is `false`. When `HasUnevenRows` is set to `true`, each item in the `ListView` can have a different height. The height of each item will be derived from the contents of the item's `DataTemplate`, and so each item will be sized to its content.

Individual `Listview` items can be programmatically resized at runtime by changing layout related properties of elements within the `DataTemplate`, provided that the `HasUnevenRows` property is `true`. The following example changes the height of an `Image` object when it's tapped:

```
void OnImageTapped(object sender, EventArgs args)  
{  
    Image image = sender as Image;  
    ViewCell viewCell = image.Parent.Parent as ViewCell;  
  
    if (image.HeightRequest < 250)  
    {  
        image.HeightRequest = image.Height + 100;  
        viewCell.ForceUpdateSize();  
    }  
}
```

In this example, the `OnImageTapped` event handler is executed in response to an `Image` object being tapped. The event handler updates the height of the `Image` and the `Cell.ForceUpdateSize` method updates the cell's size, even when it isn't currently visible.

WARNING

Overuse of dynamic item sizing can cause `ListView` performance to degrade.

Right-to-left layout

`ListView` 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="ListViewDemos.RightToLeftListPage"
    Title="Right to left list"
    FlowDirection="RightToLeft">
    <StackLayout Margin="20">
        <ListView ItemsSource="{Binding Monkeys}">
            ...
        </ListView>
    </StackLayout>
</ContentPage>

```

The default `FlowDirection` for an element with a parent is `MatchParent`. Therefore, the `ListView` inherits the `FlowDirection` property value from the `StackLayout`, which in turn inherits the `FlowDirection` property value from the `ContentPage`.

Display grouped data

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.

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 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:

```

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:

```
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.

`ListView` will display grouped data, provided that the data has been grouped correctly, by setting the `IsGroupingEnabled` property to `true`:

```

<ListView ItemsSource="{Binding Animals}"
          IsGroupingEnabled="True">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <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>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>

```

The equivalent C# code is:

```

ListView listView = new ListView
{
    IsGroupingEnabled = true
};
listView.SetBinding(ItemsView.ItemsSourceProperty, "Animals");
// ...

```

The appearance of each item in the `ListView` is defined by setting its `ItemTemplate` property to a `DataTemplate`. For more information, see [Define item appearance](#).

The following screenshot shows the `ListView` displaying grouped data:



NOTE

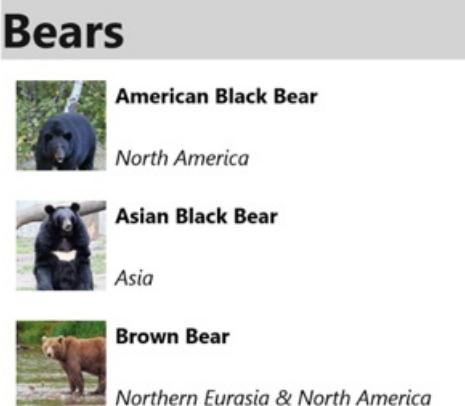
By default, `ListView` will display the group name in the group header. This behavior can be changed by customizing the group header.

Customize the group header

The appearance of each group header can be customized by setting the `ListView.GroupHeaderTemplate` property to a `DataTemplate`:

```
<ListView ItemsSource="{Binding Animals}"
    IsGroupingEnabled="True">
    <ListView.GroupHeaderTemplate>
        <DataTemplate>
            <ViewCell>
                <Label Text="{Binding Name}"
                    BackgroundColor="LightGray"
                    FontSize="18"
                    FontAttributes="Bold" />
            </ViewCell>
        </DataTemplate>
    </ListView.GroupHeaderTemplate>
    ...
</ListView>
```

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 screenshot shows the customized group header:



Group without templates

`ListView` can display correctly grouped data without setting the `ItemTemplate` property to a `DataTemplate`:

```
<ListView ItemsSource="{Binding Animals}"
    IsGroupingEnabled="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.

Control scrolling

`ListView` defines two `ScrollTo` methods, that scroll items into view. One of the overloads scrolls the specified item into view, while the other scrolls the specified item in the specified group into view. Both overloads have additional arguments that allow the exact position of the item after the scroll has completed to be specified, and whether to animate the scroll.

`ListView` 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 `ShouldAnimate`, `Element`, `Mode`, and `Position`. Some of these properties are set from the arguments specified in the `ScrollTo` method calls.

In addition, `ListView` 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.

Detect scrolling

`ListView` 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:

- `ScrollX`, of type `double`, represents the X position of the scroll
- `ScrollY`, of type `double`, represents the Y position of the scroll.

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)
{
    // Custom logic
}
```

IMPORTANT

The `Scrolled` event is fired for user initiated scrolls, and for programmatic scrolls.

Scroll an item into view

The `ScrollTo` method scrolls the specified item into view. Given a `ListView` object named `listView`, 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");
listView.ScrollTo(monkey, ScrollToPosition.MakeVisible, true);
```

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");
listView.ScrollTo(monkey, group, ScrollToPosition.MakeVisible, true);
```

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 `animated` argument of the `ScrollTo` method to `false`:

```
listView.ScrollTo(monkey, position: ScrollToPosition.MakeVisible, 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:

```
listView.ScrollTo(monkey, position: ScrollToPosition.MakeVisible, animate: true);
```

Start

The `ScrollToPosition.Start` member indicates that the item should be scrolled to the start of the view:

```
listView.ScrollTo(monkey, position: ScrollToPosition.Start, animate: true);
```

Center

The `ScrollToPosition.Center` member indicates that the item should be scrolled to the center of the view:

```
listView.ScrollTo(monkey, position: ScrollToPosition.Center, animate: true);
```

End

The `ScrollToPosition.End` member indicates that the item should be scrolled to the end of the view:

```
listView.ScrollTo(monkey, position: ScrollToPosition.End, animate: true);
```

Scroll bar visibility

`ListView` 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.

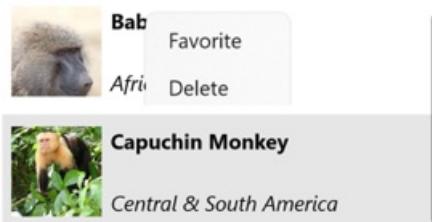
Add context menus

`ListView` supports context menus items, which are defined as `MenuItem` objects that are added to the `ViewCell.ContextActions` collection in the `DataTemplate` for each item:

```
<ListView x:Name="listView"
          ItemsSource="{Binding Monkeys}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <ViewCell.ContextActions>
                    <MenuItem Text="Favorite"
                              Command="{Binding Source={x:Reference listView},
Path=BindingContext.FavoriteCommand}"
                              CommandParameter="{Binding}" />
                    <MenuItem Text="Delete"
                              Command="{Binding Source={x:Reference listView},
Path=BindingContext.DeleteCommand}"
                              CommandParameter="{Binding}" />
                </ViewCell.ContextActions>

                ...
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

The `MenuItem` objects are revealed when an item in the `ListView` is right-clicked:



Pull to refresh

`ListView` supports pull to refresh functionality, which enables the data being displayed to be refreshed by pulling down on the list of items.

To enable pull to refresh, set the `IsPullToRefreshEnabled` property to `true`. When a refresh is triggered, `ListView` raises the `Refreshing` event, and the `IsRefreshing` property will be set to `true`. The code required to refresh the contents of the `ListView` should then be executed by the handler for the `Refreshing` event, or by the `ICommand` implementation the `RefreshCommand` executes. Once the `ListView` is refreshed, the `IsRefreshing` property should be set to `false`, or the `EndRefresh` method should be called on the `ListView`, to indicate that the refresh is complete.

The following example shows a `ListView` that uses pull to refresh:

```
<ListView ItemsSource="{Binding Animals}"
    IsPullToRefreshEnabled="true"
    RefreshCommand="{Binding RefreshCommand}"
    IsRefreshing="{Binding IsRefreshing}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                ...
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

In this example, when the user initiates a refresh, the `ICommand` defined by the `RefreshCommand` 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 `IsRefreshing` property indicates the current state of the refresh operation. When a refresh is triggered, this property will automatically transition to `true`. Once the refresh completes, you should reset the property to `false`.

Picker

9/20/2022 • 7 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Picker` displays a short list of items, from which the user can select an item.

`Picker` defines the following properties:

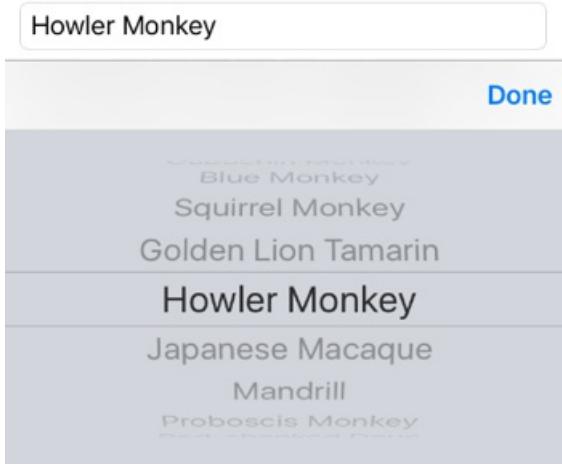
- `CharacterSpacing`, of type `double`, is the spacing between characters of the item displayed by the `Picker`.
- `FontAttributes` of type `FontAttributes`, which defaults to `FontAttributes.None`.
- `FontAutoScalingEnabled`, of type `bool`, which determines whether the text respects scaling preferences set in the operating system. The default value of this property is `true`.
- `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.
- `TextTransform`, of type `TextTransform`, which defines whether to transform the casing of text.
- `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) pattern. 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, as shown in the following iOS screenshot:



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`:

Howler Monkey

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 adding data to a `Picker`. For more information, see [Set the ItemsSource property](#).
- Adding the data to be displayed to the `Items` collection. For more information, see [Add data to the Items collection](#).

Set the `ItemsSource` property

A `Picker` can be populated with data by setting its `ItemsSource` property to an `IList` collection. Each item in the collection must be of, or derived from, type `object`. Items can be added in XAML by initializing the `ItemsSource` property from an array of items:

```
<Picker x:Name="picker"
        Title="Select a monkey">
    <Picker.ItemsSource>
        <x:Array Type="{x:Type x:String}">
            <x:String>Baboon</x:String>
            <x:String>Capuchin Monkey</x:String>
            <x:String>Blue Monkey</x:String>
            <x:String>Squirrel Monkey</x:String>
            <x:String>Golden Lion Tamarin</x:String>
            <x:String>Howler Monkey</x:String>
            <x:String>Japanese Macaque</x:String>
        </x:Array>
    </Picker.ItemsSource>
</Picker>
```

NOTE

The `x:Array` element requires a `Type` attribute indicating the type of the items in the array.

The equivalent C# code is:

```

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");

Picker picker = new Picker { Title = "Select a monkey" };
picker.ItemsSource = monkeyList;

```

Respond 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.

The following XAML example shows how to retrieve the `SelectedItem` property value from the `Picker`:

```
<Label Text="{Binding Source={x:Reference picker}, Path=SelectedItem}" />
```

The equivalent C# code is:

```

Label 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];
    }
}

```

In this example, the event handler 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. 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.

Populate 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"
    ItemsSource="{Binding Monkeys}"
    ItemDisplayBinding="{Binding Name}" />
```

The equivalent C# code is shown below:

```
Picker picker = new Picker { Title = "Select a monkey" };
picker.SetBinding(Picker.ItemsSourceProperty, "Monkeys");
picker.ItemDisplayBinding = new Binding("Name");
```

In this example, the `ItemsSource` property data binds to the `Monkeys` property of the binding context, 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.

Respond to item selection

Data binding can be used to set an object to the `SelectedItem` property value when it changes:

```
<Picker Title="Select a monkey"
    ItemsSource="{Binding Monkeys}"
    ItemDisplayBinding="{Binding Name}"
    SelectedItem="{Binding SelectedMonkey}" />
<Label Text="{Binding SelectedMonkey.Name}" ... />
<Label Text="{Binding SelectedMonkey.Location}" ... />
<Image Source="{Binding SelectedMonkey.ImageUrl}" ... />
<Label Text="{Binding SelectedMonkey.Details}" ... />
```

The equivalent C# code is:

```
Picker picker = new Picker { Title = "Select a monkey" };
picker.SetBinding(Picker.ItemsSourceProperty, "Monkeys");
picker.SetBinding(Picker.SelectedItemProperty, "SelectedMonkey");
picker.ItemDisplayBinding = new Binding("Name");

Label nameLabel = new Label { ... };
nameLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Name");

Label locationLabel = new Label { ... };
locationLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Location");

Image image = new Image { ... };
image.SetBinding(Image.SourceProperty, "SelectedMonkey.ImageUrl");

Label detailsLabel = new Label();
detailsLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Details");
```

The `SelectedItem` property data binds to the `SelectedMonkey` property of the binding context, 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

The `SelectedItem` and `SelectedIndex` properties both support two-way bindings by default.

Add data to the Items collection

An alternative process for populating a `Picker` with data is to add the data to be displayed to the read-only `Items` collection, which is of type `IList<string>`. Each item in the collection must be of type `string`. Items can be added in XAML by initializing the `Items` property with a list of `x:String` items:

```
<Picker Title="Select a monkey">
  <Picker.Items>
    <x:String>Baboon</x:String>
    <x:String>Capuchin Monkey</x:String>
    <x:String>Blue Monkey</x:String>
    <x:String>Squirrel Monkey</x:String>
    <x:String>Golden Lion Tamarin</x:String>
    <x:String>Howler Monkey</x:String>
    <x:String>Japanese Macaque</x:String>
  </Picker.Items>
</Picker>
```

The equivalent C# code is:

```
Picker picker = new Picker { Title = "Select a monkey" };
picker.Items.Add("Baboon");
picker.Items.Add("Capuchin Monkey");
picker.Items.Add("Blue Monkey");
picker.Items.Add("Squirrel Monkey");
picker.Items.Add("Golden Lion Tamarin");
picker.Items.Add("Howler Monkey");
picker.Items.Add("Japanese Macaque");
```

In addition to adding data using the `Items.Add` method, data can also be inserted into the collection by using the `Items.Insert` method.

Respond 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.

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.

TableView

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The .NET Multi-platform App UI (.NET MAUI) `TableView` displays a table of scrollable items that can be grouped into sections. A `TableView` is typically used for displaying items where each row has a different appearance, such as presenting a table of settings.

While `TableView` manages the appearance of the table, the appearance of each item in the table is defined by a `Cell`. .NET MAUI includes five cell types that are used to display different combinations of data, and you can also define custom cells that display any content you want.

`TableView` defines the following properties:

- `Intent`, of type `TableIntent`, defines the purpose of the table on iOS.
- `HasUnevenRows`, of type `bool`, indicates whether items in the table can have rows of different heights. The default value of this property is `false`.
- `Root`, of type `TableRoot`, defines the child of the `TableView`.
- `RowHeight`, of type `int`, determines the height of each row when `HasUnevenRows` is `false`.

The `HasUnevenRows` and `RowHeight` properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The value of the `Intent` property helps to define the `TableView` appearance on iOS only. This property should be set to a value of the `TableIntent` enumeration, which defines the following members:

- `Menu`, for presenting a selectable menu.
- `Settings`, for presenting a table of configuration settings.
- `Form`, for presenting a data input form.
- `Data`, for presenting data.

Create a TableView

To create a table, create a `TableView` object and set its `Intent` property to a `TableIntent` member. The child of a `TableView` must be a `TableRoot` object, which is parent to one or more `TableSection` objects. Each `TableSection` consists of an optional title whose color can also be set, and one or more `Cell` objects.

The following example shows how to create a `TableView`:

```

<TableView Intent="Menu">
    <TableRoot>
        <TableSection Title="Chapters">
            <TextCell Text="1. Introduction to .NET MAUI"
                Detail="Learn about .NET MAUI and what it provides." />
            <TextCell Text="2. Anatomy of an app"
                Detail="Learn about the visual elements in .NET MAUI" />
            <TextCell Text="3. Text"
                Detail="Learn about the .NET MAUI controls that display text." />
            <TextCell Text="4. Dealing with sizes"
                Detail="Learn how to size .NET MAUI controls on screen." />
            <TextCell Text="5. XAML vs code"
                Detail="Learn more about creating your UI in XAML." />
        </TableSection>
    </TableRoot>
</TableView>

```

In this example, the `TableView` defines a menu using `TextCell` objects:

Chapters

1. Introduction to .NET MAUI

Learn about .NET MAUI and what it provides.

2. Anatomy of an app

Learn about the visual elements in .NET MAUI

3. Text

Learn about the .NET MAUI controls that display text.

4. Dealing with sizes

Learn how to size .NET MAUI controls on screen.

5. XAML vs code

Learn more about creating your UI in XAML.

NOTE

Each `TextCell` can execute a command when tapped, provided that the `Command` property is set to a valid `ICommand` implementation.

Define cell appearance

Each item in a `TableView` is defined by a `Cell` object, and the `Cell` type used defines the appearance of the cell's data. .NET MAUI includes the following built-in cells:

- `TextCell`, which displays primary and secondary text on separate lines.
- `ImageCell`, which displays an image with primary and secondary text on separate lines.
- `SwitchCell`, which displays text and a switch that can be switched on or off.
- `EntryCell`, which displays a label and text that's editable.
- `ViewCell`, which is a custom cell whose appearance is defined by a `View`. This cell type should be used when you want to fully define the appearance of each item in a `TableView`.

Text cell

A `TextCell` displays primary and secondary text on separate lines. `TextCell` defines the following properties:

- `Text`, of type `string`, defines the primary text to be displayed.
- `TextColor`, of type `Color`, represents the color of the primary text.
- `Detail`, of type `string`, defines the secondary text to be displayed.
- `DetailColor`, of type `Color`, indicates the color of the secondary text.
- `Command`, of type `ICommand`, defines the command that's executed when the cell is tapped.

- `CommandParameter`, of type `object`, represents 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 following example shows using a `TextCell` to define the appearance of items in a `TableView`:

```
<TableView Intent="Menu">
    <TableRoot>
        <TableSection Title="Chapters">
            <TextCell Text="1. Introduction to .NET MAUI"
                      Detail="Learn about .NET MAUI and what it provides." />
            <TextCell Text="2. Anatomy of an app"
                      Detail="Learn about the visual elements in .NET MAUI" />
            <TextCell Text="3. Text"
                      Detail="Learn about the .NET MAUI controls that display text." />
            <TextCell Text="4. Dealing with sizes"
                      Detail="Learn how to size .NET MAUI controls on screen." />
            <TextCell Text="5. XAML vs code"
                      Detail="Learn more about creating your UI in XAML." />
        </TableSection>
    </TableRoot>
</TableView>
```

The following screenshot shows the resulting cell appearance:

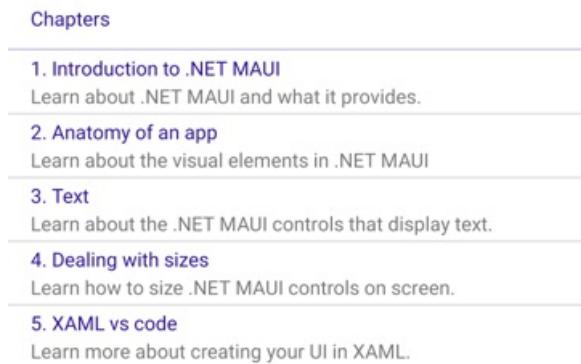


Image cell

An `ImageCell` displays an image with primary and secondary text on separate lines. `ImageCell` inherits the properties from `TextCell`, and defines the `ImageSource` property, of type `ImageSource`, which specifies the image to be displayed in the cell. This property is backed by a `BindableProperty` object, which means it can be the target of data bindings, and be styled.

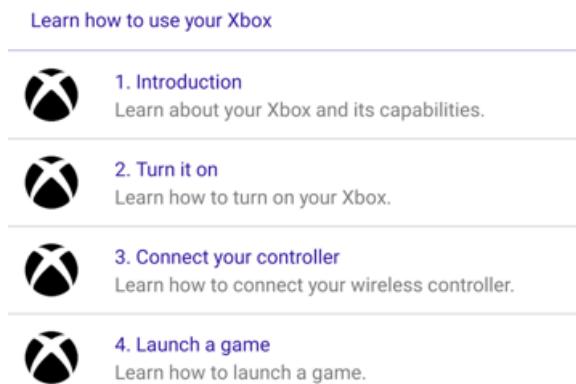
The following example shows using an `ImageCell` to define the appearance of items in a `TableView`:

```

<TableView Intent="Menu">
    <TableRoot>
        <TableSection Title="Learn how to use your XBox">
            <ImageCell Text="1. Introduction"
                Detail="Learn about your XBox and its capabilities."
                ImageSource="xbox.png" />
            <ImageCell Text="2. Turn it on"
                Detail="Learn how to turn on your XBox."
                ImageSource="xbox.png" />
            <ImageCell Text="3. Connect your controller"
                Detail="Learn how to connect your wireless controller."
                ImageSource="xbox.png" />
            <ImageCell Text="4. Launch a game"
                Detail="Learn how to launch a game."
                ImageSource="xbox.png" />
        </TableSection>
    </TableRoot>
</TableView>

```

The following screenshot shows the resulting cell appearance:



Switch cell

A `SwitchCell` displays text and a switch that can be switched on or off. `SwitchCell` defines the following properties:

- `Text`, of type `string`, defines the text to display next to the switch.
- `On`, of type `bool`, represents whether the switch is on or off.
- `OnColor`, of type `Color`, indicates the color of the switch when in its on position.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

`SwitchCell` also defines an `OnChanged` event that's raised when the switch changes state. The `ToggledEventArgs` object that accompanies this event defines a `value` property, that indicates whether the switch is on or off.

The following example shows using a `SwitchCell` to define the appearance of items in a `TableView`:

```

<TableView Intent="Settings">
    <TableRoot>
        <TableSection>
            <SwitchCell Text="Airplane Mode"
                On="False" />
            <SwitchCell Text="Notifications"
                On="True" />
        </TableSection>
    </TableRoot>
</TableView>

```

The following screenshot shows the resulting cell appearance:



Entry cell

An `EntryCell` displays a label and text data that's editable. `EntryCell` defines the following properties:

- `HorizontalTextAlignment`, of type `TextAlignment`, represents the horizontal alignment of the text.
- `Keyboard`, of type `Keyboard`, determines the keyboard to display when entering text.
- `Label`, of type `string`, represents the text to display to the left of the editable text.
- `LabelColor`, of type `Color`, defines the color of the label text.
- `Placeholder`, of type `string`, represents the text that's displayed when the `Text` property is empty.
- `Text`, of type `string`, defines the text that's editable.
- `VerticalTextAlignment`, of type `TextAlignment`, represents the vertical alignment of the text.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

`EntryCell` also defines a `Completed` event that's raised when the user hits the return key, to indicate that editing is complete.

The following example shows using an `EntryCell` to define the appearance of items in a `TableView`:

```
<TableView Intent="Settings">
    <TableRoot>
        <TableSection>
            <EntryCell Label="Login"
                Placeholder="username" />
            <EntryCell Label="Password"
                Placeholder="password" />
        </TableSection>
    </TableRoot>
</TableView>
```

The following screenshot shows the resulting cell appearance:



View cell

A `ViewCell` is a custom cell whose appearance is defined by a `View`. `ViewCell` defines a `View` property, of type `View`, which defines the view that represents the content of the cell. This property is backed by a `BindableProperty` object, which means it can be the target of data bindings, and be styled.

NOTE

The `View` property is the content property of the `ViewCell` class, and therefore does not need to be explicitly set from XAML.

The following example shows using a `ViewCell` to define the appearance of an item in a `TableView`:

```

<TableView Intent="Settings">
    <TableRoot>
        <TableSection Title="Silent">
            <ViewCell>
                <Grid RowDefinitions="Auto,Auto"
                      ColumnDefinitions="0.5*,0.5*"
                      Margin="10,10,0,0">
                    <Label Text="Vibrate"
                           HorizontalOptions="End" />
                    <Switch Grid.Column="1"
                           HorizontalOptions="End" />
                    <Slider Grid.Row="1"
                           Grid.ColumnSpan="2"
                           Margin="10"
                           Minimum="0"
                           Maximum="10"
                           Value="3" />
                </Grid>
            </ViewCell>
        </TableSection>
    </TableRoot>
</TableView>

```

Inside the `ViewCell`, layout can be managed by any .NET MAUI layout. The following screenshot shows the resulting cell appearance:



Size items

By default, all cells of the same type in a `TableView` have the same height. However, this behavior can be changed with the `HasUnevenRows` and `RowHeight` properties. By default, the `HasUnevenRows` property is `false`.

The `RowHeight` property can be set to an `int` that represents the height of each item in the `TableView`, provided that `HasUnevenRows` is `false`. When `HasUnevenRows` is set to `true`, each item in the `TableView` can have a different height. The height of each item will be derived from the contents of each cell, and so each item will be sized to its content.

Individual cells can be programmatically resized at runtime by changing layout related properties of elements within the cell, provided that the `HasUnevenRows` property is `true`. The following example changes the height of the cell when it's tapped:

```

void OnViewCellTapped(object sender, EventArgs e)
{
    label.Visible = !label.Visible;
    viewCell.ForceUpdateSize();
}

```

In this example, the `OnViewCellTapped` event handler is executed in response to the cell being tapped. The event handler updates the visibility of the `Label` object and the `Cell.ForceUpdateSize` method updates the cell's size. If the `Label` has been made visible the cell's height will increase. If the `Label` has been made invisible the cell's height will decrease.

WARNING

Overuse of dynamic item sizing can cause `TableView` performance to degrade.

Right-to-left layout

`TableView` 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="TableViewDemos.RightToLeftTablePage"
    Title="Right to left TableView"
    FlowDirection="RightToLeft">
    <TableView Intent="Settings">
        ...
    </TableView>
</ContentPage>
```

The default `FlowDirection` for an element with a parent is `MatchParent`. Therefore, the `TableView` inherits the `FlowDirection` property value from the `ContentPage`.

ContentView

9/20/2022 • 4 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `ContentView` is a control that enables the creation of custom, reusable controls.

The `ContentView` class defines a `Content` property, of type `View`, which represents the content of the `ContentView`. This property is backed by a `BindableProperty` object, which means that it can be the target of data bindings, and styled.

The `ContentView` class derives from the `TemplatedView` class, which defines the `ControlTemplate` bindable property, of type `ControlTemplate`, which defines the appearance of the control. For more information about the `ControlTemplate` property, see [Customize appearance with a ControlTemplate](#).

NOTE

A `ContentView` can only contain a single child.

Create a custom control

The `ContentView` class offers little functionality by itself but can be used to create a custom control. The process for creating a custom control is to:

1. Create a class that derives from the `ContentView` class.
2. Define any control properties or events in the code-behind file for the custom control.
3. Define the UI for the custom control.

This article demonstrates how to create a `CardView` control, which is a UI element that displays an image, title, and description in a card-like layout.

Create a ContentView-derived class

A `ContentView`-derived class can be created using the `ContentView` item template in Visual Studio. This template creates a XAML file in which the UI for the custom control can be defined, and a code-behind file in which any control properties, events, and other logic can be defined.

Define control properties

Any control properties, events, and other logic should be defined in the code-behind file for the `ContentView`-derived class.

The `CardView` custom control defines the following properties:

- `CardTitle`, of type `string`, which represents the title shown on the card.
- `CardDescription`, of type `string`, which represents the description shown on the card.
- `IconImageSource`, of type `ImageSource`, which represents the image shown on the card.
- `IconBackgroundColor`, of type `Color`, which represents the background color for the image shown on the card.
- `BorderColor`, of type `Color`, which represents the color of the card border, image border, and divider line.
- `CardColor`, of type `Color`, which represents the background color of the card.

Each property is backed by a `BindableProperty` instance.

The following example shows the `CardTitle` bindable property in the code-behind file for the `CardView` class:

```
public partial class CardView : ContentView
{
    public static readonly BindableProperty CardTitleProperty = BindableProperty.Create(nameof(CardTitle),
        typeof(string), typeof(CardView), string.Empty);

    public string CardTitle
    {
        get => (string)GetValue(CardView.CardTitleProperty);
        set => SetValue(CardView.CardTitleProperty, value);
    }
    // ...

    public CardView()
    {
        InitializeComponent();
    }
}
```

For more information about `BindableProperty` objects, see [Bindable properties](#).

Define the UI

The custom control UI can be defined in the XAML file for the `ContentView`-derived class, which uses a `ContentView` as the root element of the control:

```
<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 [Data binding](#).

NOTE

The `FallbackValue` property in the `Binding` expression provides a default value in case the binding is `null`.

Instantiate a custom control

A reference to the custom control namespace must be added to the page that instantiates the custom control. Once the reference has been added the `CardView` can be instantiated, and its properties defined:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:controls="clr-namespace:CardViewDemo.Controls"
    x:Class="CardViewDemo.CardViewXamlPage">

    <ScrollView>
        <StackLayout>
            <controls:CardView BorderColor="DarkGray"
                CardTitle="Slavko Vlasic"
                CardDescription="Lorem ipsum dolor sit amet, consectetur adipiscing elit.
                    Nulla elit dolor, convallis non interdum."
                IconBackgroundColor="SlateGray"
                IconImageSource="user.png" />
            <!-- More CardView objects -->
        </StackLayout>
    </ScrollView>
</ContentPage>
```

The following screenshot shows multiple `CardView` objects:



Customize appearance with a ControlTemplate

A custom control that derives from the `ContentView` class can define its UI using XAML or code, or may not define its UI at all. A `ControlTemplate` can be used to override the control's appearance, regardless of how that appearance is defined.

For example, a `CardView` layout might occupy too much space for some use cases. A `ControlTemplate` can be used to 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 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>
    </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 screenshot shows a standard `CardView` instance, and multiple `CardView` instances whose control templates have been overridden:

A standard CardView control is suitable for grid layouts:



A ControlTemplate overrides standard view, creating a more compact view suitable for lists:



For more information about control templates, see [Control templates](#).

Display a menu bar in a .NET MAUI desktop app

9/20/2022 • 2 minutes to read • [Edit Online](#)

A .NET Multi-platform App UI (.NET MAUI) menu bar is a container that presents a set of menus in a horizontal row, at the top of a desktop app.

Each top-level menu in the menu bar is represented by a `MenuItem` object. `MenuItem` defines the following properties:

- `Text`, of type `string`, defines the menu text.
- `IsEnabled`, of type `boolean`, specifies whether the menu is enabled. The default value of this property is `true`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

An item in a menu is represented by a `MenuFlyoutItem`, and a sub menu for a menu is represented by a `MenuFlyoutSubItem`. `MenuFlyoutSubItem` derives from `MenuFlyoutItem`, which in turn derives from `MenuItem`. `MenuItem` defines multiple properties that enable the appearance and behavior of a menu item to be specified.

Create menu bar items

`MenuItem` objects can be added to the `MenuItems` collection, of type `IList<MenuItem>`, on a `ContentPage`. .NET MAUI desktop apps will display a menu bar, containing menu items, when they are added to any `ContentPage` that's hosted in a `NavigationPage` or a Shell app.

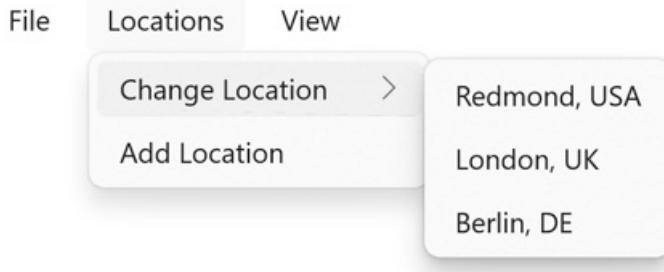
The following example shows a `ContentPage` that defines menu bar items:

```

<ContentPage ...>
    <ContentPage.MenuBarItems>
        <MenuBarItem Text="File">
            <MenuFlyoutItem Text="Exit"
                Command="{Binding ExitCommand}" />
        </MenuBarItem>
        <MenuBarItem Text="Locations">
            <MenuFlyoutSubItem Text="Change Location">
                <MenuFlyoutItem Text="Redmond, USA"
                    Command="{Binding ChangeLocationCommand}"
                    CommandParameter="Redmond" />
                <MenuFlyoutItem Text="London, UK"
                    Command="{Binding ChangeLocationCommand}"
                    CommandParameter="London" />
                <MenuFlyoutItem Text="Berlin, DE"
                    Command="{Binding ChangeLocationCommand}"
                    CommandParameter="Berlin"/>
            </MenuFlyoutSubItem>
            <MenuFlyoutItem Text="Add Location"
                Command="{Binding AddLocationCommand}" />
        </MenuBarItem>
        <MenuBarItem Text="View">
            <MenuFlyoutItem Text="Refresh"
                Command="{Binding RefreshCommand}" />
            <MenuFlyoutItem Text="Change Theme"
                Command="{Binding ChangeThemeCommand}" />
        </MenuBarItem>
    </ContentPage.MenuBarItems>
</ContentPage>

```

This example defines three top-level menus. Each top-level menu has menu items, and the second top-level menu has a sub-menu:



Each `MenuFlyoutItem` defines a menu item that executes an `ICommand` when selected.

Mac Catalyst limitations

.NET MAUI Mac Catalyst apps are limited to 50 menu items. Attempting to add more than 50 menu items to a Mac Catalyst app will result in an exception being thrown.

Additional menu items, beyond the 50 limit, can be added to a menu bar by adding the following code to your `AppDelegate` class:

```

[Export("MenuItem50:")]
internal void MenuItem50(UICommand u ICommand)
{
    u ICommand.SendClicked();
}

```

Display pop-ups

9/20/2022 • 3 minutes to read • [Edit Online](#)

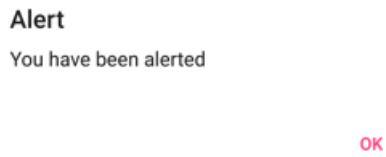
Displaying an alert, asking a user to make a choice, or displaying a prompt is a common UI task. .NET Multi-platform App UI (.NET MAUI) has three methods on the `Page` class for interacting with the user via a pop-up: `DisplayAlert`, `DisplayActionSheet`, and `DisplayPromptAsync`. Pop-ups are rendered with native controls on each platform.

Display an alert

All .NET MAUI-supported platforms have a modal pop-up to alert the user or ask simple questions of them. To display alerts, use the `DisplayAlert` method on any `Page`. The following example shows a simple message to the user:

```
await DisplayAlert("Alert", "You have been alerted", "OK");
```

The alert is displayed modally, and once dismissed the user continues interacting with the app:



The `DisplayAlert` method can also be used to capture a user's response by presenting two buttons and returning a `bool`. To get a response from an alert, supply text for both buttons and `await` the method:

```
bool answer = await DisplayAlert("Question?", "Would you like to play a game", "Yes", "No");
Debug.WriteLine("Answer: " + answer);
```



After the user selects one of the options the response will be returned as a `bool`.

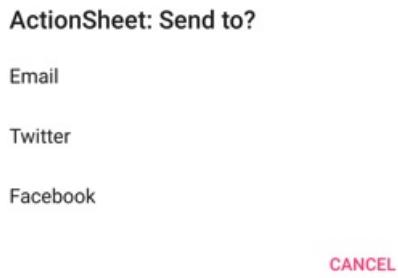
The `DisplayAlert` method also has overloads that accept a `FlowDirection` argument that specifies the direction in which UI elements flow within the alert.

Guide users through tasks

An action sheet presents the user with a set of alternatives for how to proceed with a task. To display an action sheet, use the `DisplayActionSheet` method on any `Page`, passing the message and button labels as strings:

```
string action = await DisplayActionSheet("ActionSheet: Send to?", "Cancel", null, "Email", "Twitter",
"Facebook");
Debug.WriteLine("Action: " + action);
```

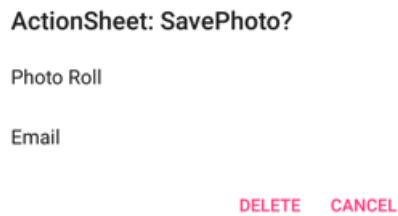
The action sheet will be displayed modally:



After the user taps one of the buttons, the button label will be returned as a `string`.

Action sheets also support a destroy button, which is a button that represents destructive behavior. The destroy button can be specified as the third string argument to the `DisplayActionSheet` method, or can be left `null`. The following example specifies a destroy button:

```
async void OnActionSheetCancelDeleteClicked(object sender, EventArgs e)
{
    string action = await DisplayActionSheet("ActionSheet: SavePhoto?", "Cancel", "Delete", "Photo Roll",
    "Email");
    Debug.WriteLine("Action: " + action);
}
```



NOTE

On iOS, the destroy button is rendered differently to the other buttons in the action sheet.

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.

Display a prompt

To display a prompt, call the `DisplayPromptAsync` on 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:

Question 1

What's your name?



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:

Question 2

What's $5 + 5$?

10

CANCEL OK

1 2 3 -

4 5 6 —

7 8 9

, 0 .

Fonts in .NET MAUI

9/20/2022 • 5 minutes to read • [Edit Online](#)

By default, .NET Multi-platform App UI (.NET MAUI) apps use the Open Sans font on each platform. However, this default can be changed, and additional fonts can be registered for use in an app.

All controls that display text define properties that can be set to change font appearance:

- `FontFamily`, of type `string`.
- `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`.
- `FontAutoScalingEnabled`, of type `bool`, which defines whether an app's UI reflects text scaling preferences set in the operating system. The default value of this property is `true`.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

All controls that display text automatically use font scaling, which means that an app's UI reflects text scaling preferences set in the operating system.

Register fonts

True type format (TTF) and open type font (OTF) fonts can be added to your app and referenced by filename or alias, with registration being performed in the `CreateMauiApp` method in the `MauiProgram` class. This is accomplished by invoking the `ConfigureFonts` method on the `MauiApplicationBuilder` object. Then, on the `IFontCollection` object, call the `AddFont` method to add the required font to your app:

```
namespace MyMauiApp
{
    public static class MauiProgram
    {
        public static MauiApp CreateMauiApp()
        {
            var builder = MauiApp.CreateBuilder();
            builder
                .UseMauiApp<App>()
                .ConfigureFonts(fonts =>
            {
                fonts.AddFont("Lobster-Regular.ttf", "Lobster");
            });

            return builder.Build();
        }
    }
}
```

In the example above, the first argument to the `AddFont` method is the font filename, while the second argument represents an optional alias by which the font can be referenced when consuming it.

A font can be added to your app project by dragging it into the `Resources\Fonts` folder of the project, where its build action will automatically be set to **MauiFont**. This creates a corresponding entry in your project file.

Alternatively, all fonts in the app can be registered by using a wildcard in your project file:

```
<ItemGroup>
    <MauiFont Include="Resources\Fonts\*" />
</ItemGroup>
```

Fonts can also be added to other folders of your app project. However, in this scenario their build action must be manually set to **MauiFont** in the **Properties** window.

At build time, fonts are copied to your app package.

NOTE

The `*` wildcard character indicates that all the files within the folder will be treated as being font files. In addition, if you want to include files from sub-folders too, then configure it using additional wildcard characters, for example,
`Resources\Fonts***`.

Consume fonts

Registered fonts can be consumed by setting the `FontFamily` property of a control that displays text to the font name, without the file extension:

```
<!-- Use font name -->
<Label Text="Hello .NET MAUI"
      FontFamily="Lobster-Regular" />
```

Alternatively, it can be consumed by referencing its alias:

```
<!-- Use font alias -->
<Label Text="Hello .NET MAUI"
      FontFamily="Lobster" />
```

The equivalent C# code is:

```
// Use font name
Label label1 = new Label
{
    Text = "Hello .NET MAUI!",
    FontFamily = "Lobster-Regular"
};

// Use font alias
Label label2 = new Label
{
    Text = "Hello .NET MAUI!",
    FontFamily = "Lobster"
};
```

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:

```
<Label Text="Font size 24"
      FontSize="24" />
```

The equivalent C# code is:

```
Label label = new Label
{
    Text = "Font size 24",
    FontSize = 24
};
```

NOTE

The `FontSize` value is measured in device-independent units.

Disable font auto scaling

All controls that display text have font scaling enabled by default, which means that an app's UI reflects text scaling preferences set in the operating system. However, this behavior can be disabled by setting the `FontAutoScalingEnabled` property on text-based control's to `false`:

```
<Label Text="Scaling disabled"
      FontSize="18"
      FontAutoScalingEnabled="False" />
```

This approach is useful when you want to guarantee that text is displayed at a specific size.

NOTE

Font auto scaling also works with font icons. For more information, see [Display font icons](#).

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:

```
<Label Text="Different font properties on different platforms"
      FontSize="{OnPlatform iOS=20, Android=22, WinUI=24}">
    <Label.FontFamily>
      <OnPlatform x:TypeArguments="x:String">
        <On Platform="iOS" Value="MarkerFelt-Thin" />
        <On Platform="Android" Value="Lobster-Regular" />
        <On Platform="WinUI" Value="ArimaMadurai-Black" />
      </OnPlatform>
    </Label.FontFamily>
</Label>
```

The `DeviceInfo.Platform` 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 = DeviceInfo.Platform == DevicePlatform.iOS ? 20 :
    DeviceInfo.Platform == DevicePlatform.Android ? 22 : 24;
label.FontFamily = DeviceInfo.Platform == DevicePlatform.iOS ? "MarkerFelt-Thin" :
    DeviceInfo.Platform == DevicePlatform.Android ? "Lobster-Regular" : "ArimaMadurai-Black";
```

For more information about providing platform-specific values, see [Device information](#). For information about the `OnPlatform` markup extension, see [OnPlatform markup extension](#).

Display font icons

Font icons can be displayed by .NET MAUI apps 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="#xf30c;" 
                    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.FromArgb("#D1D1D1") };
image.Source = new FontImageSource
{
    Glyph = "\uf30c",
    FontFamily = DeviceInfo.Platform == DevicePlatform.iOS ? "Ionicons" : "ionicons.ttf#",
    Size = 44
};
```

The following screenshot shows several font icons being displayed:



Graphics

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) provides a cross-platform graphics canvas on which 2D graphics can be drawn using types from the `Microsoft.Maui.Graphics` namespace. This canvas supports drawing and painting shapes and images, compositing operations, and graphical object transforms.

There are many similarities between the functionality provided by `Microsoft.Maui.Graphics`, and the functionality provided by .NET MAUI shapes and brushes. However, each is aimed at different scenarios:

- `Microsoft.Maui.Graphics` functionality must be consumed on a drawing canvas, enables performant graphics to be drawn, and provides a convenient approach for writing graphics-based controls. For example, a control that replicates the GitHub contribution profile can be more easily implemented using `Microsoft.Maui.Graphics` than by using .NET MAUI shapes.
- .NET MAUI shapes can be consumed directly on a page, and brushes can be consumed by all controls. This functionality is provided to help you produce an attractive UI.

For more information about .NET MAUI shapes, see [Shapes](#).

Drawing canvas

In .NET MAUI, the `GraphicsView` enables consumption of the `Microsoft.Maui.Graphics` functionality, via a drawing canvas that's exposed as an `ICanvas` object. For more information about the `GraphicsView`, see [GraphicsView](#).

`ICanvas` defines the following properties that affect the appearance of objects that are drawn on the canvas:

- `Alpha`, of type `float`, indicates the opacity of an object.
- `Antialias`, of type `bool`, specifies whether anti-aliasing is enabled.
- `BlendMode`, of type `BlendMode`, defines the blend mode, which determines what happens when an object is rendered on top of an existing object.
- `DisplayScale`, of type `float`, represents the scaling factor to scale the UI by on a canvas.
- `FillColor`, of type `Color`, indicates the color used to paint an object's interior.
- `Font`, of type `IFont`, defines the font when drawing text.
- `FontColor`, of type `Color`, specifies the font color when drawing text.
- `FontSize`, of type `float`, defines the size of the font when drawing text.
- `MiterLimit`, of type `float`, specifies the limit of the miter length of line joins in an object.
- `StrokeColor`, of type `Color`, indicates the color used to paint an object's outline.
- `StrokeDashPattern`, of type `float[]`, specifies the pattern of dashes and gaps that are used to outline an object.
- `StrokeLineCap`, of type `LineCap`, describes the shape at the start and end of a line.
- `StrokeLineJoin`, of type `LineJoin`, specifies the type of join that is used at the vertices of a shape.
- `StrokeSize`, of type `float`, indicates the width of an object's outline.

By default, an `ICanvas` sets `StrokeSize` to 1, `StrokeColor` to black, `StrokeLineJoin` to `LineJoin.Miter`, and `StrokeLineCap` to `LineJoin.Cap`.

Drawing canvas state

The drawing canvas on each platform has the ability to maintain its state. This enables you to persist the current graphics state, and restore it when required.

However, not all elements of the canvas are elements of the graphics state. The graphics state does not include drawing objects, such as paths, and paint objects, such as gradients. Typical elements of the graphics state on each platform include stroke and fill data, and font data.

The graphics state of each `ICanvas` can be manipulated with the following methods:

- `SaveState`, which saves the current graphics state.
- `RestoreState`, which sets the graphics state to the most recently saved state.
- `ResetState`, which resets the graphics state to its default values.

NOTE

The state that's persisted by these methods is platform dependent.

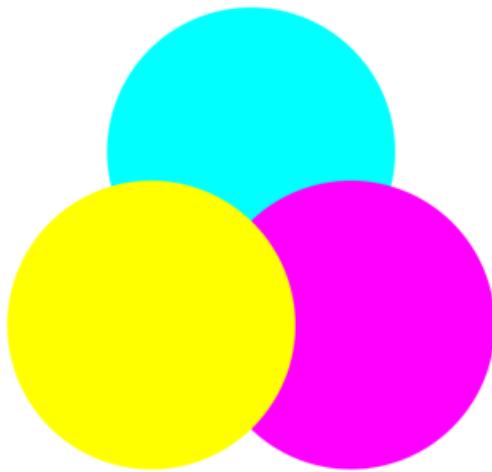
Blend modes

9/20/2022 • 7 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) graphics enables different compositing operations for graphical objects to be specified by the `ICanvas.BlendMode` property. This property determines what happens when a graphical object (called the *source*), is rendered on top of an existing graphical object (called the *destination*).

By default, the last drawn object obscures the objects drawn underneath it:



In this example, the cyan circle is drawn first, followed by the magenta circle, then the yellow circle. Each circle obscures the circle drawn underneath it. This occurs because the default blend mode is `Normal`, which means that the source is drawn over the destination. However, it's possible to specify a different blend mode for a different result. For example, if you specify `DestinationOver`, then in the area where the source and destination intersect, the destination is drawn over the source.

The 28 members of the `BlendMode` enumeration can be divided into three categories:

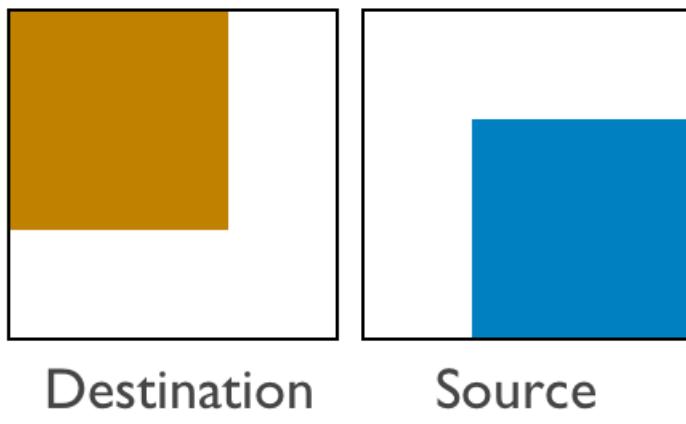
SEPARABLE	NON-SEPARABLE	PORTER-DUFF
<code>Normal</code>	<code>Hue</code>	<code>Clear</code>
<code>Multiply</code>	<code>Saturation</code>	<code>Copy</code>
<code>Screen</code>	<code>Color</code>	<code>SourceIn</code>
<code>Overlay</code>	<code>Luminosity</code>	<code>SourceOut</code>
<code>Darken</code>		<code>SourceAtop</code>
<code>Lighten</code>		<code>DestinationOver</code>
<code>ColorDodge</code>		<code>DestinationIn</code>
<code>ColorBurn</code>		<code>DestinationOut</code>

SEPARABLE	NON-SEPARABLE	PORTER-DUFF
SoftLight		DestinationAtop
HardLight		Xor
Difference		PlusDarker
Exclusion		PlusLighter

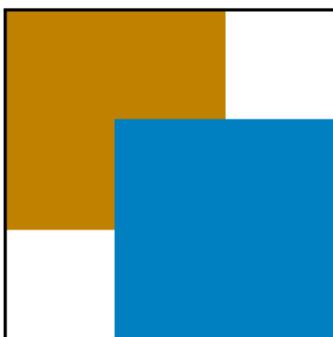
The order that the members are listed in the table above is the same as in the `BlendMode` enumeration. The first column lists the 12 *separable* blend modes, while the second column lists the *non-separable* blend modes. Finally, the third column lists the *Porter-Duff* blend modes.

Porter-Duff blend modes

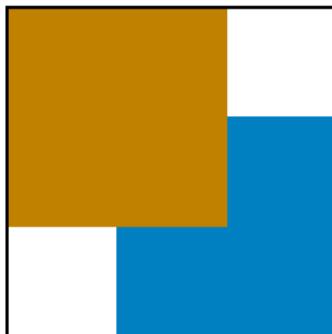
The Porter-Duff blend modes, named after Thomas Porter and Tom Duff, define 12 compositing operators that describe how to compute the color resulting from the composition of the source with the destination. These compositing operators can best be described by considering the case of drawing two rectangles that contain transparent areas:



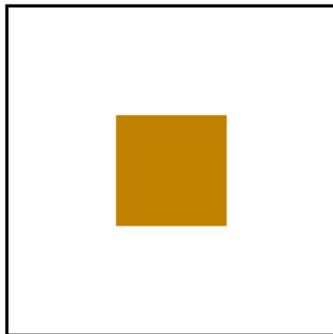
In the image above, the destination is a transparent rectangle except for a brown area that occupies the left and top two-thirds of the display surface. The source is also a transparent rectangle except for a blue area that occupies the right and bottom two-thirds of the display surface. Displaying the source on the destination produces the following result:



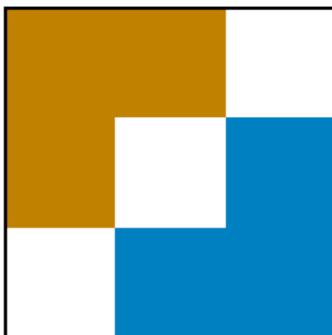
The transparent pixels of the source allow the background to show through, while the blue source pixels obscure the background. This is the normal case, using the default blend mode of `Normal`. However, it's possible to specify that in the area where the source and destination intersect, the destination appears instead of the source, using the `DestinationOver` blend mode:



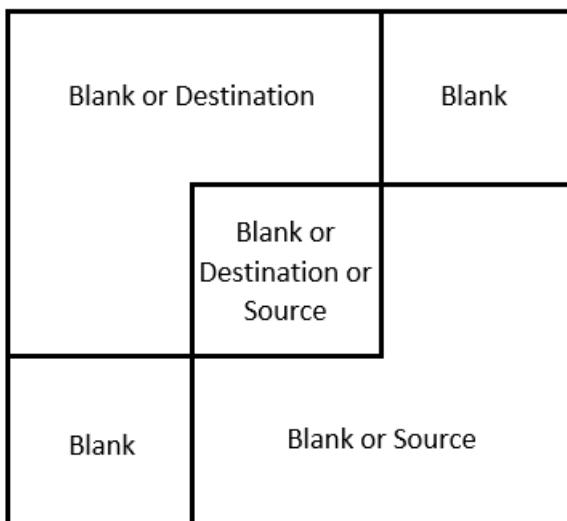
The `DestinationIn` blend mode displays only the area where the destination and source intersect, using the destination color:



The `xor` blend mode causes nothing to appear where the two areas overlap:



The colored destination and source rectangles effectively divide the display surface into four unique areas that can be colored in different ways, corresponding to the presence of the destination and source rectangles:



The upper-right and lower-left rectangles are always blank because both the destination and source are

transparent in those areas. The destination color occupies the upper-left area, so that area can either be colored with the destination color or not at all. Similarly, the source color occupies the lower-right area, so that area can be colored with the source color or not at all.

The following table lists the Porter-Duff blend modes provided by `Microsoft.Maui.Graphics`, and how they color each of the three non-blank areas in the diagram above:

BLEND MODE	DESTINATION	INTERSECTION	SOURCE
<code>Clear</code>			
<code>Copy</code>		Source	X
<code>SourceIn</code>		Source	
<code>SourceOut</code>			X
<code>SourceAtop</code>	X	Source	
<code>DestinationOver</code>	X	Destination	X
<code>DestinationIn</code>		Destination	
<code>DestinationOut</code>	X		
<code>DestinationAtop</code>		Destination	X
<code>Xor</code>	X		X
<code>PlusDarker</code>	X	Sum	X
<code>PlusLighter</code>	X	Sum	X

The naming convention of the modes follows a few simple rules:

- The *Over* suffix indicates what is visible in the intersection. Either the source or destination is drawn over the other.
- The *In* suffix means that only the intersection is colored. The output is restricted to only the part of the source or destination that is in the other.
- The *Out* suffix means that the intersection isn't colored. The output is only the part of the source or destination that is out of the intersection.
- The *Atop* suffix is the union of *In* and *Out*. It includes the area where the source or destination is atop of the other.

NOTE

These blend modes are symmetrical. The source and destination can be exchanged, and all the modes are still available.

The `PlusLighter` blend mode sums the source and destination. Then, for values above 1, white is displayed.

Similarly, the `PlusDarker` blend mode sums the source and destination, but subtracts 1 from the resulting values, with values below 0 becoming black.

Separable blend modes

The separable blend modes alter the individual red, green, and blue color components of a graphical object.

The following table shows the separable blend modes, with brief explanations of what they do. In the table, `Dc` and `Sc` refer to the destination and source colors, and the second column shows the source color that produces no change:

BLEND MODE	NO CHANGE	OPERATION
Normal		No blending, source selected
Multiply	White	Darkens by multiplying $Sc \cdot Dc$
Screen	Black	Complements the product of complements: $Sc + Dc - Sc \cdot Dc$
Overlay	Gray	Inverse of HardLight
Darken	White	Minimum of colors: $\min(Sc, Dc)$
Lighten	Black	Maximum of colors: $\max(Sc, Dc)$
ColorDodge	Black	Brightens the destination based on the source
ColorBurn	White	Darkens the destination based on the source
SoftLight	Gray	Similar to the effect of a soft spotlight
HardLight	Gray	Similar to the effect of a harsh spotlight
Difference	Black	Subtracts the darker color from the lighter color: $Abs(Dc - Sc)$
Exclusion	Black	Similar to Difference but lower contrast

NOTE

If the source is transparent, then the separable blend modes have no effect.

The following example uses the `Multiply` blend mode to draw three overlapping circles of cyan, magenta, and yellow:

```

PointF center = new PointF(dirtyRect.Center.X, dirtyRect.Center.Y);
float radius = Math.Min(dirtyRect.Width, dirtyRect.Height) / 4;
float distance = 0.8f * radius;

PointF center1 = new PointF(distance * (float)Math.Cos(9 * Math.PI / 6) + center.X,
    distance * (float)Math.Sin(9 * Math.PI / 6) + center.Y);
PointF center2 = new PointF(distance * (float)Math.Cos(1 * Math.PI / 6) + center.X,
    distance * (float)Math.Sin(1 * Math.PI / 6) + center.Y);
PointF center3 = new PointF(distance * (float)Math.Cos(5 * Math.PI / 6) + center.X,
    distance * (float)Math.Sin(5 * Math.PI / 6) + center.Y);

canvas.BlendMode = BlendMode.Multiply;

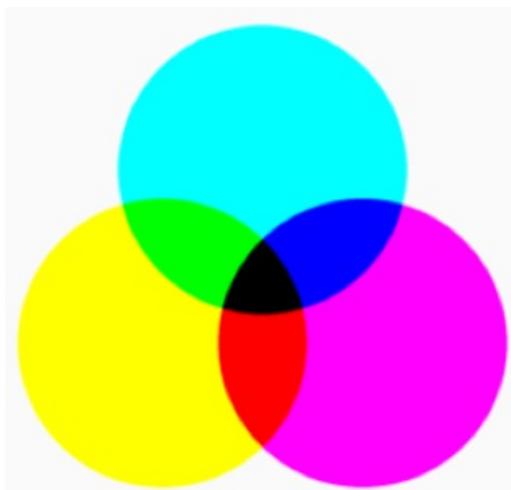
canvas.FillColor = Colors.Cyan;
canvas.FillCircle(center1, radius);

canvas.FillColor = Colors.Magenta;
canvas.FillCircle(center2, radius);

canvas.FillColor = Colors.Yellow;
canvas.FillCircle(center3, radius);

```

The result is that a combination of any two colors produces red, green, and blue, and a combination of all three colors produces black.



Non-separable blend modes

The non-separable blend modes combine hue, saturation, and luminosity values from the destination and the source. Understanding these blend modes requires an understanding of the Hue-Saturation-Luminosity (HSL) color model:

- The hue value represents the dominant wavelength of the color. Hue values range from 0 to 360, and cycle through the additive and subtractive primary colors. Red is the value 0, yellow is 60, green is 120, cyan is 180, blue is 240, magenta is 300, and the cycle returns to red at 360. If there is no dominant color, for example the color is white or black or a gray shade, then the hue is undefined and usually set to 0.
- The saturation value indicates the purity of the color, and can range from 0 to 100. A saturation value of 100 is the purest color while values lower than 100 cause the color to become more grayish. A saturation value of 0 results in a shade of gray.
- The luminosity value indicates how bright the color is. A luminosity value of 0 is black regardless of other values. Similarly, a luminosity value of 100 is white.

The HSL value (0,100,50) is the RGB value (255,0,0), which is pure red. The HSL value (180,100,50) is the RGB value (0, 255, 255), which is pure cyan. As the saturation is decreased, the dominant color component is decreased and the other components are increased. At a saturation level of 0, all the components are the same

and color is a gray shade.

The non-separable blend modes conceptually perform the following steps:

1. Convert the source and destination objects from their original color space to the HSL color space.
2. Create the composited object from a combination of hue, saturation, and luminosity components, from the source and destination objects.
3. Convert the result back to the original color space.

The following table lists how which HSL components are composited for each non-separable blend mode:

BLEND MODE	SOURCE COMPONENTS	DESTINATION COMPONENTS
Hue	Hue	Saturation and Luminosity
Saturation	Saturation	Hue and Luminosity
Color	Hue and Saturation	Luminosity
Luminosity	Luminosity	Hue and Saturation

Colors

9/20/2022 • 4 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The `Color` class, in the `Microsoft.Maui.Graphics` namespace, 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 `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 `float` values ranging from 0 to 1.

NOTE

The default `Color` constructor creates a black `Color` object.

You can also use the following static methods to create `Color` objects:

- `Color.FromRgb` from `float` RGB values that range from 0 to 1.
- `Color.FromRgb` from `double` RGB values that range from 0 to 1.
- `Color.FromRgb` from `byte` RGB values that range from 0 to 255.
- `Color.FromRgba` from `float` RGBA values that range from 0 to 1.
- `Color.FromRgba` from `double` RGBA values that range from 0 to 1.
- `Color.FromRgba` from `byte` RGBA values that range from 0 to 255.
- `Color.FromRgba` from a `string`-based hexadecimal value in the form "#RRGGBBAA" or "#RRGGBB" or "#RGBA" or "#RGB", where each letter corresponds to a hexadecimal digit for the alpha, red, green, and blue channels.
- `Color.FromHsla` from `float` HSLA values.
- `Color.FromHsla` from `double` HSLA values.
- `Color.FromHsv` from `float` HSV values that range from 0 to 1.
- `Color.FromHsv` from `int` HSV values that range from 0 to 255.
- `Color.FromHsva` from `float` HSVA values.
- `Color.FromHsva` from `int` HSV values.
- `Color.FromInt` from an `int` value calculated as $(B + 256 * (G + 256 * (R + 256 * A)))$.
- `Color.FromUint` from a `uint` value calculated as $(B + 256 * (G + 256 * (R + 256 * A)))$.
- `Color.FromArgb` from a `string`-based hexadecimal value 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.

NOTE

In addition to the methods listed above, the `Color` class also has `Parse` and `TryParse` methods that create `Color` objects from `string` arguments.

Once created, a `Color` object is immutable. The characteristics of the color can be obtained from the following `float` properties, that range from 0 to 1:

- `Red`, which represents the red channel of the color.
- `Green`, which represents the green channel of the color.
- `Blue`, which represents the blue channel of the color.
- `Alpha`, which represents the alpha channel of the color.

In addition, the characteristics of the color can be obtained from the following methods:

- `GetHue`, which returns a `float` that represents the hue channel of the color.
- `GetSaturation`, which returns a `float` that represents the saturation channel of the color.
- `Luminosity`, which returns a `float` that represents the luminosity channel of the color.

Named colors

The `colors` class defines 148 public static read-only fields for common colors, such as `AntiqueWhite`, `MidnightBlue`, and `YellowGreen`.

Modify a color

The following instance methods modify an existing color to create a new color:

- `AddLuminosity` returns a `Color` by adding the luminosity value to the supplied delta value.
- `GetComplementary` returns the complementary `Color`.
- `MultiplyAlpha` returns a `Color` by multiplying the alpha value by the supplied `float` value.
- `WithAlpha` returns a `Color`, replacing the alpha value with the supplied `float` value.
- `WithHue` returns a `Color`, replacing the hue value with the supplied `float` value.
- `WithLuminosity` returns a `Color`, replacing the luminosity value with the supplied `float` value.
- `WithSaturation` returns a `Color`, replacing the saturation value with the supplied `float` value.

Conversions

The following instance methods convert a `Color` to an alternative representation:

- `AsPaint` returns a `SolidPaint` object whose `Color` property is set to the color.
- `ToHex` returns a hexadecimal `string` representation of a `Color`.
- `ToArgbHex` returns an ARGB hexadecimal `string` representation of a `Color`.
- `ToRgbaHex` returns an RGBA hexadecimal `string` representation of a `Color`.
- `ToInt` returns an ARGB `int` representation of a `Color`.
- `ToUint` returns an ARGB `uint` representation of a `Color`.
- `ToRgb` converts a `Color` to RGB `byte` values that are returned as `out` arguments.
- `ToRgba` converts a `Color` to RGBA `byte` values that are returned as `out` arguments.
- `ToHsl` converts a `Color` to HSL `float` values that are passed as `out` arguments.

Examples

In XAML, colors are typically referenced using their named values, or with hexadecimal:

```
<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" />
```

In C#, colors are typically referenced using their named values, or with their static methods:

```
Label red      = new Label { Text = "Red",      TextColor = Colors.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 AliceBlue, iOS=MidnightBlue}"
                   IsRunning="True" />
```

The equivalent C# code is:

```
ActivityIndicator activityIndicator = new ActivityIndicator
{
    Color = DeviceInfo.Platform == DevicePlatform.iOS ? Colors.MidnightBlue : Colors.AliceBlue,
    IsRunning = true
};
```

Draw graphical objects

9/20/2022 • 15 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) graphics, in the `Microsoft.Maui.Graphics` namespace, enables you to draw graphical objects on a canvas that's defined as an `ICanvas` object.

The .NET MAUI `GraphicsView` control provides access to an `ICanvas` object, on which properties can be set and methods invoked to draw graphical objects. For more information about the `GraphicsView`, see [GraphicsView](#).

IMPORTANT

Graphical objects are drawn on an `ICanvas` in units of pixels.

Draw a line

Lines can be drawn on an `ICanvas` using the `DrawLine` method, which requires four `float` arguments that represent the start and end points of the line.

The following example shows how to draw a line:

```
canvas.StrokeColor = Colors.Red;
canvas.StrokeSize = 6;
canvas.DrawLine(10, 10, 90, 100);
```

In this example, a red diagonal line is drawn from (10,10) to (90,100):



NOTE

There's also a `DrawLine` overload that takes two `PointF` arguments.

The following example shows how to draw a dashed line:

```
canvas.StrokeColor = Colors.Red;
canvas.StrokeSize = 4;
canvas.StrokeDashPattern = new float[] { 2, 2 };
canvas.DrawLine(10, 10, 90, 100);
```

In this example, a red dashed diagonal line is drawn from (10,10) to (90,100):



For more information about dashed lines, see [Draw dashed objects](#).

Draw an ellipse

Ellipses and circles can be drawn on an `ICanvas` using the `DrawEllipse` method, which requires `x`, `y`, `width`, and `height` arguments, of type `float`.

The following example shows how to draw an ellipse:

```
canvas.StrokeColor = Colors.Red;
canvas.StrokeSize = 4;
canvas.DrawEllipse(10, 10, 100, 50);
```

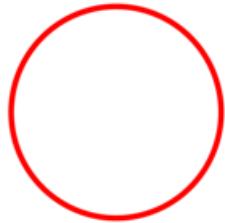
In this example, a red ellipse with dimensions 100x50 is drawn at (10,10):



To draw a circle, make the `width` and `height` arguments to the `DrawEllipse` method equal:

```
canvas.StrokeColor = Colors.Red;
canvas.StrokeSize = 4;
canvas.DrawEllipse(10, 10, 100, 100);
```

In this example, a red circle with dimensions 100x100 is drawn at (10,10):



NOTE

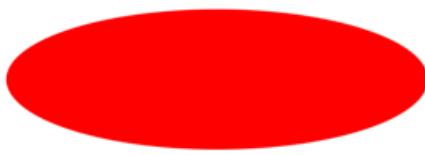
Circles can also be drawn with the `DrawCircle` method.

For information about drawing a dashed ellipse, see [Draw dashed objects](#).

A filled ellipse can be drawn with the `FillEllipse` method, which also requires `x`, `y`, `width`, and `height` arguments, of type `float`:

```
canvas.FillColor = Colors.Red;
canvas.FillEllipse(10, 10, 150, 50);
```

In this example, a red filled ellipse with dimensions 150x50 is drawn at (10,10):



The `FillColor` property of the `ICanvas` object must be set to a `Color` before invoking the `FillEllipse` method.

Filled circles can also be drawn with the `FillCircle` method.

NOTE

There are `DrawEllipse` and `FillEllipse` overloads that take `Rect` and `RectF` arguments. In addition, there are also `DrawCircle` and `FillCircle` overloads.

Draw a rectangle

Rectangles and squares can be drawn on an `ICanvas` using the `DrawRectangle` method, which requires `x`, `y`, `width`, and `height` arguments, of type `float`.

The following example shows how to draw a rectangle:

```
canvas.StrokeColor = Colors.DarkBlue;
canvas.StrokeSize = 4;
canvas.DrawRectangle(10, 10, 100, 50);
```

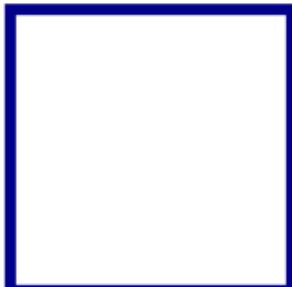
In this example, a dark blue rectangle with dimensions 100x50 is drawn at (10,10):



To draw a square, make the `width` and `height` arguments to the `DrawRectangle` method equal:

```
canvas.StrokeColor = Colors.DarkBlue;
canvas.StrokeSize = 4;
canvas.DrawRectangle(10, 10, 100, 100);
```

In this example, a dark blue square with dimensions 100x100 is drawn at (10,10):



For information about drawing a dashed rectangle, see [Draw dashed objects](#).

A filled rectangle can be drawn with the `FillRectangle` method, which also requires `x`, `y`, `width`, and `height` arguments, of type `float`:

```
canvas.FillColor = Colors.DarkBlue;
canvas.FillRectangle(10, 10, 100, 50);
```

In this example, a dark blue filled rectangle with dimensions 100x50 is drawn at (10,10):



The `FillColor` property of the `ICanvas` object must be set to a `Color` before invoking the `FillRectangle` method.

NOTE

There are `DrawRectangle` and `FillRectangle` overloads that take `Rect` and `RectF` arguments.

Draw a rounded rectangle

Rounded rectangles and squares can be drawn on an `ICanvas` using the `DrawRoundedRectangle` method, which requires `x`, `y`, `width`, `height`, and `cornerRadius` arguments, of type `float`. The `cornerRadius` argument specifies the radius used to round the corners of the rectangle.

The following example shows how to draw a rounded rectangle:

```
canvas.StrokeColor = Colors.Green;
canvas.StrokeSize = 4;
canvas.DrawRoundedRectangle(10, 10, 100, 50, 12);
```

In this example, a green rectangle with rounded corners and dimensions 100x50 is drawn at (10,10):



For information about drawing a dashed rounded rectangle, see [Draw dashed objects](#).

A filled rounded rectangle can be drawn with the `FillRoundedRectangle` method, which also requires `x`, `y`, `width`, `height`, and `cornerRadius` arguments, of type `float`:

```
canvas.FillColor = Colors.Green;
canvas.FillRoundedRectangle(10, 10, 100, 50, 12);
```

In this example, a green filled rectangle with rounded corners and dimensions 100x50 is drawn at (10,10):



The `FillColor` property of the `ICanvas` object must be set to a `Color` before invoking the `FillRoundedRectangle` method.

NOTE

There are `DrawRoundedRectangle` and `FillRoundedRectangle` overloads that take `Rect` and `RectF` arguments, and overloads that enable the radius of each corner to be separately specified.

Draw an arc

Arcs can be drawn on an `ICanvas` using the `DrawArc` method, which requires `x`, `y`, `width`, `height`, `startAngle`, and `endAngle` arguments of type `float`, and `clockwise` and `closed` arguments of type `bool`. The `startAngle` argument specifies the angle from the x-axis to the starting point of the arc. The `endAngle` argument specifies the angle from the x-axis to the end point of the arc. The `clockwise` argument specifies the direction in which the arc is drawn, and the `closed` argument specifies whether the end point of the arc will be connected to the start point.

The following example shows how to draw an arc:

```
canvas.StrokeColor = Colors.Teal;
canvas.StrokeSize = 4;
canvas.DrawArc(10, 10, 100, 100, 0, 180, true, false);
```

In this example, a teal arc of dimensions 100x100 is drawn at (10,10). The arc is drawn in a clockwise direction from 0 degrees to 180 degrees, and isn't closed:



For information about drawing a dashed arc, see [Draw dashed objects](#).

A filled arc can be drawn with the `FillArc` method, which requires `x`, `y`, `width`, `height`, `startAngle`, and `endAngle` arguments of type `float`, and a `clockwise` argument of type `bool`:

```
canvas.FillColor = Colors.Teal;
canvas.FillArc(10, 10, 100, 100, 0, 180, true);
```

In this example, a filled teal arc of dimensions 100x100 is drawn at (10,10). The arc is drawn in a clockwise direction from 0 degrees to 180 degrees, and is closed automatically:



The `FillColor` property of the `ICanvas` object must be set to a `Color` before invoking the `FillArc` method.

NOTE

There are `DrawArc` and `FillArc` overloads that take `Rect` and `RectF` arguments.

Draw a path

A path is a collection of one or more *contours*. Each contour is a collection of *connected* straight lines and curves. Contours are not connected to each other but they might visually overlap. Sometimes a single contour can overlap itself.

Paths are used to draw curves and complex shapes and can be drawn on an `ICanvas` using the `DrawPath` method, which requires a `PathF` argument.

A contour generally begins with a call to the `PathF.MoveTo` method, which you can express either as a `PointF` value or as separate `x` and `y` coordinates. The `MoveTo` call establishes a point at the beginning of the contour and an initial current point. You can then call the following methods to continue the contour with a line or curve from the current point to a point specified in the method, which then becomes the new current point:

- `LineTo` to add a straight line to the path.
- `AddArc` to add an arc, which is a line on the circumference of a circle or ellipse.
- `CurveTo` to add a cubic Bezier spline.
- `QuadTo` to add a quadratic Bezier spline.

None of these methods contain all of the data necessary to describe the line or curve. Instead, each method works with the current point established by the method call immediately preceding it. For example, the `LineTo` method adds a straight line to the contour based on the current point.

A contour ends with another call to `MoveTo`, which begins a new contour, or a call to `close`, which closes the contour. The `close` method automatically appends a straight line from the current point to the first point of the contour, and marks the path as closed.

The `PathF` class also defines other methods and properties. The following methods add entire contours to the path:

- `AppendEllipse` appends a closed ellipse contour to the path.
- `AppendCircle` appends a closed circle contour to the path.
- `AppendRectangle` appends a closed rectangle contour to the path.
- `AppendRoundedRectangle` appends a closed rectangle with rounded corners to the path.

The following example shows how to draw a path:

```
PathF path = new PathF();
path.MoveTo(40, 10);
path.LineTo(70, 80);
path.LineTo(10, 50);
path.Close();
canvas.StrokeColor = Colors.Green;
canvas.StrokeSize = 6;
canvas.DrawPath(path);
```

In this example, a closed green triangle is drawn:



A filled path can be drawn with the `FillPath`, which also requires a `PathF` argument:

```
PathF path = new PathF();
path.MoveTo(40, 10);
path.LineTo(70, 80);
path.LineTo(10, 50);
canvas.FillColor = Colors.SlateBlue;
canvas.FillPath(path);
```

In this example, a filled slate blue triangle is drawn:



The `FillColor` property of the `ICanvas` object must be set to a `color` before invoking the `FillPath` method.

IMPORTANT

The `FillPath` method has an overload that enables a `WindingMode` to be specified, which sets the fill algorithm that's used. For more information, see [Winding modes](#).

Draw an image

Images can be drawn on an `ICanvas` using the `DrawImage` method, which requires an `IImage` argument, and `x`, `y`, `width`, and `height` arguments, of type `float`.

The following example shows how to load an image and draw it to the canvas:

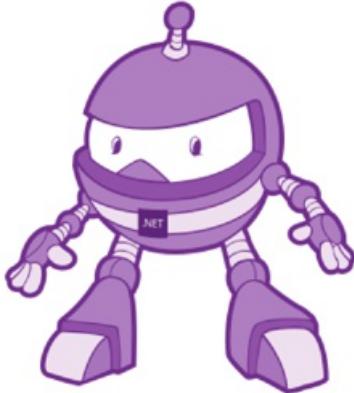
```
using Microsoft.Maui.Graphics.Platform;
...
IImage image;
Assembly assembly = GetType().GetTypeInfo().Assembly;
using (Stream stream =
assembly.GetManifestResourceStream("GraphicsViewDemos.Resources.Images.dotnet_bot.png"))
{
    image = PlatformImage.FromStream(stream);
}

if (image != null)
{
    canvas.DrawImage(image, 10, 10, image.Width, image.Height);
}
```

WARNING

The `PlatformImage` type isn't supported on Windows.

In this example, an image is retrieved from the assembly and loaded as a stream. It's then drawn at actual size at (10,10):



IMPORTANT

Loading an image that's embedded in an assembly requires the image to have its build action set to **Embedded Resource** rather than **MaulImage**.

Draw a string

Strings can be drawn on an `ICanvas` using one of the `DrawString` overloads. The appearance of each string can be defined by setting the `Font`, `FontColor`, and `FontSize` properties. String alignment can be specified by horizontal and vertical alignment options that perform alignment within the string's bounding box.

NOTE

The bounding box for a string is defined by its `x`, `y`, `width`, and `height` arguments.

The following examples show how to draw strings:

```
canvas.FontColor = Colors.Blue;
canvas.FontSize = 18;

canvas.Font = Font.Default;
canvas.DrawString("Text is left aligned.", 20, 20, 380, 100, HorizontalAlignment.Left,
VerticalAlignment.Top);
canvas.DrawString("Text is centered.", 20, 60, 380, 100, HorizontalAlignment.Center, VerticalAlignment.Top);
canvas.DrawString("Text is right aligned.", 20, 100, 380, 100, HorizontalAlignment.Right,
VerticalAlignment.Top);

canvas.Font = Font.DefaultBold;
canvas.DrawString("This text is displayed using the bold system font.", 20, 140, 350, 100,
HorizontalAlignment.Left, VerticalAlignment.Top);

canvas.Font = new Font("Arial");
canvas.FontColor = Colors.Black;
canvas.SetShadow(new SizeF(6, 6), 4, Colors.Gray);
canvas.DrawString("This text has a shadow.", 20, 200, 300, 100, HorizontalAlignment.Left,
VerticalAlignment.Top);
```

In this example, strings with different appearance and alignment options are displayed:

Text is left aligned.

Text is centered.

Text is right aligned.

This text is displayed using the bold system font.

THIS TEXT HAS A SHADOW.

NOTE

The `DrawString` overloads also enable truncation and line spacing to be specified.

For information about drawing shadows, see [Draw a shadow](#).

Draw attributed text

Attributed text can be drawn on an `ICanvas` using the `DrawText` method, which requires an `IAttributedString` argument, and `x`, `y`, `width`, and `height` arguments, of type `float`. Attributed text is a string with associated attributes for parts of its text, that typically represents styling data.

The following example shows how to draw attributed text:

```
using Microsoft.Maui.Graphics.Text;
...
canvas.Font = new Font("Arial");
canvas.FontSize = 18;
canvas.FontColor = Colors.Blue;

string markdownText = @"This is *italic text*, **bold text**, __underline text__, and ***bold italic text***.";
IAttributedString attributedText = MarkdownAttributedStringReader.Read(markdownText); // Requires the Microsoft.Maui.Graphics.Text.Markdig package
canvas.DrawText(attributedText, 10, 10, 400, 400);
```

In this example, markdown is converted to attributed text and displayed with the correct styling:

THIS IS ITALIC TEXT, BOLD TEXT, UNDERLINE TEXT, AND BOLD ITALIC TEXT.

IMPORTANT

Drawing attributed text requires you to have added the `Microsoft.Maui.Graphics.Text.Markdig` NuGet package to your project.

Draw with fill and stroke

Graphical objects with both fill and stroke can be drawn to the canvas by calling a draw method *after* a fill method. For example, to draw an outlined rectangle, set the `FillColor` and `StrokeColor` properties to colors, then call the `FillRectangle` method followed by the `DrawRectangle` method.

The following example draws a filled circle, with a stroke outline, as a path:

```

float radius = Math.Min(dirtyRect.Width, dirtyRect.Height) / 4;

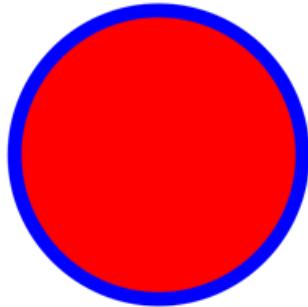
PathF path = new PathF();
path.AppendCircle(dirtyRect.Center.X, dirtyRect.Center.Y, radius);

canvas.StrokeColor = Colors.Blue;
canvas.StrokeSize = 10;
canvas.FillColor = Colors.Red;

canvas.FillPath(path);
canvas.DrawPath(path);

```

In this example, the stroke and fill colors for a `PathF` object are specified. The filled circle is drawn, then the outline stroke of the circle:



WARNING

Calling a draw method before a fill method will result in an incorrect z-order. The fill will be drawn over the stroke, and the stroke won't be visible.

Draw a shadow

Graphical objects drawn on an `ICanvas` can have a shadow applied using the `SetShadow` method, which takes the following arguments:

- `offset`, of type `SizeF`, specifies an offset for the shadow, which represents the position of a light source that creates the shadow.
- `blur`, of type `float`, represents the amount of blur to apply to the shadow.
- `color`, of type `Color`, defines the color of the shadow.

The following examples show how to add shadows to filled objects:

```

canvas.FillColor = Colors.Red;
canvas.SetShadow(new SizeF(10, 10), 4, Colors.Grey);
canvas.FillRectangle(10, 10, 90, 100);

canvas.FillColor = Colors.Green;
canvas.SetShadow(new SizeF(10, -10), 4, Colors.Grey);
canvas.FillEllipse(110, 10, 90, 100);

canvas.FillColor = Colors.Blue;
canvas.SetShadow(new SizeF(-10, 10), 4, Colors.Grey);
canvas.FillRoundedRectangle(210, 10, 90, 100, 25);

```

In these examples, shadows whose light sources are in different positions are added to the filled objects, with identical amounts of blur:



Draw dashed objects

`ICanvas` objects have a `StrokeDashPattern` property, of type `float[]`. This property is an array of `float` values that indicate the pattern of dashes and gaps that are to be used when drawing the stroke for an object. Each `float` in the array specifies the length of a dash or gap. The first item in the array specifies the length of a dash, while the second item in the array specifies the length of a gap. Therefore, `float` values with an even index value specify dashes, while `float` values with an odd index value specify gaps.

The following example shows how to draw a dashed square, using a regular dash:

```
canvas.StrokeColor = Colors.Red;
canvas.StrokeSize = 4;
canvas.StrokeDashPattern = new float[] { 2, 2 };
canvas.DrawRectangle(10, 10, 90, 100);
```

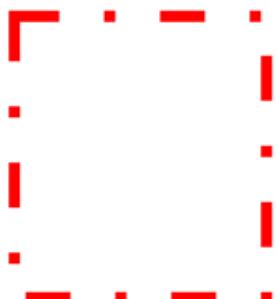
In this example, a square with a regular dashed stroke is drawn:



The following example shows how to draw a dashed square, using an irregular dash:

```
canvas.StrokeColor = Colors.Red;
canvas.StrokeSize = 4;
canvas.StrokeDashPattern = new float[] { 4, 4, 1, 4 };
canvas.DrawRectangle(10, 10, 90, 100);
```

In this example, a square with an irregular 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 start and end of a line.

`ICanvas` objects have a `strokeLineCap` property, of type `LineCap`, that describes the start and end of a line. The `LineCap` enumeration defines the following members:

- `Butt`, which represents a line with a square end, drawn to extend to the exact endpoint of the line. This is the default value of the `StrokeLineCap` property.
- `Round`, which represents a line with a rounded end.
- `Square`, which represents a line with a square end, drawn to extend beyond the endpoint to a distance equal to half the line width.

The following example shows how to set the `strokeLineCap` property:

```
canvas.StrokeSize = 10;
canvas.StrokeColor = Colors.Red;
canvas.StrokeLineCap = LineCap.Round;
canvas.DrawLine(10, 10, 110, 110);
```

In this example, the red line is rounded at the start and end of the line:



Control line joins

`ICanvas` objects have a `strokeLineJoin` property, of type `LineJoin`, that specifies the type of join that is used at the vertices of an object. The `LineJoin` enumeration defines the following members:

- `Miter`, which represents angular vertices that produce a sharp or clipped corner. This is the default value of the `StrokeLineJoin` property.
- `Round`, which represents rounded vertices that produce a circular arc at the corner.
- `Bevel`, which represents beveled vertices that produce a diagonal corner.

NOTE

When the `StrokeLineJoin` property is set to `Miter`, the `MiterLimit` property can be set to a `float` to limit the miter length of line joins in the object.

The following example shows how to set the `strokeLineJoin` property:

```

PathF path = new PathF();
path.MoveTo(10, 10);
path.LineTo(110, 50);
path.LineTo(10, 110);

canvas.StrokeSize = 20;
canvas.StrokeColor = Colors.Blue;
canvas.StrokeLineJoin = LineJoin.Round;
canvas.DrawPath(path);

```

In this example, the blue `PathF` object has rounded joins at its vertices:



Clip objects

Graphical objects that are drawn to an `ICanvas` can be clipped prior to drawing, with the following methods:

- `clipPath` clips an object so that only the area that's within the region of a `PathF` object will be visible.
- `clipRectangle` clips an object so that only the area that's within the region of a rectangle will be visible. The rectangle can be specified using `float` arguments, or by a `Rect` or `RectF` argument.
- `SubtractFromClip` clips an object so that only the area that's outside the region of a rectangle will be visible. The rectangle can be specified using `float` arguments, or by a `Rect` or `RectF` argument.

The following example shows how to use the `ClipPath` method to clip an image:

```

using Microsoft.Maui.Graphics.Platform;
...

IImage image;
var assembly = GetType().GetTypeInfo().Assembly;
using (var stream = assembly.GetManifestResourceStream("GraphicsViewDemos.Resources.Images.dotnet_bot.png"))
{
    image = PlatformImage.FromStream(stream);
}

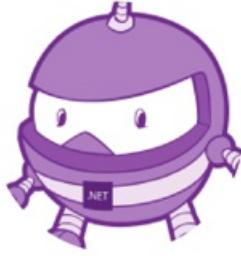
if (image != null)
{
    PathF path = new PathF();
    path.AppendCircle(100, 90, 80);
    canvas.ClipPath(path); // Must be called before DrawImage
    canvas.DrawImage(image, 10, 10, image.Width, image.Height);
}

```

WARNING

The `PlatformImage` type isn't supported on Windows.

In this example, the image is clipped using a `PathF` object that defines a circle that's centered at (100,90) with a radius of 80. The result is that only the part of the image within the circle is visible:



IMPORTANT

The `ClipPath` method has an overload that enables a `WindingMode` to be specified, which sets the fill algorithm that's used when clipping. For more information, see [Winding modes](#).

The following example shows how to use the `SubtractFromClip` method to clip an image:

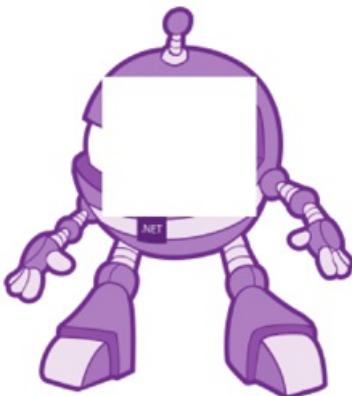
```
using Microsoft.Maui.Graphics.Platform;
...
IImage image;
var assembly = GetType().GetTypeInfo().Assembly;
using (var stream = assembly.GetManifestResourceStream("MyMauiApp.Resources.Images.dotnet_bot.png"))
{
    image = PlatformImage.FromStream(stream);
}

if (image != null)
{
    canvas.SubtractFromClip(60, 60, 90, 90);
    canvas.DrawImage(image, 10, 10, image.Width, image.Height);
}
```

WARNING

The `PlatformImage` type isn't supported on Windows.

In this example, the area defined by the rectangle that's specified by the arguments supplied to the `SubtractFromClip` method is clipped from the image. The result is that only the parts of the image outside the rectangle are visible:



Images

9/20/2022 • 4 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) graphics includes functionality to load, save, resize, and downsize images. Supported image formats are dependent on the underlying platform.

Images are represented by the `IImage` type, which defines the following properties:

- `Width`, of type `float`, that defines the width of an image.
- `Height`, of type `float`, that defines the height of an image.

An optional `ImageFormat` argument can be specified when loading and saving images. The `ImageFormat` enumeration defines `Png`, `Jpeg`, `Gif`, `Tiff`, and `Bmp` members. However, this argument is only used when the image format is supported by the underlying platform.

NOTE

.NET MAUI contains two different `IImage` interfaces. `Microsoft.Maui.Graphics.IImage` is used for image display, manipulation, and persistence when displaying graphics in a `GraphicsView`. `Microsoft.Maui.IImage` is the interface that abstracts the `Image` control.

Load an image

Image loading functionality is provided by the `GraphicsService` class. Images can be loaded from a stream by the `LoadFromStream` method, or from a byte array using the `LoadImageFromBytes` method.

The following example shows how to load an image:

```
using Microsoft.Maui.Graphics.Platform;
...
IImage image;
Assembly assembly = GetType().GetTypeInfo().Assembly;
using (Stream stream =
assembly.GetManifestResourceStream("GraphicsViewDemos.Resources.Images.dotnet_bot.png"))
{
    image = PlatformImage.FromStream(stream);
}

if (image != null)
{
    canvas.DrawImage(image, 10, 10, image.Width, image.Height);
}
```

WARNING

The `PlatformImage` type isn't supported on Windows.

In this example, the image is retrieved from the assembly, loaded as a stream, and displayed.

IMPORTANT

Loading an image that's embedded in an assembly requires the image to have its build action set to **Embedded Resource** rather than **MauiImage**.

Resize an image

Images can be resized using the `IImage.Resize` method, which requires `width` and `height` arguments, of type `float`, which represent the target dimensions of the image. The `Resize` method also accepts two optional arguments:

- A `ResizeMode` argument, that controls how the image will be resized to fit its target dimensions.
- A `bool` argument that controls whether the source image will be disposed after performing the resize operation. This argument defaults to `false`, indicating that the source image won't be disposed.

The `ResizeMode` enumeration defines the following members, which specify how to resize the image to the target size:

- `Fit`, which letterboxes the image so that it fits its target size.
- `Bleed`, which clips the image so that it fits its target size, while preserving its aspect ratio.
- `Stretch`, which stretches the image so it fills the available space. This can result in a change in the image aspect ratio.

The following example shows how to resize an image:

```
using Microsoft.Maui.Graphics.Platform;
...
IImage image;
Assembly assembly = GetType().GetTypeInfo().Assembly;
using (Stream stream =
assembly.GetManifestResourceStream("GraphicsViewDemos.Resources.Images.dotnet_bot.png"))
{
    image = PlatformImage.FromStream(stream);
}

if (image != null)
{
    IImage newImage = image.Resize(100, 60, ResizeMode.Stretch, true);
    canvas.DrawImage(newImage, 10, 10, newImage.Width, newImage.Height);
}
```

WARNING

The `PlatformImage` type isn't supported on Windows.

In this example, the image is retrieved from the assembly and loaded as a stream. The image is resized using the `Resize` method, with its arguments specifying the new size, and that it should be stretched to fill the available space. In addition, the source image is disposed. The resized image is then drawn at actual size at (10,10).

Downsize an image

Images can be downsized by one of the `IImage.Downsize` overloads. The first overload requires a single `float` value that represents the maximum width or height of the image, and downsizes the image while maintaining its aspect ratio. The second overload requires two `float` arguments, that represent the maximum width and

maximum height of the image.

The `Downsize` overloads also accept an optional `bool` argument that controls whether the source image should be disposed after performing the downsizing operation. This argument defaults to `false`, indicating that the source image won't be disposed.

The following example shows how to downsize an image:

```
using Microsoft.Maui.Graphics.Platform;
...
IImage image;
Assembly assembly = GetType().GetTypeInfo().Assembly;
using (Stream stream =
assembly.GetManifestResourceStream("GraphicsViewDemos.Resources.Images.dotnet_bot.png"))
{
    image = PlatformImage.FromStream(stream);
}

if (image != null)
{
    IImage newImage = image.Downsize(100, true);
    canvas.DrawImage(newImage, 10, 10, newImage.Width, newImage.Height);
}
```

WARNING

The `PlatformImage` type isn't supported on Windows.

In this example, the image is retrieved from the assembly and loaded as a stream. The image is downsized using the `Downsize` method, with the argument specifying that its largest dimension should be set to 100 pixels. In addition, the source image is disposed. The downsized image is then drawn at actual size at (10,10).

Save an image

Images can be saved by the `IImage.Save` and `IImage.SaveAsync` methods. Each method saves the `IImage` to a `Stream`, and enables optional `ImageFormat` and quality values to be specified.

The following example shows how to save an image:

```
using Microsoft.Maui.Graphics.Platform;
...
IImage image;
Assembly assembly = GetType().GetTypeInfo().Assembly;
using (Stream stream =
assembly.GetManifestResourceStream("GraphicsViewDemos.Resources.Images.dotnet_bot.png"))
{
    image = PlatformImage.FromStream(stream);
}

// Save image to a memory stream
if (image != null)
{
    IImage newImage = image.Downsize(150, true);
    using (MemoryStream memStream = new MemoryStream())
    {
        newImage.Save(memStream);
    }
}
```

WARNING

The `PlatformImage` type isn't supported on Windows.

In this example, the image is retrieved from the assembly and loaded as a stream. The image is downsized using the `Downsize` method, with the argument specifying that its largest dimension should be set to 150 pixels. In addition, the source image is disposed. The downsized image is then saved to a stream.

Paint graphical objects

9/20/2022 • 12 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) graphics includes the ability to paint graphical objects with solid colors, gradients, repeating images, and patterns.

The `Paint` class is an abstract class that paints an object with its output. Classes that derive from `Paint` describe different ways of painting an object. The following list describes the different paint types available in .NET MAUI graphics:

- `SolidPaint`, which paints an object with a solid color. For more information, see [Paint a solid color](#).
- `ImagePaint`, which paints an object with an image. For more information, see [Paint an image](#).
- `PatternPaint`, which paints an object with a pattern. For more information, see [Paint a pattern](#).
- `GradientPaint`, which paints an object with a gradient. For more information, see [Paint a gradient](#).

Instances of these types can be painted on an `ICanvas`, typically by using the `SetFillPaint` method to set the paint as the fill of a graphical object.

The `Paint` class also defines `BackgroundColor`, and `ForegroundColor` properties, of type `Color`, that can be used to optionally define background and foreground colors for a `Paint` object.

Paint a solid color

The `SolidPaint` class, that's derived from the `Paint` class, is used to paint a graphical object with a solid color.

The `SolidPaint` class defines a `Color` property, of type `Color`, which represents the color of the paint. The class also has an `IsTransparent` property that returns a `bool` that represents whether the color has an alpha value of less than 1.

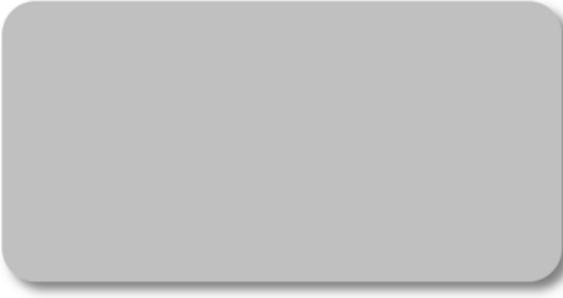
Create a SolidPaint object

The color of a `SolidPaint` object is typically specified through its constructor, using a `Color` argument:

```
SolidPaint solidPaint = new SolidPaint(Colors.Silver);

RectF solidRectangle = new RectF(100, 100, 200, 100);
canvas.SetFillPaint(solidPaint, solidRectangle);
canvas.SetShadow(new SizeF(10, 10), 10, Colors.Grey);
canvas.FillRoundedRectangle(solidRectangle, 12);
```

The `SolidPaint` object is specified as the first argument to the `SetFillPaint` method. Therefore, a filled rounded rectangle is painted with a silver `SolidPaint` object:



Alternatively, the color can be specified with the `Color` property:

```
SolidPaint solidPaint = new SolidPaint
{
    Color = Colors.Silver
};

RectF solidRectangle = new RectF(100, 100, 200, 100);
canvas.SetFillPaint(solidPaint, solidRectangle);
canvas.SetShadow(new SizeF(10, 10), 10, Colors.Grey);
canvas.FillRoundedRectangle(solidRectangle, 12);
```

Paint an image

The `ImagePaint` class, that's derived from the `Paint` class, is used to paint a graphical object with an image.

The `ImagePaint` class defines an `Image` property, of type `IImage`, which represents the image to paint. The class also has an `IsTransparent` property that returns `false`.

Create an `ImagePaint` object

To paint an object with an image, load the image and assign it to the `Image` property of the `ImagePaint` object.

NOTE

Loading an image that's embedded in an assembly requires the image to have its build action set to **Embedded Resource**.

The following example shows how to load an image and fill a rectangle with it:

```
using Microsoft.Maui.Graphics.Platform;
...

IImage image;
var assembly = GetType().GetTypeInfo().Assembly;
using (var stream = assembly.GetManifestResourceStream("GraphicsViewDemos.Resources.Images.dotnet_bot.png"))
{
    image = PlatformImage.FromStream(stream);
}

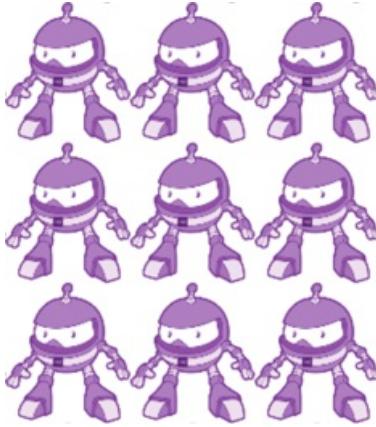
if (image != null)
{
    ImagePaint imagePaint = new ImagePaint
    {
        Image = image.Downsize(100)
    };
    canvas.SetFillPaint(imagePaint, RectF.Zero);
    canvas.FillRectangle(0, 0, 240, 300);
}
```

WARNING

The `PlatformImage` type isn't supported on Windows.

In this example, the image is retrieved from the assembly and loaded as a stream. The image is resized using the `Downsize` method, with the argument specifying that its largest dimension should be set to 100 pixels. For more information about downsizing an image, see [Downsize an image](#).

The `Image` property of the `ImagePaint` object is set to the downsized version of the image, and the `ImagePaint` object is set as the paint to fill an object with. A rectangle is then drawn that's filled with the paint:



NOTE

An `ImagePaint` object can also be created from an `IImage` object by the `AsPaint` extension method.

Alternatively, the `SetFillImage` extension method can be used to simplify the code:

```
if (image != null)
{
    canvas.SetFillImage(image.Downsize(100));
    canvas.FillRectangle(0, 0, 240, 300);
}
```

Paint a pattern

The `PatternPaint` class, that's derived from the `Paint` class, is used to paint a graphical object with a pattern.

The `PatternPaint` class defines a `Pattern` property, of type `IPattern`, which represents the pattern to paint. The class also has an `IsTransparent` property that returns a `bool` that represents whether the background or foreground color of the paint has an alpha value of less than 1.

Create a PatternPaint object

To paint an area with a pattern, create the pattern and assign it to the `Pattern` property of a `PatternPaint` object.

The following example shows how to create a pattern and fill an object with it:

```

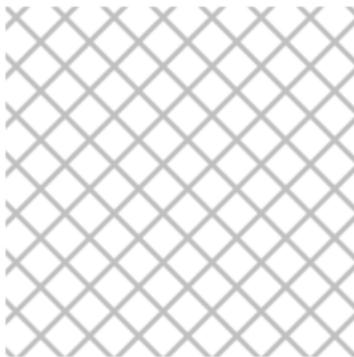
IPattern pattern;

// Create a 10x10 template for the pattern
using (PictureCanvas picture = new PictureCanvas(0, 0, 10, 10))
{
    picture.StrokeColor = Colors.Silver;
    picture.DrawLine(0, 0, 10, 10);
    picture.DrawLine(0, 10, 10, 0);
    pattern = new PicturePattern(picture.Picture, 10, 10);
}

// Fill the rectangle with the 10x10 pattern
PatternPaint patternPaint = new PatternPaint
{
    Pattern = pattern
};
canvas.SetFillPaint(patternPaint, RectF.Zero);
canvas.FillRectangle(10, 10, 250, 250);

```

In this example, the pattern is a 10x10 area that contains a diagonal line from (0,0) to (10,10), and a diagonal line from (0,10) to (10,0). The `Pattern` property of the `PatternPaint` object is set to the pattern, and the `PatternPaint` object is set as the paint to fill an object with. A rectangle is then drawn that's filled with the paint:



NOTE

A `PatternPaint` object can also be created from a `PicturePattern` object by the `AsPaint` extension method.

Paint a gradient

The `GradientPaint` class, that's derived from the `Paint` class, is an abstract base class that describes a gradient, which is composed of gradient steps. A `GradientPaint` paints a graphical object with multiple colors that blend into each other along an axis. Classes that derive from `GradientPaint` describe different ways of interpreting gradients stops, and .NET MAUI graphics provides the following gradient paints:

- `LinearGradientPaint`, which paints an object with a linear gradient. For more information, see [Paint a linear gradient](#).
- `RadialGradientPaint`, which paints an object with a radial gradient. For more information, see [Paint a radial gradient](#).

The `GradientPaint` class defines the `GradientStops` property, of type `GradientStop`, which represents the brush's gradient stops, each of which specifies a color and an offset along the gradient axis.

Gradient stops

Gradient stops are the building blocks of a gradient, 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.
- `offset`, of type `float`, which represents the location of the gradient stop within the gradient vector. 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.

IMPORTANT

The coordinate system used by gradients is relative to a bounding box for the graphical object. 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.

Gradient stops can be added to a `GradientPoint` object with the `AddOffset` method.

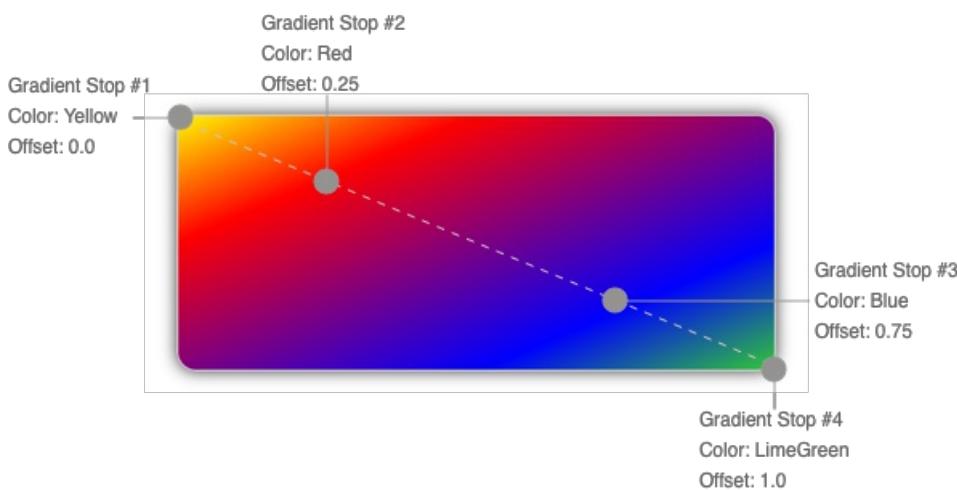
The following example creates a diagonal `LinearGradientPaint` with four colors:

```
LinearGradientPaint linearGradientPaint = new LinearGradientPaint
{
    StartColor = Colors.Yellow,
    EndColor = Colors.Green,
    StartPoint = new Point(0, 0),
    EndPoint = new Point(1, 1)
};

linearGradientPaint.AddOffset(0.25f, Colors.Red);
linearGradientPaint.AddOffset(0.75f, Colors.Blue);

RectF linearRectangle = new RectF(10, 10, 200, 100);
canvas.SetFillPaint(linearGradientPaint, linearRectangle);
canvas.SetShadow(new SizeF(10, 10), 10, Colors.Grey);
canvas.FillRoundedRectangle(linearRectangle, 12);
```

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 offset of 1.0. The points between the third and fourth gradient stops gradually change from blue to lime green.

Paint a linear gradient

The `LinearGradientPaint` class, that's derived from the `GradientPaint` class, paints a graphical object with a linear gradient. A linear gradient 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 [Paint a gradient](#).

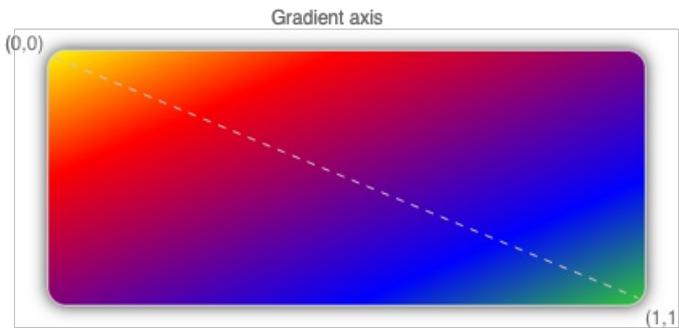
The `LinearGradientPaint` class defines the following properties:

- `StartPoint`, of type `Point`, which represents the starting two-dimensional coordinates of the linear gradient. The class constructor initializes this property to (0,0).
- `EndPoint`, of type `Point`, which represents the ending two-dimensional coordinates of the linear gradient. The class constructor initializes this property to (1,1).

Create a `LinearGradientPaint` object

A linear gradient's gradient stops are positioned along the gradient axis. The orientation and size of the gradient axis can be changed using the `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 graphical object being painted. (0,0) represents the top-left corner of the object being painted, and (1,1) represents the bottom-right corner of the object 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 `LinearGradientPaint` object and set its `startColor` and `endColor` properties. Then, set its `EndPoint` to (1,0).

The following example shows how to create a horizontal `LinearGradientPaint`:

```
LinearGradientPaint linearGradientPaint = new LinearGradientPaint
{
    StartColor = Colors.Yellow,
    EndColor = Colors.Green,
    // StartPoint is already (0,0)
    EndPoint = new Point(1, 0)
};

RectF linearRectangle = new RectF(10, 10, 200, 100);
canvas.SetFillPaint(linearGradientPaint, linearRectangle);
canvas.SetShadow(new SizeF(10, 10), 10, Colors.Grey);
canvas.FillRoundedRectangle(linearRectangle, 12);
```

In this example, the rounded rectangle is painted with a linear gradient that interpolates horizontally from

yellow to green:



Create a vertical linear gradient

To create a vertical linear gradient, create a `LinearGradientPaint` object and set its `StartColor` and `EndColor` properties. Then, set its `EndPoint` to `(0,1)`.

The following example shows how to create a vertical `LinearGradientPaint`:

```
LinearGradientPaint linearGradientPaint = new LinearGradientPaint
{
    StartColor = Colors.Yellow,
    EndColor = Colors.Green,
    // StartPoint is already (0,0)
    EndPoint = new Point(0, 1)
};

RectF linearRectangle = new RectF(10, 10, 200, 100);
canvas.SetFillPaint(linearGradientPaint, linearRectangle);
canvas.SetShadow(new SizeF(10, 10), 10, Colors.Grey);
canvas.FillRoundedRectangle(linearRectangle, 12);
```

In this example, the rounded rectangle is painted with a linear gradient that interpolates vertically from yellow to green:



Create a diagonal linear gradient

To create a diagonal linear gradient, create a `LinearGradientPaint` object and set its `StartColor` and `EndColor` properties.

The following example shows how to create a diagonal `LinearGradientPaint`:

```

LinearGradientPaint linearGradientPaint = new LinearGradientPaint
{
    StartColor = Colors.Yellow,
    EndColor = Colors.Green,
    // StartPoint is already (0,0)
    // EndPoint is already (1,1)
};

RectF linearRectangle = new RectF(10, 10, 200, 100);
canvas.SetFillPaint(linearGradientPaint, linearRectangle);
canvas.SetShadow(new SizeF(10, 10), 10, Colors.Grey);
canvas.FillRoundedRectangle(linearRectangle, 12);

```

In this example, the rounded rectangle is painted with a linear gradient that interpolates diagonally from yellow to green:



Paint a radial gradient

The `RadialGradientPaint` class, that's derived from the `GradientPaint` class, paints a graphical object with a radial gradient. A radial gradient 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 [Paint a gradient](#).

The `RadialGradientPaint` class defines the following properties:

- `Center`, of type `Point`, which represents the center point of the circle for the radial gradient. The class constructor initializes this property to (0.5,0.5).
- `Radius`, of type `double`, which represents the radius of the circle for the radial gradient. The class constructor initializes this property to 0.5.

Create a RadialGradientPaint object

A radial gradient'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 `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 `RadialGradientPaint` object and set its `StartColor` and `EndColor` properties. Then, set its `Center` and `Radius` properties.

The following example shows how to create a centered `RadialGradientPaint`:

```

RadialGradientPaint radialGradientPaint = new RadialGradientPaint
{
    StartColor = Colors.Red,
    EndColor = Colors.DarkBlue
    // Center is already (0.5,0.5)
    // Radius is already 0.5
};

RectF radialRectangle = new RectF(10, 10, 200, 100);
canvas.SetFillPaint(radialGradientPaint, radialRectangle);
canvas.SetShadow(new SizeF(10, 10), 10, Colors.Grey);
canvas.FillRoundedRectangle(radialRectangle, 12);

```

In this example, the rounded rectangle is painted with a radial gradient that interpolates from red to dark blue. The center of the radial gradient is positioned in the center of the rectangle:



The following example moves the center of the radial gradient to the top-left corner of the rectangle:

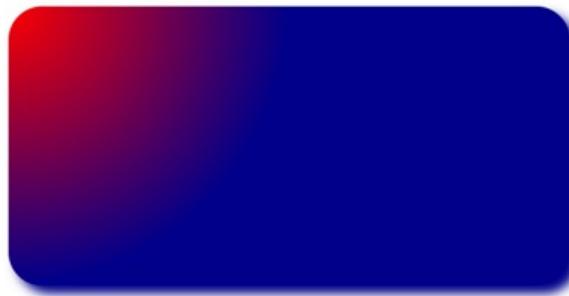
```

RadialGradientPaint radialGradientPaint = new RadialGradientPaint
{
    StartColor = Colors.Red,
    EndColor = Colors.DarkBlue,
    Center = new Point(0.0, 0.0)
    // Radius is already 0.5
};

RectF radialRectangle = new RectF(10, 10, 200, 100);
canvas.SetFillPaint(radialGradientPaint, radialRectangle);
canvas.SetShadow(new SizeF(10, 10), 10, Colors.Grey);
canvas.FillRoundedRectangle(radialRectangle, 12);

```

In this example, the rounded rectangle is painted with a radial gradient that interpolates from red to dark blue. The center of the radial gradient is positioned in the top-left of the rectangle:



The following example moves the center of the radial gradient to the bottom-right corner of the rectangle:

```
RadialGradientPaint radialGradientPaint = new RadialGradientPaint
{
    StartColor = Colors.Red,
    EndColor = Colors.DarkBlue,
    Center = new Point(1.0, 1.0)
    // Radius is already 0.5
};

RectF radialRectangle = new RectF(10, 10, 200, 100);
canvas.SetFillPaint(radialGradientPaint, radialRectangle);
canvas.SetShadow(new SizeF(10, 10), 10, Colors.Grey);
canvas.FillRoundedRectangle(radialRectangle, 12);
```

In this example, the rounded rectangle is painted with a radial gradient that interpolates from red to dark blue. The center of the radial gradient is positioned in the bottom-right of the rectangle:



Transforms

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) graphics supports traditional graphics transforms, which are implemented as methods on the `ICanvas` object. Mathematically, transforms alter the coordinates and sizes that you specify in `ICanvas` drawing methods, when the graphical objects are rendered.

The following transforms are supported:

- *Translate* to shift coordinates from one location to another.
- *Scale* to increase or decrease coordinates and sizes.
- *Rotate* to rotate coordinates around a point.
- *Skew* to shift coordinates horizontally or vertically.

These transforms are known as *affine* transforms. Affine transforms always preserve parallel lines and never cause a coordinate or size to become infinite.

The .NET MAUI `VisualElement` class also supports the following transform properties: `TranslationX`, `TranslationY`, `Scale`, `Rotation`, `RotationX`, and `RotationY`. However, there are several differences between `Microsoft.Maui.Graphics` transforms and `Microsoft.Maui.Controls.VisualElement` transforms:

- `Microsoft.Maui.Graphics` transforms are methods, while `VisualElement` transforms are properties. Therefore, `Microsoft.Maui.Graphics` transforms perform an operation while `VisualElement` transforms set a state. `Microsoft.Maui.Graphics` transforms apply to subsequently drawn graphics objects, but not to graphics objects that are drawn before the transform is applied. In contrast, a `VisualElement` transform applies to a previously rendered element as soon as the property is set. Therefore, `Microsoft.Maui.Graphics` transforms are cumulative as methods are called, while `VisualElement` transforms are replaced when the property is set with another value.
- `Microsoft.Maui.Graphics` transforms are applied to the `ICanvas` object, while `VisualElement` transforms are applied to a `VisualElement` derived object.
- `Microsoft.Maui.Graphics` transforms are relative to the upper-left corner of the `ICanvas`, while `VisualElement` transforms are relative to the upper-left corner of the `VisualElement` to which they're applied.

Translate transform

The translate transform shifts graphical objects in the horizontal and vertical directions. Translation can be considered unnecessary because the same result can be accomplished by changing the coordinates of the drawing method you're using. However, when displaying a path, all the coordinates are encapsulated in the path, and so it's often easier to apply a translate transform to shift the entire path.

The `Translate` method requires `x` and `y` arguments of type `float` that cause subsequently drawn graphic objects to be shifted horizontally and vertically. Negative `x` values move an object to the left, while positive values move an object to the right. Negative `y` values move up an object, while positive values move down an object.

A common use of the translate transform is for rendering a graphical object that has been originally created using coordinates that are convenient for drawing. The following example creates a `PathF` object for an 11-pointed star:

```

PathF path = new PathF();
for (int i = 0; i < 11; i++)
{
    double angle = 5 * i * 2 * Math.PI / 11;
    PointF point = new PointF(100 * (float)Math.Sin(angle), -100 * (float)Math.Cos(angle));

    if (i == 0)
        path.MoveTo(point);
    else
        path.LineTo(point);
}

canvas.FillColor = Colors.Red;
canvas.Translate(150, 150);
canvas.FillPath(path);

```

The center of the star is at (0,0), and the points of the star are on a circle surrounding that point. Each point is a combination of sine and cosine values of an angle that increases by 5/11ths of 360 degrees. The radius of the circle is set as 100. If the `PathF` object is displayed without any transforms, the center of the star will be positioned at the upper-left corner of the `ICanvas`, and only a quarter of it will be visible. Therefore, a translate transform is used to shift the star horizontally and vertically to (150,150):



Scale transform

The scale transform changes the size of a graphical object, and can also often cause coordinates to move when a graphical object is made larger.

The `Scale` method requires `x` and `y` arguments of type `float` that let you specify different values for horizontal and vertical scaling, otherwise known as *anisotropic* scaling. The values of `x` and `y` have a significant 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.

The following example demonstrates the `Scale` method:

```

canvas.StrokeColor = Colors.Red;
canvas.StrokeSize = 4;
canvas.StrokeDashPattern = new float[] { 2, 2 };
canvas.FontColor = Colors.Blue;
canvas.FontSize = 18;

canvas.DrawRoundedRectangle(50, 50, 80, 20, 5);
canvas.DrawString(".NET MAUI", 50, 50, 80, 20, HorizontalAlignment.Left, VerticalAlignment.Top);

canvas.Scale(2, 2);
canvas.DrawRoundedRectangle(50, 100, 80, 20, 5);
canvas.DrawString(".NET MAUI", 50, 100, 80, 20, HorizontalAlignment.Left, VerticalAlignment.Top);

```

In this example, ".NET MAUI" is displayed inside a rounded rectangle stroked with a dashed line. The same graphical objects drawn after the `Scale` call increase in size proportionally:



The text and the rounded rectangle are both subject to the same scaling factors.

NOTE

Anisotropic scaling causes the stroke size to become different for lines aligned with the horizontal and vertical axes.

Order matters when you combine `Translate` and `Scale` calls. If the `Translate` call comes after the `Scale` call, the translation factors are scaled by the scaling factors. If the `Translate` call comes before the `Scale` call, the translation factors aren't scaled.

Rotate transform

The rotate transform rotates a graphical object around a point. Rotation is clockwise for increasing angles. Negative angles and angles greater than 360 degrees are allowed.

There are two `Rotate` overloads. The first requires a `degrees` argument of type `float` that defines the rotation angle, and centers the rotation around the upper-left corner of the canvas (0,0). The following example demonstrates this `Rotate` method:

```
canvas.FontColor = Colors.Blue;
canvas.FontSize = 18;

canvas.Rotate(45);
canvas.DrawString(".NET MAUI", 50, 50, HorizontalAlignment.Left);
```

In this example, ".NET MAUI" is rotated 45 degrees clockwise:



Alternatively, graphical objects can be rotated centered around a specific point. To rotate around a specific point, use the `Rotate` overload that accepts `degrees`, `x`, and `y` arguments of type `float`:

```
canvas.FontColor = Colors.Blue;
canvas.FontSize = 18;

canvas.Rotate(45, dirtyRect.Center.X, dirtyRect.Center.Y);
canvas.DrawString(".NET MAUI", dirtyRect.Center.X, dirtyRect.Center.Y, HorizontalAlignment.Left);
```

In this example, `.NET MAUI` is rotated 45 degrees around the center of the canvas.

Combine transforms

The simplest way to combine transforms is to begin with global transforms, followed by local transforms. For example, translation, scaling, and rotation can be combined to draw an analog clock. The clock can be drawn using an arbitrary coordinate system based on a circle that's centered at (0,0) with a radius of 100. Translation and scaling expand and center the clock on the canvas, and rotation can then be used to draw the minute and hour marks of the clock and to rotate the hands:

```
canvas.StrokeLineCap = LineCap.Round;
canvas.FillColor = Colors.Gray;

// Translation and scaling
canvas.Translate(dirtyRect.Center.X, dirtyRect.Center.Y);
float scale = Math.Min(dirtyRect.Width / 200f, dirtyRect.Height / 200f);
canvas.Scale(scale, scale);

// Hour and minute marks
for (int angle = 0; angle < 360; angle += 6)
{
    canvas.FillCircle(0, -90, angle % 30 == 0 ? 4 : 2);
    canvas.Rotate(6);
}

DateTime now = DateTime.Now;

// Hour hand
canvas.StrokeSize = 20;
canvas.SaveState();
canvas.Rotate(30 * now.Hour + now.Minute / 2f);
canvas.DrawLine(0, 0, 0, -50);
canvas.RestoreState();

// Minute hand
canvas.StrokeSize = 10;
canvas.SaveState();
canvas.Rotate(6 * now.Minute + now.Second / 10f);
canvas.DrawLine(0, 0, 0, -70);
canvas.RestoreState();

// Second hand
canvas.StrokeSize = 2;
canvas.SaveState();
canvas.Rotate(6 * now.Second);
canvas.DrawLine(0, 10, 0, -80);
canvas.RestoreState();
```

In this example, the `Translate` and `Scale` calls apply globally to the clock, and so are called before the `Rotate` method

There are 60 marks of two different sizes that are drawn in a circle around the clock. The `FillCircle` call draws that circle at (0,-90), which relative to the center of the clock corresponds to 12:00. The `Rotate` call increments the rotation angle by 6 degrees after every tick mark. The `angle` variable is used solely to determine if a large circle or a small circle is drawn. Finally, the current time is obtained and rotation degrees are calculated for the hour, minute, and second hands. Each hand is drawn in the 12:00 position so that the rotation angle is relative to that position:



Concatenate transforms

A transform can be described in terms of a 3×3 affine transformation matrix, which performs transformations in 2D space. The following table shows the structure of a 3×3 affine transformation matrix:

M11

M12

0.0

M21

M22

0.0

M31

M32

1.0

An affine transformation matrix has its final column equal to $(0,0,1)$, so only members in the first two columns need to be specified. Therefore, the 3×3 matrix is represented by the `Matrix3x2` struct, from the `System.Numerics` namespace, which is a collection of three rows and two columns of `float` values.

The six cells in the first two columns of the transform matrix represent values that performing scaling, shearing, and translation:

ScaleX

ShearY

0.0

ShearX

ScaleY

0.0

TranslateX

TranslateY

1.0

For example, if you change the `M31` value to 100, you can use it to translate a graphical object 100 pixels along

the x-axis. If you change the `M22` value to 3, you can use it to stretch a graphical object to three times its current height. If you change both values, you move the graphical object 100 pixels along the x-axis and stretch its height by a factor of 3.

You can define a new transform matrix with the `Matrix3x2` constructor. The advantage of specifying transforms with a transform matrix is that composite transforms can be applied as a single transform, which is referred to as *concatenation*. The `Matrix3x2` struct also defines methods that can be used to manipulate matrix values.

The only `ICanvas` method that accepts a `Matrix3x2` argument is the `ConcatenateTransform` method, which combines multiple transforms into a single transform. The following example shows how to use this method to transform a `PathF` object:

```
PathF path = new PathF();
for (int i = 0; i < 11; i++)
{
    double angle = 5 * i * 2 * Math.PI / 11;
    PointF point = new PointF(100 * (float)Math.Sin(angle), -100 * (float)Math.Cos(angle));

    if (i == 0)
        path.MoveTo(point);
    else
        path.LineTo(point);
}

Matrix3x2 transform = new Matrix3x2(1.5f, 1, 0, 1, 150, 150);
canvas.ConcatenateTransform(transform);
canvas.FillColor = Colors.Red;
canvas.FillPath(path);
```

In this example, the `PathF` object is scaled and sheared on the x-axis, and translated on the x-axis and the y-axis.

Winding modes

9/20/2022 • 2 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) graphics provides a `WindingMode` enumeration that enables you to specify the fill algorithm to be used by the `FillPath` method. Contours in a path can overlap, and any enclosed area can potentially be filled, but you might not want to fill all the enclosed areas. For more information about paths, see [Draw a path](#).

The `WindingMode` enumeration defines `NonZero` and `EvenOdd` members. Each member represents a different algorithm for determining whether a point is in the fill region of an enclosed area.

NOTE

The `ClipPath` method has an overload that enables a `WindingMode` argument to be specified. By default, this argument is set to `WindingMode.NonZero`.

NonZero

The `NonZero` winding mode draws a hypothetical ray from the point to infinity in any direction and then examines the places where a path contour crosses the ray. The count starts at zero and is incremented each time a contour crosses the ray from left to right and decremented each time a contour crosses the ray from right to left. If the count of crossings is zero, the area isn't filled. Otherwise, the area is filled.

The following example fills a five-pointed star using the `NonZero` winding mode:

```
float radius = 0.45f * Math.Min(dirtyRect.Width, dirtyRect.Height);

PathF path = new PathF();
path.MoveTo(dirtyRect.Center.X, dirtyRect.Center.Y - radius);

for (int i = 1; i < 5; i++)
{
    double angle = i * 4 * Math.PI / 5;
    path.LineTo(new PointF(radius * (float)Math.Sin(angle) + dirtyRect.Center.X, -radius *
(float)Math.Cos(angle) + dirtyRect.Center.Y));
}
path.Close();

canvas.StrokeSize = 15;
canvas.StrokeLineJoin = LineJoin.Round;
canvas.StrokeColor = Colors.Red;
canvas.FillColor = Colors.Blue;
canvas.FillPath(path); // Overload automatically uses a NonZero winding mode
canvas.DrawPath(path);
```

In this example, the path is drawn twice. The `FillPath` method is used to fill the path with blue, while the `DrawPath` method outlines the path with a red stroke. The `FillPath` overload used omits the `WindingMode` argument, and instead automatically uses the `NonZero` winding mode. This results in all the enclosed areas of the path being filled:



NOTE

For many paths, the `NonZero` winding mode often fills all the enclosed areas of a path.

EvenOdd

The `EvenOdd` winding mode draws a hypothetical ray from the point to infinity in any direction and counts the number of path contours that the ray crosses. If this number is odd, then the area is filled. Otherwise, the area isn't filled.

The following example fills a five-pointed star using the `EvenOdd` winding mode:

```
float radius = 0.45f * Math.Min(dirtyRect.Width, dirtyRect.Height);

PathF path = new PathF();
path.MoveTo(dirtyRect.Center.X, dirtyRect.Center.Y - radius);

for (int i = 1; i < 5; i++)
{
    double angle = i * 4 * Math.PI / 5;
    path.LineTo(new PointF(radius * (float)Math.Sin(angle) + dirtyRect.Center.X, -radius *
(float)Math.Cos(angle) + dirtyRect.Center.Y));
}
path.Close();

canvas.StrokeSize = 15;
canvas.StrokeLineJoin = LineJoin.Round;
canvas.StrokeColor = Colors.Red;
canvas.FillColor = Colors.Blue;
canvas.FillPath(path, WindingMode.EvenOdd);
canvas.DrawPath(path);
```

In this example, the path is drawn twice. The `FillPath` method is used to fill the path with blue, while the `DrawPath` method outlines the path with a red stroke. The `FillPath` overload used specifies that the `EvenOdd` winding mode is used. This mode results in the central area of the star not being filled:



Change a .NET MAUI app icon

9/20/2022 • 9 minutes to read • [Edit Online](#)

Every app has a logo icon that represents it, and that icon typically appears in multiple places. For example, on iOS the app icon appears on the Home screen and throughout the system, such as in Settings, notifications, and search results, and in the App Store. On Android, the app icon appears as a launcher icon and throughout the system, such as on the action bar, notifications, and in the Google Play Store. On Windows, the app icon appears in the app list in the start menu, the taskbar, the app's tile, and in the Microsoft Store.

In a .NET Multi-platform App UI (.NET MAUI) app project, an app icon can be specified in a single location in your app project. At build time, this icon can be automatically resized to the correct resolution for the target platform and device, and added to your app package. This avoids having to manually duplicate and name the app icon on a per platform basis. By default, bitmap (non-vector) image formats aren't automatically resized by .NET MAUI.

A .NET MAUI app icon can use any of the standard platform image formats, including Scalable Vector Graphics (SVG) files.

IMPORTANT

.NET MAUI converts SVG files to Portable Network Graphic (PNG) files. Therefore, when adding an SVG file to your .NET MAUI app project, it should be referenced from XAML or C# with a *.png* extension. The only reference to the SVG file should be in your project file.

Change the icon

In your project file, the `<MauiIcon>` item designates the icon to use for your app. You may only have one icon defined for your app. Any subsequent `<MauiIcon>` items are ignored.

To comply with Android resource naming rules, app icon filenames must be lowercase, start and end with a letter character, and contain only alphanumeric characters or underscores. For more information, see [App resources overview](#) on developer.android.com.

The icon defined by your app can be composed of a single image, by specifying the file as the `Include` attribute:

```
<ItemGroup>
  <MauiIcon Include="Resources\AppIcon\appicon.svg" />
</ItemGroup>
```

IMPORTANT

Only the first `<MauiIcon>` item defined in the project file is processed by .NET MAUI. If you want to use a different file as the icon, first delete the existing icon from your project, and then add the new icon. Next, in the **Solution Explorer** pane, select the file, and then in the **Properties** pane, set the **Build Action** to **MauiIcon**. Instead of adding a new icon file to the project, consider replacing the existing icon file instead.

After changing the icon file, you may need to clean the project in Visual Studio. To clean the project, right-click on the project file in the **Solution Explorer** pane, and select **Clean**. You also may need to uninstall the app from the target platform you're testing with.

Caution

If you don't clean the project and uninstall the app from the target platform, you may not see your new icon.

After changing the icon, review the [Platform specific configuration](#) information.

Composed icon

Alternatively, the app icon can be composed of two images, one image representing the background and another representing the foreground. Since icons are transformed into PNG files, the composed app icon will be first layered with the background image, typically an image of a pattern or solid color, followed by the foreground image. In this case, the `Include` attribute represents the icon background image, and the `Foreground` attribute represents the foreground image:

```
<ItemGroup>
    <MauiIcon Include="Resources\AppIcon\appicon.svg" ForegroundFile="Resources\AppIcon\appiconfg.svg" />
</ItemGroup>
```

IMPORTANT

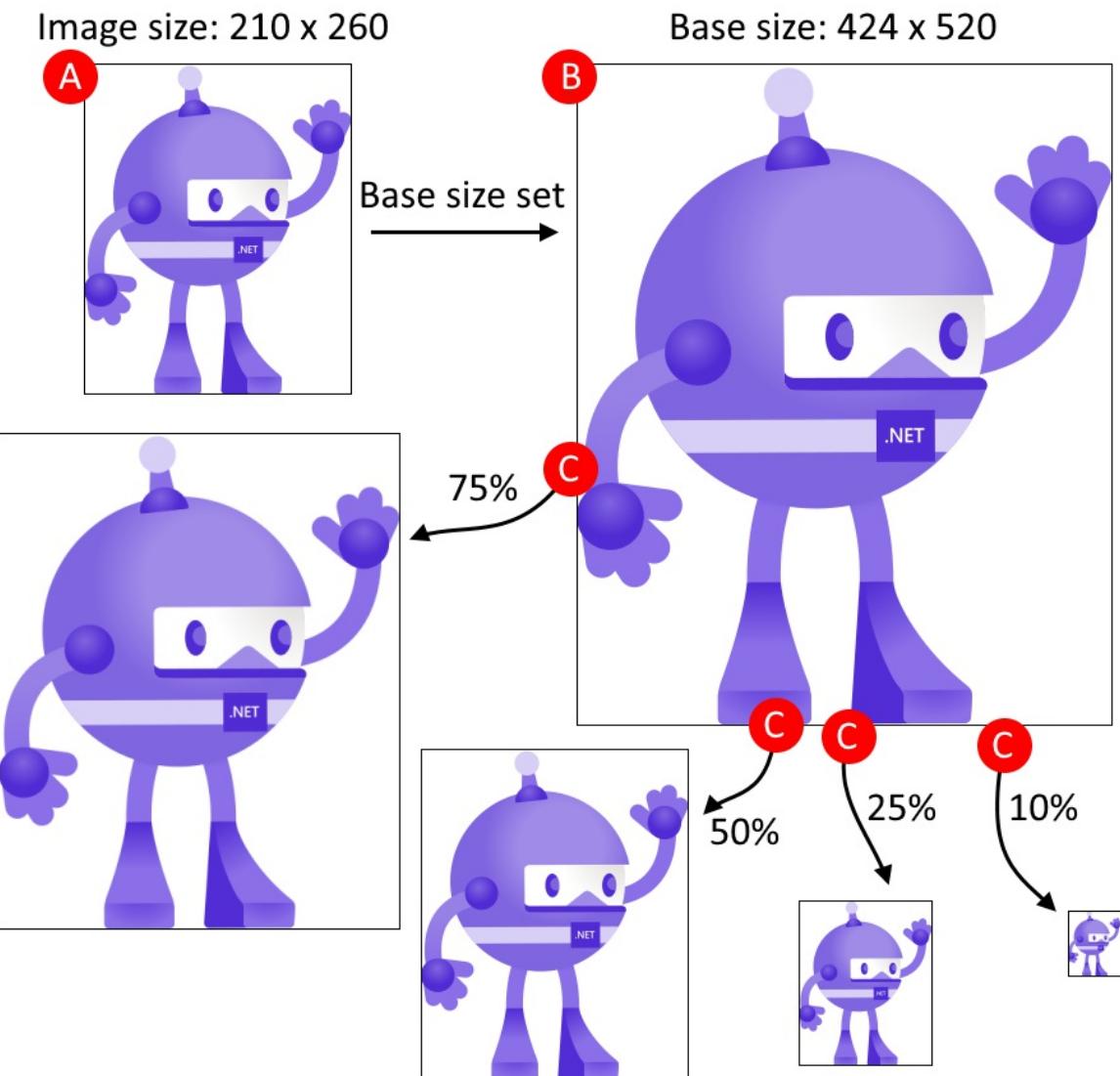
The background image (`Include` attribute) must be specified for the `<MauiIcon>` item. The foreground image (`ForegroundFile` attribute) is optional.

Set the base size

.NET MAUI uses your icon across multiple platforms and devices, and attempts to resize the icon according for each platform and device. The app icon is also used for different purposes, such as a store entry for your app or the icon used to represent the app after it's installed on a device.

The base size of your icon represents baseline density of the image, and is effectively the 1.0 scale factor that all other sizes are derived. If you don't specify the base size for a bitmap-based app icon, such as a PNG file, the image isn't resized. If you don't specify the base size for a vector-based app icon, such as an SVG file, the dimensions specified in the image are used as the base size. To stop a vector image from being resized, set the `Resize` attribute to `false`.

The following figure illustrates how base size affects an image:



The process shown in the previous figure follows these steps:

- A: The image is added as the .NET MAUI icon and has dimensions of 210x260, and the base size is set to 424x520.
- B: .NET MAUI automatically scales the image to match the base size of 424x520.
- C: As different target platforms require different sizes of the image, .NET MAUI automatically scales the image from the base size to different sizes.

TIP

Use an SVG image as your icon. SVG images can upscale to larger sizes and still look crisp and clean. Bitmap-based images, such as a PNG or JPG image, look blurry when upscaled.

The base size is specified with the `BaseSize="W,H"` attribute, where `W` is the width of the icon and `H` is the height of the icon. The value specified as the base size must be divisible by 8. The following example sets the base size:

```
<ItemGroup>
  <MauiIcon Include="Resources\AppIcon\appicon.png" BaseSize="128,128" />
</ItemGroup>
```

And the following example stops the automatic resizing of a vector-based image:

```
<ItemGroup>
  <MauiIcon Include="Resources\AppIcon\appicon.svg" Resize="false" />
</ItemGroup>
```

Recolor the background

If the background image used in composing the app icon uses transparency, it can be recolored by specifying the `Color` attribute on the `<MauiIcon>`. The following example sets the background color of the app icon to red:

```
<ItemGroup>
  <MauiIcon Include="Resources\AppIcon\appicon.svg" Color="#FF0000" />
</ItemGroup>
```

Color values can be specified in hexadecimal, using the format: `#RRGGBB` or `#AARRGGBB`. The value of `RR` represents the red channel, `GG` the green channel, `BB` the blue channel, and `AA` the alpha channel. Instead of a hexadecimal value, you may use a named .NET MAUI color, such as `Red` or `PaleVioletRed`.

Recolor the foreground

If the app icon is composed of a background (`Include`) image and a foreground (`ForegroundFile`) image, the foreground image can be tinted. To tint the foreground image, specify a color with the `TintColor` attribute. The following example tints the foreground image yellow:

```
<ItemGroup>
  <MauiIcon Include="Resources\AppIcon\appicon.png" ForegroundFile="Resources\AppIcon\appiconfg.svg"
  TintColor="Yellow" />
</ItemGroup>
```

Color values can be specified in hexadecimal, using the format: `#RRGGBB` or `#AARRGGBB`. The value of `RR` represents the red channel, `GG` the green channel, `BB` the blue channel, and `AA` the alpha channel. Instead of a hexadecimal value, you may use a named .NET MAUI color, such as `Red` or `PaleVioletRed`.

Use a different icon per platform

If you want to use different icon resources or settings per platform, add the `Condition` attribute to the `<MauiIcon>` item, and query for the specific platform. If the condition is met, the `<MauiIcon>` item is processed. Only the first valid `<MauiIcon>` item is used by .NET MAUI, so all conditional items should be declared first, followed by a default `<MauiIcon>` item without a condition. The following XML demonstrates declaring a specific icon for Windows and a fallback icon for all other platforms:

```
<ItemGroup>
  <!-- App icon for Windows -->
  <MauiIcon Condition="$([MSBuild]::GetTargetPlatformIdentifier('$(TargetFramework)')) == 'windows'"
  Include="Resources\AppIcon\backicon.png" ForegroundFile="Resources\AppIcon\appiconfg.svg"
  TintColor="#40FF00FF" />

  <!-- App icon for all other platforms -->
  <MauiIcon Include="Resources\AppIcon\appicon.png" ForegroundFile="Resources\AppIcon\appiconfg.svg"
  TintColor="Yellow" />
</ItemGroup>
```

You can set the target platform by changing the value compared in the condition to one of the following values:

- `'ios'`

- 'maccatalyst'
- 'android'
- 'windows'

For example, a condition that targets Android would be

```
Condition="[$([MSBuild]::GetTargetPlatformIdentifier('$(TargetFramework)')) == 'android'" .
```

Platform-specific configuration

While the project file declares which resources the app icon is composed from, you're still required to update the individual platform configurations with reference to those app icons. The following information describes these platform-specific settings.

- [Android](#)
- [iOS & macOS](#)
- [Windows](#)

The icon Android uses is specified in the Android manifest, which is located at

`Platforms\Android\AndroidManifest.xml`. The `manifest/application` node contains two attributes to define the icon: `android:icon` and `android:roundIcon`. The values of these two attributes follow this format:

`@mipmap/{name}` and `@mipmap/{name}_round`, respectively. The value for `{name}` is derived from the .NET MAUI project file's `<MauiIcon>` item, specifically the file name defined by the `Include` attribute, without its path or extension.

Consider the following example, which defines the resource `Resources\AppIcon\healthapp.png` as the icon:

```
<ItemGroup>
    <MauiIcon Include="Resources\AppIcon\healthapp.png" ForegroundFile="Resources\AppIcon\appiconfg.svg"
    TintColor="Yellow" />
</ItemGroup>
```

The transformed name, the resource without the path or extension, is `healthapp`. The values for `android:icon` and `android:roundIcon` would be `@mipmap/healthapp` and `@mipmap/healthapp_round`, respectively. The android manifest should be updated to match `healthapp` as the icon:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <application android:allowBackup="true" android:icon="@mipmap/healthapp"
    android:roundIcon="@mipmap/healthapp_round" android:supportsRtl="true"></application>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

TIP

Instead of creating new image files for the icon, simply replace the two image files provided by the .NET MAUI template: `Resources\AppIcon\appicon.svg` for the background and `Resources\AppIcon\appiconfg.svg` for the foreground.

Adaptive launcher

.NET MAUI supports creating an adaptive launcher icon on Android 8.0 and higher, from the app icon. Adaptive launcher icons can display as various shapes across different device models, including circular and square. For more information about adaptive icons, see the [Android developer guide: Adaptive icons](#).

Adaptive launcher icons are composed icons, using a background layer and a foreground layer, and an optional

scaling value. For more information, see the [Composed icon section](#). If the composed icon is defined, by specifying the `ForegroundFile` attribute, an adaptive launcher icon will be generated. The following XML demonstrates defining an icon that is used as an adaptive launcher icon:

```
<ItemGroup>
    <MauiIcon Include="Resources\AppIcon\appicon.svg" ForegroundFile="Resources\AppIcon\appiconfg.svg"
    ForegroundScale="0.65" Color="#512BD4" />
</ItemGroup>
```

The `ForegroundScale` attribute can be optionally specified to change the scaling of the foreground layer being rendered over the background layer.

Add images to a .NET MAUI app project

9/20/2022 • 2 minutes to read • [Edit Online](#)

Images are a crucial part of app navigation, usability, and branding. However, each platform has differing image requirements that typically involve creating multiple versions of each image at different resolutions. Therefore, a single image typically has to be duplicated multiple times per platform, at different resolutions, with the resulting images having to use different filename and folder conventions on each platform.

In a .NET Multi-platform App UI (.NET MAUI) app project, images can be specified in a single location in your app project, and at build time they can be automatically resized to the correct resolution for the target platform and device, and added to your app package. This avoids having to manually duplicate and name images on a per platform basis. By default, bitmap (non-vector) image formats, including animated GIFs, are not automatically resized by .NET MAUI.

.NET MAUI images can use any of the standard platform image formats, including Scalable Vector Graphics (SVG) files.

IMPORTANT

.NET MAUI converts SVG files to PNG files. Therefore, when adding an SVG file to your .NET MAUI app project, it should be referenced from XAML or C# with a .png extension. The only reference to the SVG file should be in your project file.

An image can be added to your app project by dragging it into the `Resources\Images` folder of the project, where its build action will automatically be set to `MauiImage`. This creates a corresponding entry in your project file:

```
<ItemGroup>
  <MauiImage Include="Resources\Images\logo.svg" />
</ItemGroup>
```

NOTE

Images can also be added to other folders of your app project. However, in this scenario their build action must be manually set to `MauiImage` in the **Properties** window.

To comply with Android resource naming rules, image filenames must be lowercase, start and end with a letter character, and contain only alphanumeric characters or underscores. For more information, see [App resources overview](#) on developer.android.com.

The base size of the image can be specified by setting the `BaseSize` attribute to values that are divisible by 8:

```
<MauiImage Include="Resources\Images\logo.jpg" BaseSize="376,678" />
```

The value of the `BaseSize` attribute represents the baseline density of the image, and is effectively the 1.0 scale factor for the image (the size you would typically use in your code to specify the image size) from which all other density sizes are derived. This value will be used to ensure that images are correctly resized to different display densities. If you don't specify a `BaseSize` for a bitmap image, the image isn't resized. If you don't specify a `BaseSize` value for a vector image, the dimensions specified in the SVG are assumed to be the base size. To stop vector images being resized, set the `Resize` attribute to `false`:

```
<MauiImage Include="Resources\Images\logo.svg" Resize="false" />
```

To add a tint to your images, which is useful when you have icons or simple images you'd like to render in a different color to the source, set the `TintColor` attribute:

```
<MauiImage Include="Resources\Images\logo.svg" TintColor="#66B3FF" />
```

A background color for an image can also be specified:

```
<MauiImage Include="Resources\Images\logo.svg" Color="#512BD4" />
```

Color values can be specified in hexadecimal, or as a .NET MAUI color. For example, `Color="Red"` is valid.

At build time, images can be resized to the correct resolutions for the target platform and device. The resulting images are then added to your app package.

Add a splash screen to a .NET MAUI app project

9/20/2022 • 3 minutes to read • [Edit Online](#)

On Android and iOS, .NET Multi-platform App UI (.NET MAUI) apps can display a splash screen while their initialization process completes. The splash screen is displayed immediately when an app is launched, providing immediate feedback to users while app resources are initialized:



Once the app is ready for interaction, its splash screen is dismissed.

In a .NET MAUI app project, a splash screen can be specified in a single location in your app project, and at build time it can be automatically resized to the correct resolution for the target platform and device, and added to your app package. This avoids having to manually duplicate and name the splash screen on a per platform basis. By default, bitmap (non-vector) image formats are not automatically resized by .NET MAUI.

A .NET MAUI splash screen can use any of the standard platform image formats, including Scalable Vector Graphics (SVG) files.

IMPORTANT

.NET MAUI converts SVG files to PNG files. Therefore, when adding an SVG file to your .NET MAUI app project, it should be referenced from XAML or C# with a .png extension. The only reference to the SVG file should be in your project file.

A splash screen can be added to your app project by dragging an image into the *Resources\Splash* folder of the project, where its build action will automatically be set to **MauiSplashScreen**. This creates a corresponding entry in your project file:

```
<ItemGroup>
  <MauiSplashScreen Include="Resources\Splash\splashscreen.svg" />
</ItemGroup>
```

NOTE

A splash screen can also be added to other folders of your app project. However, in this scenario its build action must be manually set to **MauiSplashScreen** in the **Properties** window.

To comply with Android resource naming rules, splash screen files names must be lowercase, start and end with a letter character, and contain only alphanumeric characters or underscores. For more information, see [App resources overview](#) on developer.android.com.

The base size of the splash screen can be specified by setting the `BaseSize` attribute to values that are divisible by 8:

```
<MauiSplashScreen Include="Resources\Splash\splashscreen.jpg" BaseSize="128,128" />
```

The value of the `BaseSize` attribute represents the baseline density of the splash screen, and is effectively the 1.0 scale factor for the splash screen from which all other density sizes are derived. This value will be used to ensure that splash screens are correctly resized to different display densities. If you don't specify a `BaseSize` for a bitmap-based splash screen, the image isn't resized. If you don't specify a `BaseSize` value for a vector-based splash screen, the dimensions specified in the SVG are assumed to be the base size. To stop vector images being resized, set the `Resize` attribute to `false`:

```
<MauiSplashScreen Include="Resources\Splash\splashscreen.svg" Resize="false" />
```

To add a tint to your splash screen, which is useful when you have a simple image you'd like to render in a different color to the source, set the `TintColor` attribute:

```
<MauiSplashScreen Include="Resources\Splash\splashscreen.svg" TintColor="#66B3FF" />
```

A background color for your splash screen can also be specified:

```
<MauiSplashScreen Include="Resources\Splash\splashscreen.svg" Color="#512BD4" />
```

Color values can be specified in hexadecimal, or as a .NET MAUI color. For example, `Color="Red"` is valid.

At build time, the splash screen can be resized to the correct resolution for the target platform and device. The resulting splash screen is then added to your app package.

- [Android](#)
- [iOS](#)

On Android, the splash screen is added to your app package as `Resources/values/maui_colors.xml` and `Resources/drawable/maui_splash_image.xml`. .NET MAUI apps use the `Maui.SplashTheme` by default, which ensures that a splash screen will be displayed if present. Therefore, you should not specify a different theme in your manifest file or in your `MainActivity` class:

```
using Android.App;
using Android.Content.PM;

namespace MyMauiApp
{
    [Activity(Theme = "@style/Maui.SplashTheme", MainLauncher = true, ConfigurationChanges =
ConfigChanges.ScreenSize | ConfigChanges.Orientation | ConfigChanges.UiMode | ConfigChanges.ScreenLayout | 
ConfigChanges.SmallestScreenSize)]
    public class MainActivity : MauiAppCompatActivity
    {
    }
}
```

For more advanced splash screen scenarios, per-platform approaches apply.

Shadow

9/20/2022 • 2 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) `Shadow` class paints a shadow around a layout or view. The `VisualElement` class has a `Shadow` bindable property, of type `Shadow`, that enables a shadow to be added to any layout or view.

The `Shadow` class defines the following properties:

- `Radius`, of type `float`, defines the radius of the blur used to generate the shadow. The default value of this property is 10.
- `Opacity`, of type `float`, indicates the opacity of the shadow. The default value of this property is 1.
- `Brush`, of type `Brush`, represents the brush used to colorize the shadow.
- `offset`, of type `Point`, specifies the offset for the shadow, which represents the position of the light source that creates the shadow.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

IMPORTANT

The `Brush` property only currently supports a `SolidColorBrush`.

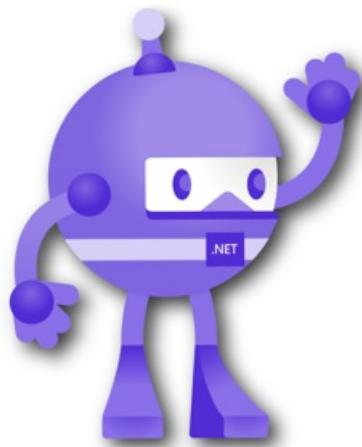
Create a Shadow

To add a shadow to a control, set the control's `Shadow` property to a `Shadow` object whose properties define its appearance.

The following XAML example shows how to add a shadow to an `Image`:

```
<Image Source="dotnet_bot.png"
       WidthRequest="250"
       HeightRequest="310">
    <Image.Shadow>
        <Shadow Brush="Black"
               Offset="20,20"
               Radius="40"
               Opacity="0.8" />
    </Image.Shadow>
</Image>
```

In this example, a black shadow is painted around the outline of the image, with its offset specifying that it appears at the right and bottom of the image:



Shadows can also be added to clipped objects, as shown in the following example:

```
<Image Source="https://aka.ms/campus.jpg"
    Aspect="AspectFill"
    HeightRequest="220"
    WidthRequest="220"
    HorizontalOptions="Center">
    <Image.Clip>
        <EllipseGeometry Center="220,250"
            RadiusX="220"
            RadiusY="220" />
    </Image.Clip>
    <Image.Shadow>
        <Shadow Brush="Black"
            Offset="10,10"
            Opacity="0.8" />
    </Image.Shadow>
</Image>
```

In this example, a black shadow is painted around the outline of the `EllipseGeometry` that clips the image:



For more information about clipping an element, see [Clip with a Geometry](#).

Style apps using XAML

9/20/2022 • 13 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) apps often contain multiple controls that have an identical appearance. For example, an app may have multiple `Label` instances that have the same font options and layout options:

```
<Label Text="These labels"
       HorizontalOptions="Center"
       VerticalOptions="Center"
       FontSize="18" />
<Label Text="are not"
       HorizontalOptions="Center"
       VerticalOptions="Center"
       FontSize="18" />
<Label Text="using styles"
       HorizontalOptions="Center"
       VerticalOptions="Center"
       FontSize="18" />
```

In this example, each `Label` object has identical property values for controlling the appearance of the text displayed by the `Label`. However, 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.

Introduction to styles

An app can be styled by using the `Style` class to group a collection of property values into one object that can then be applied to multiple visual elements. This helps to reduce repetitive markup, and allows an app's appearance to be more easily changed.

Although styles are designed primarily for XAML-based apps, they can also be created in C#:

- `Style` objects 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 app.
- `Style` objects 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 app-level can be applied throughout the app.

Each `Style` object 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` object can be *explicit*, or *implicit*.

- An *explicit* `style` object 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, see [Explicit styles](#).
- An *implicit* `style` object is defined by specifying only a `TargetType`. The `Style` object will then automatically be applied to all elements of that type. However, the subclasses of the `TargetType` do not

automatically have the `Style` applied. For more information, see [Implicit styles](#).

When creating a `Style`, the `TargetType` property is always required. The following example shows an *explicit* style:

```
<Style x:Key="labelStyle" TargetType="Label">
    <Setter Property="HorizontalOptions" Value="Center" />
    <Setter Property="VerticalOptions" Value="Center" />
    <Setter Property="FontSize" Value="18" />
</Style>
```

To apply a `Style`, the target object must be a `VisualElement` that matches the `TargetType` property value of the `Style`:

```
<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 app-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.

Styles do not respond to property changes, and remain unchanged for the duration of an app. However, apps can respond to style changes dynamically at runtime by using dynamic resources. For more information, see [Dynamic styles](#).

Explicit styles

To create 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 example shows *explicit* styles in a page's `ResourceDictionary`, and applied to the page's `Label` objects:

```

<ContentPage ...>
    <ContentPage.Resources>
        <Style x:Key="labelRedStyle"
            TargetType="Label">
            <Setter Property="HorizontalOptions" Value="Center" />
            <Setter Property="VerticalOptions" Value="Center" />
            <Setter Property="FontSize" Value="18" />
            <Setter Property="TextColor" Value="Red" />
        </Style>
        <Style x:Key="labelGreenStyle"
            TargetType="Label">
            <Setter Property="HorizontalOptions" Value="Center" />
            <Setter Property="VerticalOptions" Value="Center" />
            <Setter Property="FontSize" Value="18" />
            <Setter Property="TextColor" Value="Green" />
        </Style>
        <Style x:Key="labelBlueStyle"
            TargetType="Label">
            <Setter Property="HorizontalOptions" Value="Center" />
            <Setter Property="VerticalOptions" Value="Center" />
            <Setter Property="FontSize" Value="18" />
            <Setter Property="TextColor" Value="Blue" />
        </Style>
    </ContentPage.Resources>
    <StackLayout>
        <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>

```

In this example, the `ResourceDictionary` defines three styles that are explicitly set on the page's `Label` objects. 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. In addition, while the final `Label` has a `Style` set on it, it also overrides the `TextColor` property to a different `color` value.

Implicit styles

To create 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 in scope 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 in a page's `ResourceDictionary`, and applied to the page's `Entry` objects:

```

<ContentPage ...>
    <ContentPage.Resources>
        <Style TargetType="Entry">
            <Setter Property="HorizontalOptions" Value="Fill" />
            <Setter Property="VerticalOptions" Value="Center" />
            <Setter Property="BackgroundColor" Value="Yellow" />
            <Setter Property="FontAttributes" Value="Italic" />
            <Setter Property="TextColor" Value="Blue" />
        </Style>
    </ContentPage.Resources>
    <StackLayout>
        <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>

```

In this example, the `ResourceDictionary` defines a single *implicit* style that are implicitly set on the page's `Entry` objects. 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` objects implicitly as they match the `TargetType` property of the `Style` exactly. However, the `Style` is not applied to the `CustomEntry` object, which is a subclassed `Entry`. In addition, the fourth `Entry` overrides the `BackgroundColor` and `TextColor` properties of the style to different `Color` values.

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` objects 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` objects.

Global styles

Styles can be defined globally by adding them to the app's resource dictionary. These styles can then be consumed throughout an app, and help to avoid style duplication across pages and controls.

The following example shows a `Style` defined at the app-level:

```

<Application xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:Styles"
    x:Class="Styles.App">
    <Application.Resources>
        <Style x:Key="buttonStyle" TargetType="Button">
            <Setter Property="HorizontalOptions"
                Value="Center" />
            <Setter Property="VerticalOptions"
                Value="CenterAndExpand" />
            <Setter Property="BorderColor"
                Value="Lime" />
            <Setter Property="CornerRadius"
                Value="5" />
            <Setter Property="BorderWidth"
                Value="5" />
            <Setter Property="WidthRequest"
                Value="200" />
            <Setter Property="TextColor"
                Value="Teal" />
        </Style>
    </Application.Resources>
</Application>

```

In this example, the `ResourceDictionary` defines a single *explicit* style, `buttonStyle`, which will be used to set the appearance of `Button` objects.

NOTE

Global styles can be *explicit* or *implicit*.

The following example shows a page consuming the `buttonStyle` on the page's `Button` objects:

```

<ContentPage ...>
    <StackLayout>
        <Button Text="These buttons"
            Style="{StaticResource buttonStyle}" />
        <Button Text="are demonstrating"
            Style="{StaticResource buttonStyle}" />
        <Button Text="application styles"
            Style="{StaticResource buttonStyle}" />
    </StackLayout>
</ContentPage>

```

Style inheritance

Styles can inherit from other styles to reduce duplication and enable reuse. This is achieved by setting the `Style.BasedOn` property to an existing `Style`. In XAML, this can be achieved by setting the `BasedOn` property to a `StaticResource` markup extension that references a previously created `Style`.

Styles that inherit from a base style can include `Setter` instances for new properties, or use them to override setters 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` objects, styles that are based on the base style can target `View` objects or types that derive from the `View` class, such as `Label` and `Button` objects.

A style can only inherit from styles at the same level, or above, in the view hierarchy. This means that:

- An app-level style can only inherit from other app-level styles.
- A page-level style can inherit from app-level styles, and other page-level styles.
- A control-level style can inherit from app-level styles, page-level styles, and other control-level styles.

The following example shows *explicit* style inheritance:

```
<ContentPage ...>
    <ContentPage.Resources>
        <Style x:Key="baseStyle"
            TargetType="View">
            <Setter Property="HorizontalOptions" Value="Center" />
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>
    </ContentPage.Resources>
    <StackLayout>
        <StackLayout.Resources>
            <Style x:Key="labelStyle"
                TargetType="Label"
                BasedOn="{StaticResource baseStyle}">
                <Setter Property="FontSize" Value="18" />
                <Setter Property="FontAttributes" Value="Italic" />
                <Setter Property="TextColor" Value="Teal" />
            </Style>
            <Style x:Key="buttonStyle"
                TargetType="Button"
                BasedOn="{StaticResource baseStyle}">
                <Setter Property="BorderColor" Value="Lime" />
                <Setter Property="CornerRadius" Value="5" />
                <Setter Property="BorderWidth" Value="5" />
                <Setter Property="WidthRequest" Value="200" />
                <Setter Property="TextColor" Value="Teal" />
            </Style>
        </StackLayout.Resources>
        <Label Text="This label uses style inheritance"
            Style="{StaticResource labelStyle}" />
        <Button Text="This button uses style inheritance"
            Style="{StaticResource buttonStyle}" />
    </StackLayout>
</ContentPage>
```

In this example, the `baseStyle` targets `View` objects, 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` objects are then set on a `Label` and `Button`.

IMPORTANT

An implicit style can be derived from an explicit style, but an explicit style can't be derived from an implicit style.

Dynamic styles

Styles do not respond to property changes, and remain unchanged for the duration of an app. For example, after assigning a `Style` to a visual element, if one of the `Setter` objects is modified, removed, or a new `Setter` added, the changes won't be applied to the visual element. However, apps 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 app.

The following example shows *dynamic* styles:

```
<ContentPage ...>
    <ContentPage.Resources>
        <Style x:Key="baseStyle"
            TargetType="View"
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>
        <Style x:Key="blueSearchBarStyle"
            TargetType="SearchBar"
            BasedOn="{StaticResource baseStyle}"
            <Setter Property="FontAttributes" Value="Italic" />
            <Setter Property="PlaceholderColor" Value="Blue" />
        </Style>
        <Style x:Key="greenSearchBarStyle"
            TargetType="SearchBar"
            <Setter Property="FontAttributes" Value="None" />
            <Setter Property="PlaceholderColor" Value="Green" />
        </Style>
    </ContentPage.Resources>
    <StackLayout>
        <SearchBar Placeholder="SearchBar demonstrating dynamic styles"
            Style="{DynamicResource blueSearchBarStyle}" />
    </StackLayout>
</ContentPage>
```

In this example, the `SearchBar` object uses the `DynamicResource` markup extension to set a `Style` named `blueSearchBarStyle`. The `SearchBar` can then have its `Style` definition updated in code:

```
Resources["blueSearchBarStyle"] = Resources["greenSearchBarStyle"];
```

In this example, the `blueSearchBarStyle` definition is updated to use the values from the `greenSearchBarStyle` definition. When this code is executed, the `SearchBar` will be updated to use the `Setter` objects defined in `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 example shows *dynamic* style inheritance:

```

<ContentPage ...>
    <ContentPage.Resources>
        <Style x:Key="baseStyle"
            TargetType="View">
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>
        <Style x:Key="blueSearchBarStyle"
            TargetType="SearchBar"
            BasedOn="{StaticResource baseStyle}">
            <Setter Property="FontAttributes" Value="Italic" />
            <Setter Property="TextColor" Value="Blue" />
        </Style>
        <Style x:Key="greenSearchBarStyle"
            TargetType="SearchBar">
            <Setter Property="FontAttributes" Value="None" />
            <Setter Property="TextColor" Value="Green" />
        </Style>
        <Style x:Key="tealSearchBarStyle"
            TargetType="SearchBar"
            BaseResourceKey="blueSearchBarStyle">
            <Setter Property="BackgroundColor" Value="Teal" />
            <Setter Property="CancelButtonColor" Value="White" />
        </Style>
    </ContentPage.Resources>
    <StackLayout>
        <SearchBar Text="SearchBar demonstrating dynamic style inheritance"
            Style="{StaticResource tealSearchBarStyle}" />
    </StackLayout>
</ContentPage>

```

In this example, the `SearchBar` object uses the `StaticResource` markup extension to reference a `Style` named `tealSearchBarStyle`. This `Style` sets some additional properties and uses the `BaseResourceKey` property to reference `blueSearchBarStyle`. 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 `blueSearchBarStyle` and is updated when the base style changes.

The `blueSearchBarStyle` definition can be updated in code:

```
Resources["blueSearchBarStyle"] = Resources["greenSearchBarStyle"];
```

In this example, the `blueSearchBarStyle` definition is updated to use the values from the `greenSearchBarStyle` definition. When this code is executed, the `SearchBar` will be updated to use the `Setter` objects defined in `greenSearchBarStyle`.

Style classes

Style classes enable multiple styles to be applied to a control, without resorting to style inheritance.

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>

```

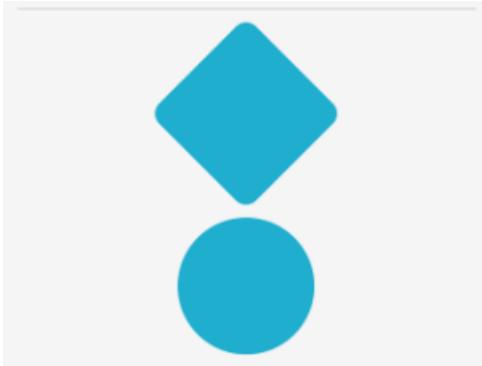
In this example, the `Separator`, `Rounded`, and `Circle` style classes each set `BoxView` properties to specific values. The `Rotated` style class has a `TargetType` of `VisualElement`, which means it can only be applied to `VisualElement` instances. However, its `ApplyToDerivedTypes` property is set 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](#).

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>
        <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.

Style apps using Cascading Style Sheets

9/20/2022 • 13 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) apps can be styled 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. 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 example shows some .NET MAUI compliant CSS:

```
navigationpage {  
    -maui-bar-background-color: lightgray;  
}  
  
^contentpage {  
    background-color: lightgray;  
}  
  
#listView {  
    background-color: lightgray;  
}  
  
stacklayout {  
    margin: 20;  
    -maui-spacing: 6;  
}  
  
grid {  
    row-gap: 6;  
    column-gap: 6;  
}  
.mainPageTitle {  
    font-style: bold;  
    font-size: 14;  
}  
  
.mainPageSubtitle {  
    margin-top: 15;  
}  
  
.detailPageTitle {  
    font-style: bold;  
    font-size: 14;  
    text-align: center;  
}  
  
.detailPageSubtitle {  
    text-align: center;  
    font-style: italic;  
}  
  
listview image {  
    height: 60;  
    width: 60;  
}  
  
stacklayout>image {  
    height: 200;  
    width: 200;  
}
```

In .NET MAUI, CSS style sheets are parsed and evaluated at runtime, rather than compile time, and style sheets are re-parsed on use.

IMPORTANT

It's not possible to fully style a .NET MAUI app using CSS. However, XAML styles can be used to supplement CSS. For more information about XAML styles, see [Style apps using XAML](#).

Consume a style sheet

The process for adding a style sheet to a .NET MAUI app is as follows:

1. Add an empty CSS file to your .NET MAUI app project. The CSS file can be placed in any folder, with the *Resources* folder being the recommended location.
2. Set the build action of the CSS file to **MauiCss**.

Loading a style sheet

There are a number of approaches that can be used to load a style sheet.

NOTE

It's not possible to change a style sheet at runtime and have the new style sheet applied.

Load a style sheet in XAML

A style sheet can be loaded and parsed with the `StyleSheet` class before being added to a `ResourceDictionary`:

```
<Application ...>
  <Application.Resources>
    <StyleSheet Source="/Resources/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 **MauiCss**.

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](#).

Load a style sheet in C#

In C#, a style sheet can be loaded from a `StringReader` and added to a `ResourceDictionary`:

```
using Microsoft.Maui.Controls.StyleSheets;

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.

Select elements and apply 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 app 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 app resources will be overwritten by a style defined in the control resources, if they set the same properties.

NOTE

CSS variables are unsupported.

Select 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 .NET MAUI, 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 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="/Resources/styles.css" />  
    </ContentPage.Resources>  
    <StackLayout>  
        <ListView x:Name="listView">  
            ...  
        </ListView>  
    </StackLayout>  
</ContentPage>
```

Select 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: 14;
    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 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="/Resources/styles.css" />
    </ContentPage.Resources>
    <ScrollView>
        <StackLayout>
            <Label ... StyleClass="detailPageTitle" />
            <Label ... StyleClass="detailPageSubtitle"/>
        </StackLayout>
    </ScrollView>
</ContentPage>

```

Select 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="/Resources/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.

Select 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 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="/Resources/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 .NET MAUI:

SELECTOR	EXAMPLE	DESCRIPTION
<code>.class</code>	<code>.header</code>	Selects all elements with the <code>StyleClass</code> property containing 'header'. This selector is case sensitive.
<code>#id</code>	<code>#email</code>	Selects all elements with <code>StyleId</code> set to <code>email</code> . If <code>StyleId</code> is not set, fallback to <code>x:Name</code> . When using XAML, <code>x:Name</code> is preferred over <code>StyleId</code> . This selector is case sensitive.
<code>*</code>	<code>*</code>	Selects all elements.

SELECTOR	EXAMPLE	DESCRIPTION
<code>element</code>	<code>label</code>	Selects all elements of type <code>Label</code> , but not subclasses. This selector is case insensitive.
<code>^base</code>	<code>^contentpage</code>	Selects all elements with <code>ContentPage</code> as the base class, including <code>ContentPage</code> itself. This selector is case insensitive and isn't part of the CSS specification.
<code>element,element</code>	<code>label,button</code>	Selects all <code>Button</code> elements and all <code>Label</code> elements. This selector is case insensitive.
<code>element element</code>	<code>stacklayout label</code>	Selects all <code>Label</code> elements inside a <code>StackLayout</code> . This selector is case insensitive.
<code>element>element</code>	<code>stacklayout>label</code>	Selects all <code>Label</code> elements with <code>StackLayout</code> as a direct parent. This selector is case insensitive.
<code>element+element</code>	<code>label+entry</code>	Selects all <code>Entry</code> elements directly after a <code>Label</code> . This selector is case insensitive.
<code>element~element</code>	<code>label~entry</code>	Selects all <code>Entry</code> elements preceded by a <code>Label</code> . 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 unsupported:

- `[attribute]`
- `@media` and `@supports`
- `:` and `::`

NOTE

Specificity, and specificity overrides are unsupported.

Property reference

The following CSS properties are supported by .NET MAUI (in the `Values` column, types are *italic*, while string literals are `gray`):

PROPERTY	APPLIES TO	VALUES	EXAMPLE
	FlexLayout	stretch center start end spacebetween spacearound spaceevenly flex-start flex-end space-between space-around initial	align-content: space-between;
	FlexLayout	stretch center start end flex-start flex-end initial	align-items: flex-start;
	VisualElement	auto stretch center start end flex-start flex-end initial	align-self: flex-end;
background-color="background-color">	VisualElement	color initial	background-color: springgreen;
background-image="background-image">	Page	string initial	background-image: bg.png;
border-color="border-color">	Button , Frame , ImageButton	color initial	border-color: #9acd32;
border-radius="border-radius">	BoxView , Button , Frame , ImageButton	double initial	border-radius: 10;
border-width="border-width">	Button , ImageButton	double initial	border-width: .5;
color="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="column-gap">	Grid	double initial	column-gap: 9;
direction="direction">	VisualElement	ltr rtl inherit initial	direction: rtl;
flex-direction="flex-direction">	FlexLayout	column columnreverse row rowreverse row-reverse column-reverse initial	flex-direction: column-reverse;

PROPERTY	APPLIES TO	VALUES	EXAMPLE
<code>flex-basis</code>	<code>VisualElement</code>	<code>float</code> <code>auto</code> <code>initial</code> . In addition, a percentage in the range 0% to 100% can be specified with the <code>%</code> sign.	<code>flex-basis: 25%;</code>
<code>flex-grow</code>	<code>VisualElement</code>	<code>float</code> <code>initial</code>	<code>flex-grow: 1.5;</code>
<code>flex-shrink</code>	<code>VisualElement</code>	<code>float</code> <code>initial</code>	<code>flex-shrink: 1;</code>
<code>flex-wrap</code>	<code>VisualElement</code>	<code>nowrap</code> <code>wrap</code> <code>reverse</code> <code>wrap-reverse</code> <code>initial</code>	<code>flex-wrap: wrap-reverse;</code>
<code>font-family</code>	<code>Button</code> , <code>DatePicker</code> , <code>Editor</code> , <code>Entry</code> , <code>Label</code> , <code>Picker</code> , <code>SearchBar</code> , <code>TimePicker</code> , <code>Span</code>	<code>string</code> <code>initial</code>	<code>font-family: Consolas;</code>
<code>font-size</code>	<code>Button</code> , <code>DatePicker</code> , <code>Editor</code> , <code>Entry</code> , <code>Label</code> , <code>Picker</code> , <code>SearchBar</code> , <code>TimePicker</code> , <code>Span</code>	<code>double</code> <code>initial</code>	<code>font-size: 12;</code>
<code>font-style</code>	<code>Button</code> , <code>DatePicker</code> , <code>Editor</code> , <code>Entry</code> , <code>Label</code> , <code>Picker</code> , <code>SearchBar</code> , <code>TimePicker</code> , <code>Span</code>	<code>bold</code> <code>italic</code> <code>initial</code>	<code>font-style: bold;</code>
<code>height</code>	<code>VisualElement</code>	<code>double</code> <code>initial</code>	<code>min-height: 250;</code>
<code>justify-content</code>	<code>FlexLayout</code>	<code>start</code> <code>center</code> <code>end</code> <code>spacebetween</code> <code>spacearound</code> <code>spaceevenly</code> <code>flex-start</code> <code>flex-end</code> <code>space-between</code> <code>space-around</code> <code>initial</code>	<code>justify-content: flex-end;</code>
<code>letter-spacing</code>	<code>Button</code> , <code>DatePicker</code> , <code>Editor</code> , <code>Entry</code> , <code>Label</code> , <code>Picker</code> , <code>SearchBar</code> , <code>SearchHandler</code> , <code>Span</code> , <code>TimePicker</code>	<code>double</code> <code>initial</code>	<code>letter-spacing: 2.5;</code>
<code>line-height</code>	<code>Label</code> , <code>Span</code>	<code>double</code> <code>initial</code>	<code>line-height: 1.8;</code>
<code>margin</code>	<code>View</code>	<code>thickness</code> <code>initial</code>	<code>margin: 6 12;</code>
<code>margin-left</code>	<code>View</code>	<code>thickness</code> <code>initial</code>	<code>margin-left: 3;</code>
<code>margin-top</code>	<code>View</code>	<code>thickness</code> <code>initial</code>	<code>margin-top: 2;</code>

PROPERTY	APPLIES TO	VALUES	EXAMPLE
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;
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;

PROPERTY	APPLIES TO	VALUES	EXAMPLE
<code>transform</code>	<code>VisualElement</code>	<code>none</code> , <code>rotate</code> , <code>rotateX</code> , <code>rotateY</code> , <code>scale</code> , <code>scaleX</code> , <code>scaleY</code> , <code>translate</code> , <code>translateX</code> , <code>translateY</code> , <code>initial</code>	<code>transform: rotate(180), scaleX(2.5);</code>
<code>transform-origin</code>	<code>VisualElement</code>	<code>double</code> , <code>double</code> <code>initial</code>	<code>transform-origin: 7.5, 12.5;</code>
<code>vertical-align</code>	<code>Label</code>	<code>left</code> <code>top</code> <code>right</code> <code>bottom</code> <code>start</code> <code>center</code> <code>middle</code> <code>end</code> <code>initial</code>	<code>vertical-align: bottom;</code>
<code>visibility</code>	<code>VisualElement</code>	<code>true</code> <code>visible</code> <code>false</code> <code>hidden</code> <code>collapse</code> <code>initial</code>	<code>visibility: hidden;</code>
<code>width</code>	<code>VisualElement</code>	<code>double</code> <code>initial</code>	<code>min-width: 320;</code>

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 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`.

IMPORTANT

`Span` elements can't be targeted using CSS.

.NET MAUI specific properties

The following .NET MAUI specific CSS properties are also supported (in the **Values** column, types are *italic*, while string literals are `gray`):

PROPERTY	APPLIES TO	VALUES	EXAMPLE
<code>-maui-bar-background-color</code>	<code>NavigationPage</code> , <code>TabbedPage</code>	<code>color</code> <code>initial</code>	<code>-xf-bar-background-color: teal;</code>
<code>-maui-bar-text-color</code>	<code>NavigationPage</code> , <code>TabbedPage</code>	<code>color</code> <code>initial</code>	<code>-xf-bar-text-color: gray</code>

PROPERTY	APPLIES TO	VALUES	EXAMPLE
-maui-horizontal-scroll-bar-visibility	ScrollView	default always never initial	-xf-horizontal-scrollbar-visibility: never;
-maui-max-length	Entry , Editor , SearchBar	int initial	-xf-max-length: 20;
-maui-max-track-color	Slider	color initial	-xf-max-track-color: red;
-maui-min-track-color	Slider	color initial	-xf-min-track-color: yellow;
-maui-orientation	ScrollView , StackLayout	horizontal vertical both initial . both is only supported on a ScrollView .	-xf-orientation: horizontal;
-maui-placeholder	Entry , Editor , SearchBar	<i>quoted text</i> initial	-xf-placeholder: Enter name;
-maui-placeholder-color	Entry , Editor , SearchBar	color initial	-xf-placeholder-color: green;
-maui-spacing	StackLayout	double initial	-xf-spacing: 8;
-maui-thumb-color	Slider , Switch	color initial	-xf-thumb-color: limegreen;
-maui-vertical-scroll-bar-visibility	ScrollView	default always never initial	-xf-vertical-scrollbar-visibility: always;
-maui-vertical-text-alignment	Label	start center end initial	-xf-vertical-text-alignment: end;
-maui-visual	VisualElement	string initial	-xf-visual: material;

.NET MAUI Shell specific properties

The following .NET MAUI Shell specific CSS properties are also supported (in the **Values** column, types are *italic*, while string literals are `gray`):

PROPERTY	APPLIES TO	VALUES	EXAMPLE
-maui-flyout-background	Shell	color initial	-xf-flyout-background: red;
-maui-shell-background	Element	color initial	-xf-shell-background: green;
-maui-shell-disabled	Element	color initial	-xf-shell-disabled: blue;

PROPERTY	APPLIES TO	VALUES	EXAMPLE
-maui-shell-foreground	Element	color initial	-xf-shell-foreground: yellow;
-maui-shell-tabbar-background	Element	color initial	-xf-shell-tabbar-background: white;
-maui-shell-tabbar-disabled	Element	color initial	-xf-shell-tabbar-disabled: black;
-maui-shell-tabbar-foreground	Element	color initial	-xf-shell-tabbar-foreground: gray;
-maui-shell-tabbar-title	Element	color initial	-xf-shell-tabbar-title: lightgray;
-maui-shell-tabbar-unselected	Element	color initial	-xf-shell-tabbar-unselected: cyan;
-maui-shell-title	Element	color initial	-xf-shell-title: teal;
-maui-shell-unselected	Element	color initial	-xf-shell-unselected: limegreen;

Color

The following `color` values are supported:

- `x11 colors`, which match CSS colors and .NET MAUI colors. These color values are case insensitive.
- hex colors: `#rgb`, `#argb`, `#rrggbb`, `#aarrggbb`
- rgb colors: `rgb(255,0,0)`, `rgb(100%,0%,0%)`. Values are in the range 0-255, or 0%-100%.
- rgba colors: `rgba(255, 0, 0, 0.8)`, `rgba(100%, 0%, 0%, 0.8)`. The opacity value is in the range 0.0-1.0.
- hsl colors: `hsl(120, 100%, 50%)`. The h value is in the range 0-360, while s and l are in the range 0%-100%.
- hsla colors: `hsla(120, 100%, 50%, .8)`. The opacity value is in the range 0.0-1.0.

Thickness

One, two, three, or four `thickness` values are supported, each separated by white space:

- A single value indicates uniform thickness.
- Two values indicate vertical then horizontal thickness.
- Three values indicate top, then horizontal (left and right), then bottom thickness.
- Four values indicate top, then right, then bottom, then left thickness.

NOTE

CSS `thickness` values differ from XAML `Thickness` values. For example, in XAML a two-value `Thickness` indicates horizontal then vertical thickness, while a four-value `Thickness` indicates left, then top, then right, then bottom thickness. In addition, XAML `Thickness` values are comma delimited.

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.

Theme an app

9/20/2022 • 4 minutes to read • [Edit Online](#)

 [Browse the sample](#)

.NET Multi-platform App UI (.NET MAUI) apps 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 .NET MAUI apps.

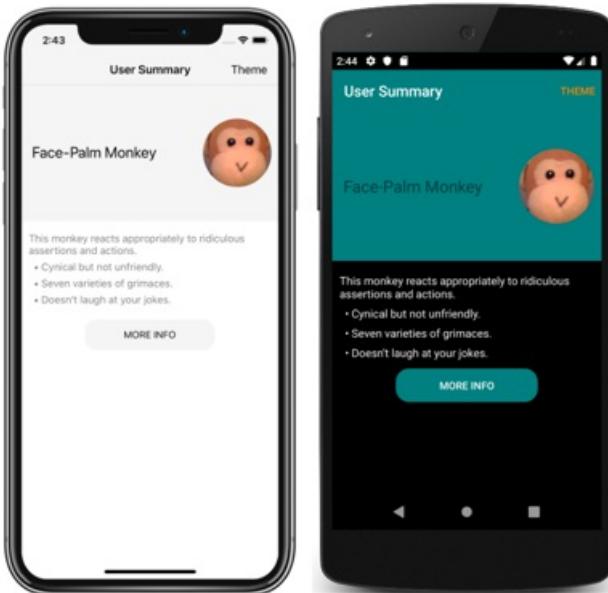
The process for implementing runtime theming in a .NET MAUI app is as follows:

1. Define the resources for each theme in a `ResourceDictionary`. For more information, see [Define themes](#).
2. Set a default theme in the app's `App.xaml`/file. For more information, see [Set a default theme](#).
3. Consume theme resources in the app, using the `DynamicResource` markup extension. For more information, see [Consume theme resources](#).
4. Add code to load a theme at runtime. For more information, see [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 screenshot shows themed pages, with the iOS app using a light theme and the Android app using a dark theme:



NOTE

Changing a theme at runtime requires the use of XAML styles, and is not possible using CSS.

.NET MAUI also has the ability to respond to 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. For more information, see [Respond to system theme changes](#).

Define themes

A theme is defined as a collection of resource objects stored in a `ResourceDictionary`.

The following example shows a `ResourceDictionary` for a light theme named `LightTheme`:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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 a `ResourceDictionary` for a dark theme named `DarkTheme`:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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 Dictionaries](#).

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 app 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 app-level `ResourceDictionary` that's defined in `App.xaml`.

```
<Application xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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 app 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 that can be applied to all `Label` objects in app:

```
<Application xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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 app-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://schemas.microsoft.com/dotnet/2021/maui"
    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 [Style apps using XAML](#). For more information about the `DynamicResource` markup extension, see [Dynamic styles](#).

Load a theme at runtime

When a theme is selected at runtime, an app should:

1. Remove the current theme from the app. This is achieved by clearing the `MergedDictionaries` property of the app-level `ResourceDictionary`.
2. Load the selected theme. This is achieved by adding an instance of the selected theme to the `MergedDictionaries` property of the app-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:

The following example shows removing the current theme and loading a new theme:

```
ICollection<ResourceDictionary> mergedDictionaries = Application.Current.Resources.MergedDictionaries;
if (mergedDictionaries != null)
{
    mergedDictionaries.Clear();
    mergedDictionaries.Add(new DarkTheme());
}
```

Respond to system theme changes

9/20/2022 • 4 minutes to read • [Edit Online](#)

 [Browse 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. Apps should respect these system themes, and respond immediately when the system theme changes.

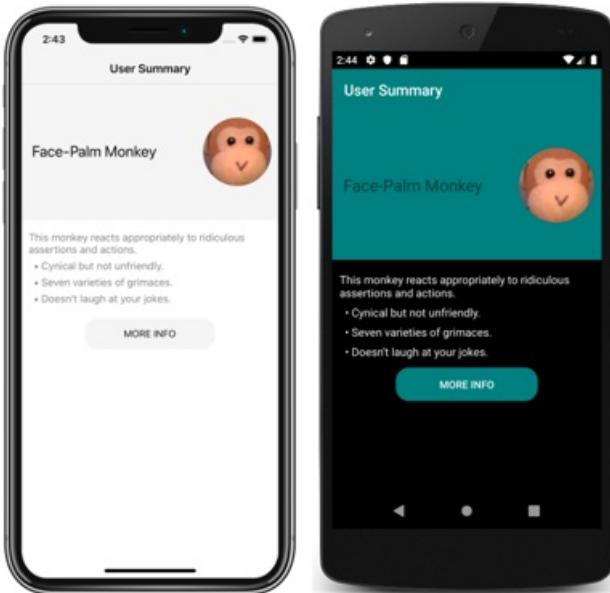
The system theme may change for a variety of reasons, depending on the device configuration. This includes the system theme being explicitly changed by the user, it changing due to the time of day, and it changing due to environmental factors such as low light.

.NET Multi-platform App UI (.NET MAUI) apps can respond to system theme changes by consuming resources with the `AppThemeBinding` markup extension, and the `SetAppThemeColor` and `SetAppTheme<T>` extension methods.

NOTE

.NET MAUI apps can respond to system theme changes on iOS 13 or greater, Android 10 (API 29) or greater, macOS 10.14 or greater, and Windows 10 or greater.

The following screenshot shows themed pages, for the light system theme on iOS and the dark system theme on Android:



Define and consume theme resources

Resources for light and dark themes can be consumed with the `AppThemeBinding` markup extension, and the `SetAppThemeColor` and `SetAppTheme<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:

```

<StackLayout>
    <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>

```

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

.NET MAUI includes `SetAppThemeColor` and `SetAppTheme<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, Colors.Green, Colors.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 `SetAppTheme<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.SetAppTheme<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:

```
AppTheme currentTheme = Application.Current.RequestedTheme;
```

The `RequestedTheme` property returns an `AppTheme` enumeration member. The `AppTheme` 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 app can be set with the `Application.UserAppTheme` property, which is of type `AppTheme`, regardless of which system theme is currently operational:

```
Application.Current.UserAppTheme = AppTheme.Dark;
```

In this example, the app 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 `AppTheme.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. .NET MAUI 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 `AppTheme`. This property can be examined to detect the requested system theme.

IMPORTANT

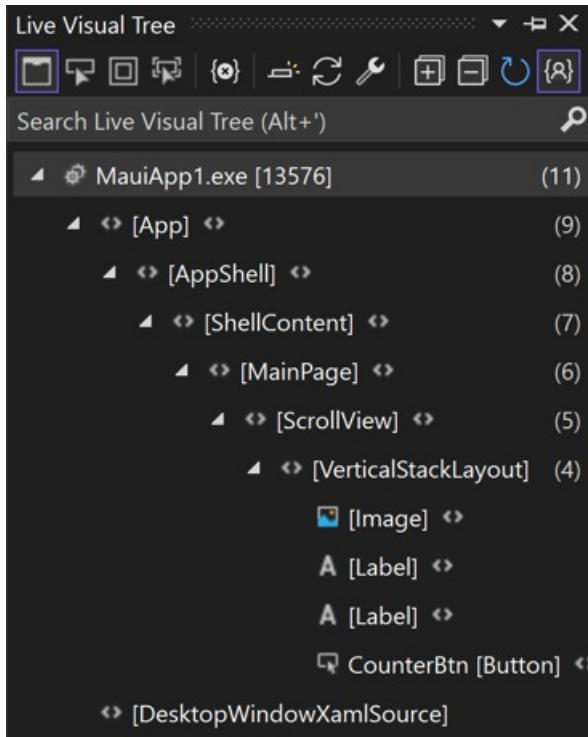
To respond to theme changes on Android your `MainActivity` class must include the `ConfigChanges.UiMode` flag in the `Activity` attribute. .NET MAUI apps created with the Visual Studio project templates automatically include this flag.

Inspect the visual tree of a .NET MAUI app

9/20/2022 • 2 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) Live Visual Tree is a Visual Studio feature that provides a tree view of the UI elements in your running .NET MAUI app.

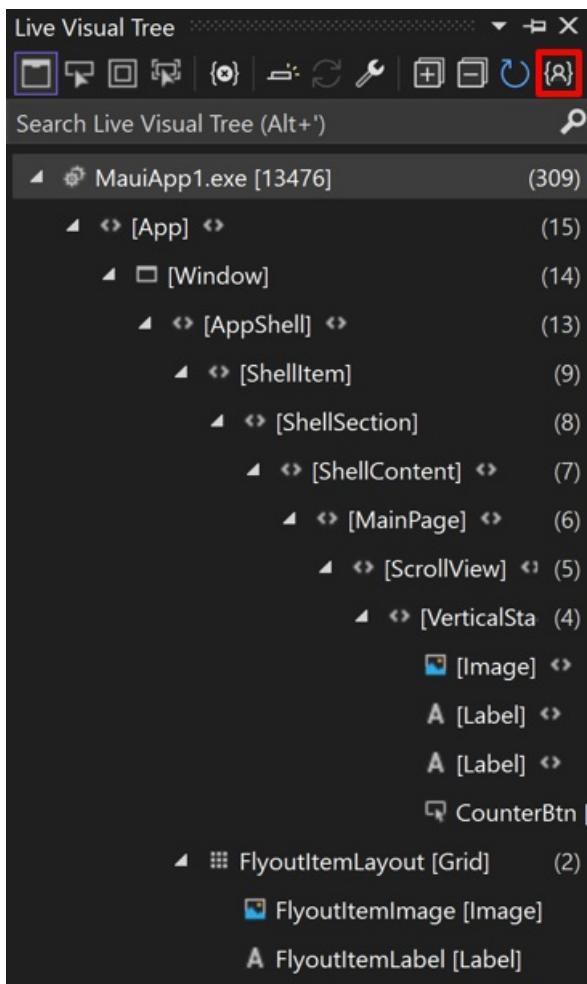
When your .NET MAUI app is running in debug configuration, with the debugger attached, the Live Visual Tree window can be opened by selecting **Debug > Windows > Live Visual Tree** from the Visual Studio menu bar:



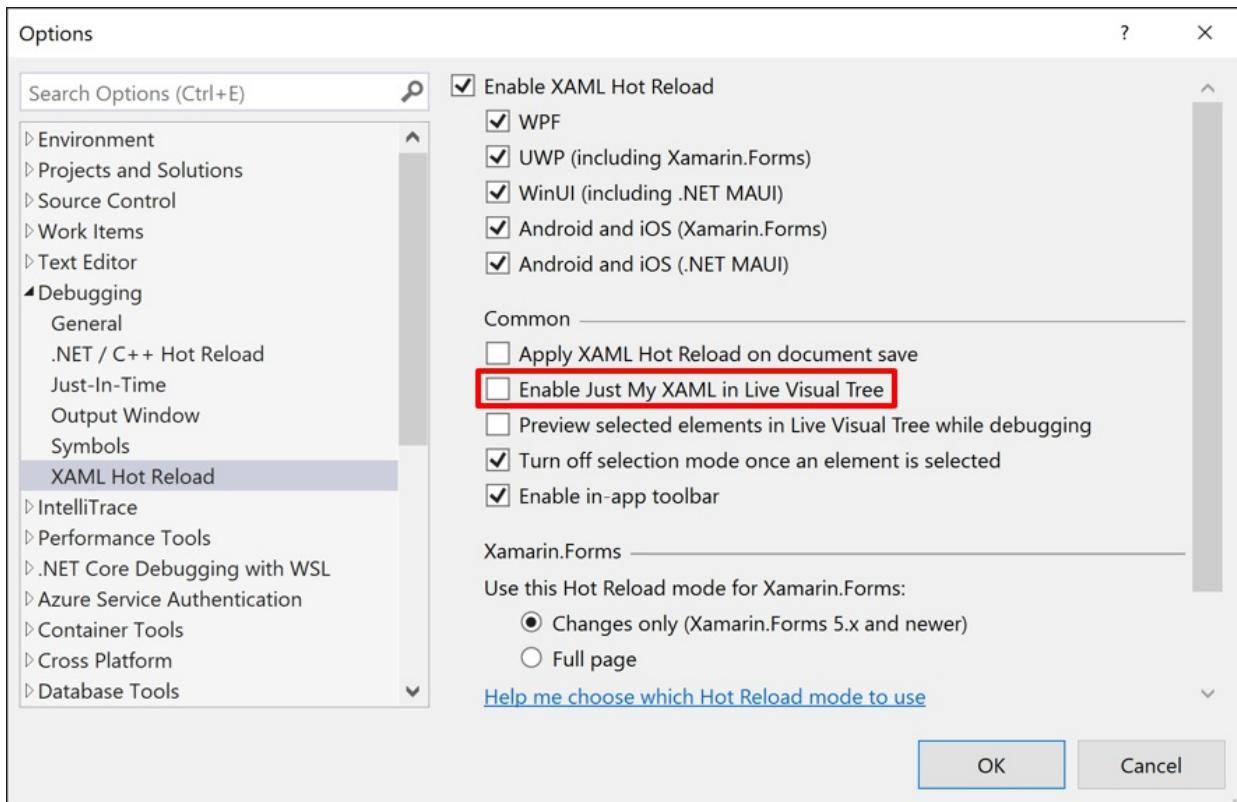
Provided that Hot Reload is enabled, the **Live Visual Tree** window will display the hierarchy of your app's UI elements regardless of whether the app's UI is built using XAML or C#. However, you will have to disable Just My XAML to display the hierarchy of your app's UI elements for UIs built using C#.

Just My XAML

The view of the UI elements is simplified by default using a feature called Just My XAML. Clicking the **Show Just My XAML** button disables the feature and shows all UI elements in the visual tree:



Just My XAML can be permanently disabled by selecting **Debug > Options > XAML Hot Reload** from the Visual Studio menu bar. Next, in the **Options** dialog box, ensure that **Enable Just My XAML in Live Visual Tree** is disabled:



Find a UI element

The structure of a XAML UI has a lot of elements that you may not be interested in, and if you don't have a full understanding of the app's source code you might have a difficult time navigating the visual tree to find the UI element that you're looking for. Therefore, the **Live Visual Tree** window has multiple approaches that let you use the app's UI to help you find the element you want to examine:

- **Select element in the running application.** You can enable this mode by clicking the **Select Element in the Running Application** button in the **Live Visual Tree** toolbar:



With this mode enabled, when you can select a UI element in the app the **Live Visual Tree** window automatically updates to show the node in the tree corresponding to that element.

- **Display layout adorners in the running application.** You can enable this mode by clicking the **Display Layout Adorners in the Running Application** button in the **Live Visual Tree** toolbar:



When this mode is enabled, it causes the app 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 clicking the **Preview Selected Item** button in the **Live Visual Tree** toolbar:



This mode shows the XAML source code where the element was declared, provided that you have access to the app source code.

Visual states

9/20/2022 • 9 minutes to read • [Edit Online](#)

The .NET Multi-platform App UI (.NET MAUI) Visual State Manager provides a structured way to make visual changes to the user interface from code. In most cases, the user interface of an app is defined in XAML, and this XAML can include markup describing how the Visual State Manager affects the visuals of the user interface.

The Visual State Manager introduces the concept of *visual states*. A .NET MAUI 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 .NET MAUI Visual State Manager defines a visual state group named `CommonStates` with the following visual states:

- Normal
- Disabled
- Focused
- Selected

The `Normal`, `Disabled`, and `Focused` visual states are supported on all classes that derive from `VisualElement`, which is the base class for `View` and `Page`. In addition, you can also define your own visual state groups and visual states.

The advantage of using the Visual State Manager to define appearance, rather than accessing visual elements directly from code-behind, 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.

NOTE

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 confusing. With the Visual State Manager, 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 visual 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, has input focus, or is selected. 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 Visual State Manager markup to an individual view, or you can define it in a style if it applies to multiple views.

Define visual states on a view

The `VisualStateManager` class defines a `VisualStateGroups` attached property, that's used to attach visual states to a view. The `visualStateGroups` property is of type `VisualStateGroupList`, which is a collection of `VisualStateGroup` objects. Therefore, the child of the `VisualStateManager.VisualStateGroup` attached property is a `VisualStateGroup` object. This object defines an `x:Name` attribute that indicates the name of the group. Alternatively, the `VisualStateGroup` class defines a `Name` property that you can use instead. For more information about attached properties, see [Attached properties](#).

The `VisualStateGroup` class defines a property named `States`, which is a collection of `VisualState` objects. `States` is the content property of the `VisualStateGroups` class so you can include the `VisualState` objects as children of the `VisualStateGroup`. Each `VisualState` object should be identified using `x:Name` or `Name`.

The `visualState` class 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` isn't the content property of `visualState`, so it's necessary to include property element tags for the `Setters` property. `Setter` objects should be inserted as children of `Setters`. Each `Setter` object indicates the value of a property when that state is current. Any property referenced by a `Setter` object must be backed by a bindable property.

IMPORTANT

In order for visual state `Setter` objects to function correctly, a `VisualStateGroup` must contain a `VisualState` object for the `Normal` state. If this visual state does not have any `Setter` objects, it should be included as an empty visual state (`<VisualState x:Name="Normal" />`).

The following example shows visual states defined on an `Entry`:

```
<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>
```

The following screenshot shows the `Entry` in its three defined visual states:



When the `Entry` is in the `Normal` state, its background is lime. When the `Entry` gains input focus its font size

doubles. When the `Entry` becomes disabled, its background becomes pink. The `Entry` doesn't retain its lime background when it gains input focus. As the Visual State Manager switches between the visual states, the properties set by the previous state are unset. Therefore, the visual states are mutually exclusive.

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>
```

Define visual states in a style

It's often necessary to share the same visual states in two or more views. In this scenario, the visual states can be defined in a `Style`. This can be achieved by adding a `Setter` object for the `VisualStateManager.VisualStateGroups` property. The content property for the `Setter` object is its `Value` property, which can therefore be specified as the child of the `Setter` object. The `VisualStateGroups` property is of type `VisualStateGroupList`, and so the child of the `Setter` object is a `VisualStateGroupList` to which a `VisualStateGroup` can be added that contains `VisualState` objects.

The following example shows an implicit style for an `Entry` that defines the common visual states:

```
<Style TargetType="Entry">
    <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>
```

When this style is included in a page-level resource dictionary, the `Style` object will be applied to all `Entry` objects on the page. Therefore, all `Entry` objects on the page will respond in the same way to their visual states.

Visual states in .NET MAUI

The following table lists the visual states that are defined in .NET MAUI:

CLASS	STATES	MORE INFORMATION
Button	Pressed	Button visual states
CheckBox	IsChecked	CheckBox visual states
ImageButton	Pressed	ImageButton visual states
RadioButton	Checked , Unchecked	RadioButton visual states
Switch	On , Off	Switch visual states
VisualElement	Normal , Disabled , Focused	Common states

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 object that the `Setter` for a visual state will manipulate. When the `TargetName` property is defined, the `Setter` sets the `Property` of the object 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 following example shows how to set state on multiple objects, from a single visual state group:

```

<StackLayout>
    <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>

```

In this example, the `Normal` state is active when the `Button` isn't pressed, and a response can be entered into the `Entry`. The `Pressed` state becomes active when the `Button` is pressed, and specifies that 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 unsupported in `Setter` elements that specify the `TargetName` property.

Define custom visual states

Custom visual states can be implemented by defining them as you would define visual states for the common states, but with names of your choosing, and then calling the `VisualStateManager.GoToState` method to activate a state.

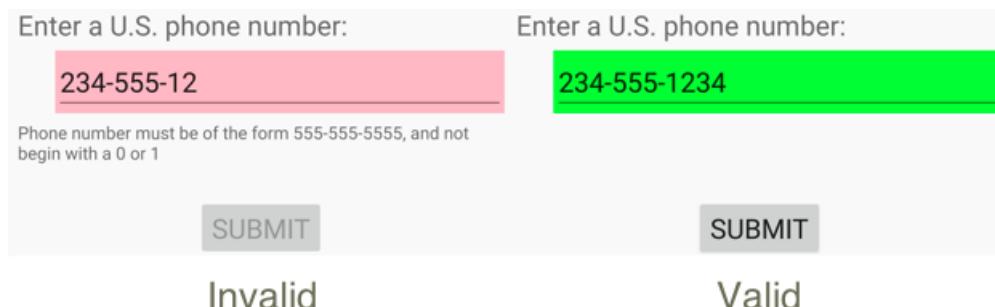
The following example shows how to use the Visual State Manager for input validation:

```

<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    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="18" />
        <Entry x:Name="entry"
            Placeholder="555-555-5555"
            FontSize="18"
            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="18"
            Margin="0, 20"
            VerticalOptions="Center"
            HorizontalOptions="Center" />
    </StackLayout>
</ContentPage>

```

In this example, visual states are attached to the `StackLayout`, and there are two mutually-exclusive states named `Valid` and `Invalid`. 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 `GoToState` method in the code-behind file calls the static `visualStateManager.GoToState` method on the `StackLayout` object:

```
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);
    }
}
```

In this example, the `GoToState` method is called from the constructor to initialize the state. There should always be a current state. The code-behind file then calls `VisualStateManager.GoToState`, with a state name, on the object that defines the visual states.

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, .NET MAUI 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](#).

Platform integration

9/20/2022 • 5 minutes to read • [Edit Online](#)

Each platform that .NET Multi-platform App UI (.NET MAUI) supports offers unique operating system and platform APIs that you can access from C#. .NET MAUI provides cross-platform APIs to access much of this platform functionality, which includes access to sensors, accessing information about the device an app is running on, checking network connectivity, storing data securely, and initiating browser-based authentication flows.

.NET MAUI separates these cross-platform APIs into different areas of functionality.

Application model

.NET MAUI provides the following functionality in the `Microsoft.Maui.ApplicationModel` namespace:

FUNCTIONALITY	DESCRIPTION
App actions	The <code>AppActions</code> class enables you to create and respond to app shortcuts, which provide additional ways of starting your app. For more information, see App actions .
App information	The <code>AppInfo</code> class provides access to basic app information, which includes the app name and version, and the current active theme for the device. For more information, see App information .
Browser	The <code>Browser</code> class enables an app to open a web link in an in-app browser, or the system browser. For more information, see Browser .
Launcher	The <code>Launcher</code> class enables an app to open a URI, and is often used when deep linking into another app's custom URI schemes. For more information, see Launcher .
Main thread	The <code>MainThread</code> class enables you to run code on the UI thread. For more information, see Main thread .
Maps	The <code>Map</code> class enables an app to open the system map app to a specific location or place mark. For more information, see Maps .
Permissions	The <code>Permissions</code> class enables you to check and request permissions at run-time. For more information, see Permissions .
Version tracking	The <code>VersionTracking</code> class enables you to check the app's version and build numbers, and determine if it's the first time the app has been launched. For more information, see Version tracking .

Communication

.NET MAUI provides the following functionality in the `Microsoft.Maui.ApplicationModel.Communication` namespace:

FUNCTIONALITY	DESCRIPTION
Contacts	The <code>Contacts</code> class enables an app to select a contact and read information about it. For more information, see Contacts .
Email	The <code>Email</code> class can be used to open the default email app, and can create a new email with the specified recipients, subject, and body. For more information, see Email .
Networking	The <code>Connectivity</code> class, in the <code>Microsoft.Maui.Networking</code> namespace, enables you to inspect the network accessibility of the device your app is running on. For more information, see Connectivity .
Phone dialer	The <code>PhoneDialer</code> class enables an app to open a phone number in the dialer. For more information, see Phone dialer .
SMS (messaging)	The <code>sms</code> class can be used to open the default SMS app and preload it with a recipient and message. For more information, see SMS .
Web authenticator	The <code>WebAuthenticator</code> class, in the <code>Microsoft.Maui.Authentication</code> namespace, enables you to start a browser-based authentication flow, which listens for a callback to a specific URL registered to the app. For more information, see Web authenticator .

Device features

.NET MAUI provides the following functionality in the `Microsoft.Maui.Devices` namespace:

FUNCTIONALITY	DESCRIPTION
Battery	The <code>Battery</code> class enables an app to check the device's battery information, and monitor the battery for changes. For more information, see Battery .
Device display	The <code>DeviceDisplay</code> class enables an app to read information about the device's screen metrics. For more information, see Device display .
Device information	The <code>DeviceInfo</code> class enables an app to read information about the device the app is running on. For more information, see Device information .
Device sensors	Types in the <code>Microsoft.Maui.Devices.Sensors</code> namespace provide access to the device's accelerometer, barometer, compass, gyroscope, magnetometer, and orientation sensor. For more information, see Device sensors .

FUNCTIONALITY	DESCRIPTION
Flashlight	The <code>FlashLight</code> class can toggle the device's camera flash on and off, to emulate a flashlight. For more information, see Flashlight .
Geocoding	The <code>Geocoding</code> class, in the <code>Microsoft.Maui.Devices.Sensors</code> namespace, provides APIs to geocode a place mark to a positional coordinate, and reverse geocode a coordinate to a place mark. For more information, see Geocoding .
Geolocation	The <code>Geolocation</code> class, in the <code>Microsoft.Maui.Devices.Sensors</code> namespace, provides APIs to retrieve the device's current geolocation coordinates. For more information, see Geolocation .
Haptic feedback	The <code>HapticFeedback</code> class controls haptic feedback on a device, which is generally manifested as a gentle vibration sensation to give a response to the user. For more information, see Haptic feedback .
Vibration	The <code>Vibration</code> class enables you to start and stop the vibrate functionality for a desired amount of time. For more information, see Vibration .

Media

.NET MAUI provides the following functionality in the `Microsoft.Maui.Media` namespace:

FUNCTIONALITY	DESCRIPTION
Media picker	The <code>MediaPicker</code> class enables you to prompt the user to pick or take a photo or video on the device. For more information, see Media picker .
Screenshot	The <code>Screenshot</code> class enables you to capture the current displayed screen of the app. For more information, see Screenshot .
Text-to-speech	The <code>TextToSpeech</code> class enables an app to utilize the built-in text-to-speech engines to speak text from the device. For more information, see Text-to-Speech .
Unit converters	The <code>UnitConverters</code> class provides unit converters to help you convert from one unit of measurement to another. For more information, see Unit converters .

Sharing

.NET MAUI provides the following functionality in the `Microsoft.Maui.ApplicationModel.DataTransfer` namespace:

FUNCTIONALITY	DESCRIPTION

FUNCTIONALITY	DESCRIPTION
Clipboard	The <code>Clipboard</code> class enables an app copy and paste text to and from the system clipboard. For more information, see Clipboard .
Share files and text	The <code>Share</code> class provides an API to send data, such as text or web links, to the device's share function. For more information, see Share .

Storage

.NET MAUI provides the following functionality in the `Microsoft.Maui.Storage` namespace:

FUNCTIONALITY	DESCRIPTION
File picker	The <code>FilePicker</code> class enables you to prompt the user to pick one or more files from the device. For more information, see File picker .
File system helpers	The <code>FileSystem</code> class provides helper methods that access the app's cache and data folders, and helps access files that are stored in the app package. For more information, see File system helpers .
Preferences	The <code>Preferences</code> class helps to store app preferences in a key/value store. For more information, see Preferences .
Secure storage	The <code>SecureStorage</code> class helps to securely store simple key/value pairs. For more information, see Secure storage .

Access platform APIs

.NET MAUI platform-specifics allow you to consume specific functionality that's only available on a specific platform. For more information, see [Android platform-specifics](#), [iOS platform-specifics](#), and [Windows platform-specifics](#).

In situations where .NET MAUI doesn't provide any APIs for accessing specific platform APIs, you can write your own code to access the required platform APIs. For more information, see [Invoke platform code](#).

App actions

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IAppActions` interface, which lets you create and respond to app shortcuts. App shortcuts are helpful to users because they allow you, as the app developer, to present them with extra ways of starting your app. For example, if you were developing an email and calendar app, you could present two different app actions, one to open the app directly to the current day of the calendar, and another to open to the email inbox folder.

The default implementation of the `IAppActions` interface is available through the `AppActions.Current` property. Both the `IAppActions` interface and `AppActions` class are contained in the `Microsoft.Maui.ApplicationModel` namespace.

Get started

To access the `AppActions` functionality, the following platform specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

In the `Platforms/Android/MainActivity.cs` file, add the following `IntentFilter` attribute to the `MainActivity` class:

```
[Activity(Theme = "@style/Maui.SplashTheme", MainLauncher = true, ConfigurationChanges =
ConfigChanges.ScreenSize | ConfigChanges.Orientation | ConfigChanges.UiMode | ConfigChanges.ScreenLayout |
ConfigChanges.SmallestScreenSize | ConfigChanges.Density)]
[IntentFilter(new[] { Platform.Intent.ActionAppAction },
    Categories = new[] { global::Android.Content.Intent.CategoryDefault })]
public class MainActivity : MauiAppCompatActivity
```

Create actions

App actions can be created at any time, but are often created when an app starts. To configure app actions, invoke the `ConfigureEssentials` method on the `MauiApplicationBuilder` object in the `MauiProgram.cs` file. There are two methods you must call on the `IEssentialsBuilder` object to enable an app action:

1. `AddAppAction`

This method creates an action. It takes an `id` string to uniquely identify the action, and a `title` string that's displayed to the user. You can optionally provide a subtitle and an icon.

2. `OnAppAction`

The delegate passed to this method is called when the user invokes an app action, provided the app action instance. Check the `Id` property of the action to determine which app action was started by the user.

The following code demonstrates how to configure the app actions at app startup:

```

public static MauiApp CreateMauiApp()
{
    var builder = MauiApp.CreateBuilder();

    builder
        .UseMauiApp<App>()
        .ConfigureFonts(fonts =>
    {
        fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
        fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
    })
        .ConfigureEssentials(essentials =>
    {
        essentials
            .AddAppAction("app_info", "App Info", icon: "app_info_action_icon")
            .AddAppAction("battery_info", "Battery Info")
            .OnAppAction(App.HandleAppActions);
    });
}

return builder.Build();
}

```

Responding to actions

After app actions [have been configured](#), the `OnAppAction` method is called for all app actions invoked by the user. Use the `Id` property to differentiate them. The following code demonstrates handling an app action:

```

public static void HandleAppActions(AppAction appAction)
{
    App.Current.Dispatcher.Dispatch(async () =>
    {
        var page = appAction.Id switch
        {
            "battery_info" => new SensorsPage(),
            "app_info" => new AppModelPage(),
            _ => default(Page)
        };

        if (page != null)
        {
            await Application.Current.MainPage.Navigation.PopToRootAsync();
            await Application.Current.MainPage.Navigation.PushAsync(page);
        }
    });
}

```

Check if app actions are supported

When you create an app action, either at app startup or while the app is being used, check to see if app actions are supported by reading the `AppActions.Current.IsSupported` property.

Create an app action outside of the startup bootstrap

To create app actions, call the `SetAsync` method:

```

if (AppActions.Current.IsSupported)
{
    await AppActions.Current.SetAsync(new[] { new AppAction("app_info", "App Info", icon: "app_info_action_icon"),
                                                new AppAction("battery_info", "Battery Info") });
}

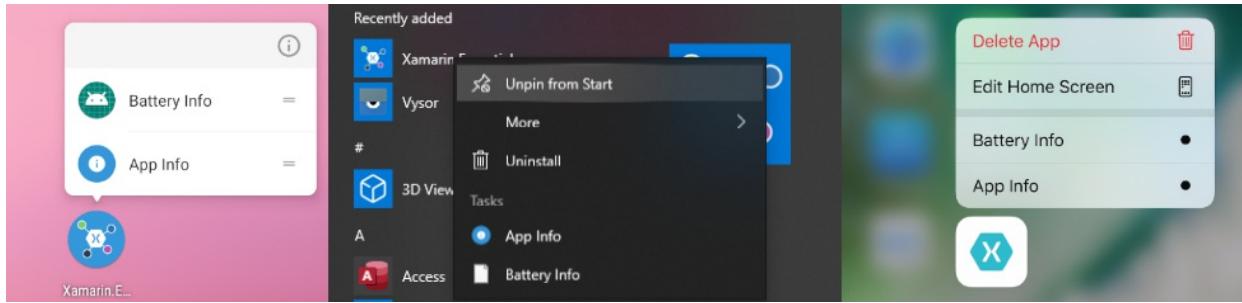
```

More information about app actions

If app actions aren't 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 subtitle to display under the title.
- **Icon:** Must match icons in the corresponding resources directory on each platform.



Get actions

You can get the current list of app actions by calling `AppActions.Current.GetAsync()`.

App information

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IAppInfo` interface, which provides information about your application.

The default implementation of the `IAppInfo` interface is available through the `AppInfo.Current` property. Both the `IAppInfo` interface and `AppInfo` class are contained in the `Microsoft.Maui.ApplicationModel` namespace.

Read the app information

There are four properties exposed by the `IAppInfo` interface:

- `IAppInfo.Name` — The application name
- `IAppInfo.PackageName` — The package name or application identifier, such as `com.microsoft.myapp`.
- `IAppInfo.VersionString` — The application version, such as `1.0.0`.
- `IAppInfo.BuildString` — The build number of the version, such as `1000`.

The following code example demonstrates accessing these properties:

```
string name = AppInfo.Current.Name;
string package = AppInfo.Current.PackageName;
string version = AppInfo.Current.VersionString;
string build = AppInfo.Current.BuildString;
```

Read the current theme

The `RequestedTheme` property provides the current requested theme by the system for your application. One of the following values is returned:

- `Unspecified`
- `Light`
- `Dark`

`Unspecified` is returned when the operating system doesn't have a specific user interface style. An example of this is on devices running versions of iOS older than 13.0.

The following code example demonstrates reading the theme:

```
ThemeInfoLabel.Text = AppInfo.Current.RequestedTheme switch
{
    AppTheme.Dark => "Dark theme",
    AppTheme.Light => "Light theme",
    _ => "Unknown"
};
```

Display app settings

The `IAppInfo` class can also display a page of settings maintained by the operating system for the application:

```
AppInfo.Current.ShowSettingsUI();
```

This settings page allows the user to change application permissions and perform other platform-specific tasks.

Platform implementation specifics

This section describes platform-specific implementation details related to the `IAppInfo` interface.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

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

Requested theme

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 may be changed when battery saver mode is enabled.

You can read more on the official [Android documentation for Dark Theme](#).

Browser

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IBrowser` interface. This interface enables an application to open a web link in the system-preferred browser or the external browser.

The default implementation of the `IBrowser` interface is available through the `Browser.Default` property. Both the `IBrowser` interface and `Browser` class are contained in the `Microsoft.Maui.ApplicationModel` namespace.

Get started

To access the browser functionality, the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

If your project's Target Android version is set to **Android 11 (R API 30)** or higher, you must update your *Android Manifest* with queries that use Android's [package visibility requirements](#).

In the *Platforms/Android/AndroidManifest.xml* file, add the following `queries/intent` nodes 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>
```

Open the browser

The browser is opened by calling the `Browser.OpenAsync` method with the `Uri` and the type of `BrowserLaunchMode`. The following code example demonstrates opening the browser:

```
private async void BrowserOpen_Clicked(object sender, EventArgs e)
{
    try
    {
        Uri uri = new Uri("https://www.microsoft.com");
        await Browser.Default.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 is launched, not after it was closed by the user. `Browser.OpenAsync` returns a `bool` value to indicate if the browser was successfully launched.

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`.

```
private async void BrowserCustomOpen_Clicked(object sender, EventArgs e)
{
    try
    {
        Uri uri = new Uri("https://www.microsoft.com");
        BrowserLaunchOptions options = new BrowserLaunchOptions()
        {
            LaunchMode = BrowserLaunchMode.SystemPreferred,
            TitleMode = BrowserTitleMode.Show,
            PreferredToolbarColor = Colors.Violet,
            PreferredControlColor = Colors.SandyBrown
        };

        await Browser.Default.OpenAsync(uri, options);
    }
    catch (Exception ex)
    {
        // An unexpected error occurred. No browser may be installed on the device.
    }
}
```

Platform differences

This section describes the platform-specific differences with the browser API.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

The `BrowserLaunchOptions.LaunchMode` determines how the browser is launched:

- `SystemPreferred`

[Custom Tabs](#) will try to be used to load the Uri and keep navigation awareness.

- `External`

An `Intent` is used to request the Uri be opened through the system's normal browser.

Launcher

9/20/2022 • 3 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `ILauncher` interface. This interface enables an application to open a URI by the system. This is often used when deep linking into another application's custom URI schemes.

The default implementation of the `ILauncher` interface is available through the `Launcher.Default` property. Both the `ILauncher` interface and `Launcher` class are contained in the `Microsoft.Maui.ApplicationModel` namespace.

IMPORTANT

To open the browser to a website, use the [Browser API](#) instead.

Get started

To access the launcher functionality, the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

No setup is required.

Open another app

To use the Launcher functionality, call the `Launcher.OpenAsync` method and pass in a `string` or `Uri` representing the app to open. Optionally, the `Launcher.CanOpenAsync` method can be used to check if the URI scheme can be handled by an app on the device. The following code demonstrates how to check if a URI scheme is supported or not, and then opens the URI:

```
bool supportsUri = await Launcher.Default.CanOpenAsync("lyft://");

if (supportsUri)
    await Launcher.Default.OpenAsync("lyft://ridetype?id=lyft_line");
```

The previous code example can be simplified by using the `TryOpenAsync`, which checks if the URI scheme can be opened, before opening it:

```
bool launcherOpened = await Launcher.Default.TryOpenAsync("lyft://ridetype?id=lyft_line");

if (launcherOpened)
{
    // Do something fun
}
```

Open another app via a file

The launcher can also be used to open an app with a selected file. .NET MAUI automatically detects the file type (MIME), and opens the default app for that file type. If more than one app is registered with the file type, an app

selection popover is shown to the user.

The following code example writes text to a file, and opens the text file with the launcher:

```
string popoverTitle = "Read text file";
string name = "File.txt";
string file = System.IO.Path.Combine(FileSystem.CacheDirectory, name);

System.IO.File.WriteAllText(file, "Hello World");

await Launcher.Default.OpenAsync(new OpenFileRequest(popoverTitle, new ReadOnlyFile(file)));
```

Set the launcher location

When requesting a share or opening launcher on iPadOS, you can present it in a popover. This specifies where the popover 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`.

Here are some extension methods that help calculate the bounds of a view:

```

public static class ViewHelpers
{
    public static Rectangle GetAbsoluteBounds(this Microsoft.Maui.Controls.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 Microsoft.Maui.Controls.View v)
            {
                absoluteX += v.X + v.Margin.Top;
                absoluteY += v.Y + v.Margin.Left;
            }
        }

        return new Rectangle(absoluteX, absoluteY, element.Width, element.Height);
    }
}

```

This can then be used when calling `RequestAsync`:

```

public Command<Microsoft.Maui.Controls.View> ShareCommand { get; } = new
Command<Microsoft.Maui.Controls.View>(Share);

async void Share(Microsoft.Maui.Controls.View element)
{
    try
    {
        await Share.Default.RequestAsync(new ShareTextRequest
        {
            PresentationSourceBounds = element.GetAbsoluteBounds(),
            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}}"/>

```

For an example of the `ViewHelpers` class, see the [.NET MAUI Sample hosted on GitHub](#).

Platform differences

This section describes the platform-specific differences with the launcher API.

- [Android](#)
- [iOS\macOS](#)

- Windows

The `Task` returned from `CanOpenAsync` completes immediately.

Create a thread on the .NET MAUI UI thread

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `MainThread` class to run code on the main UI thread. Most operating systems 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*, 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.

The `MainThread` class is available in the `Microsoft.Maui.ApplicationModel` namespace.

When is it required

Applications sometimes need to use events that call the event handler on a secondary thread, such as the `Accelerometer` or `Compass` sensors. All sensors might return information on a secondary thread when used with faster sensing speeds. If the event handler needs to access user-interface elements, it must invoke code on the main thread.

Run code on the UI thread

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, and then call that code with the `BeginInvokeOnMainThread` method:

```
void MyMainThreadCode()
{
    // Code to run on the main thread
}

MainThread.BeginInvokeOnMainThread(MyMainThreadCode);
```

Determine if invocation is required

With the `MainThread` class, you can determine if the current code is running on the main thread. The `MainThread.IsMainThread` property returns `true` if the code calling the property is running on the main thread, and `false` if it isn't. It's logical to assume that you need to determine if the code is running on the main thread before calling `BeginInvokeOnMainThread`. For example, the following code uses the `MainThread.IsMainThread` to detect if the `MyMainThreadCode` method should be called directly if the code is running on the main thread. If it isn't running on the main thread, the method is passed to `MainThread.BeginInvokeOnMainThread`:

```

if (MainThread.IsMainThread)
    MyMainThreadCode();

else
    MainThread.BeginInvokeOnMainThread(MyMainThreadCode);

```

This check isn't necessary. `BeginInvokeOnMainThread` itself tests if the current code is running on the main thread or not. If the code is running on the main thread, `BeginInvokeOnMainThread` just calls the provided method directly. If the code is running on a secondary thread, `BeginInvokeOnMainThread` invokes the provided method on the main thread. Therefore, if the code you run is the same, regardless of the main or secondary thread, simply call `BeginInvokeOnMainThread` without checking if it's required. There is negligible overhead in doing so.

The only reason you would need to check the `MainThread.IsMainThread` property is if you have branching logic that does something different based on the thread.

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.
<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.

Map

9/20/2022 • 3 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IMap` interface. This interface enables an application to open the installed map application to a specific location or place mark.

The default implementation of the `IMap` interface is available through the `Map.Default` property. Both the `IMap` interface and `Map` class are contained in the `Microsoft.Maui.ApplicationModel` namespace.

Get started

To access the browser functionality, the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

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. Google Maps supports this scheme.

In the `Platforms/Android/AndroidManifest.xml` file, add the following `queries/intent` nodes to the `manifest` node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="geo"/>
  </intent>
</queries>
```

Using the map

The map functionality works by calling the `IMap.OpenAsync` method, and passing either an instance of the `Location` or `Placemark` type. The following example opens the installed map app at a specific GPS location:

```
public async Task NavigateToBuilding25()
{
    var location = new Location(47.645160, -122.1306032);
    var options = new MapLaunchOptions { Name = "Microsoft Building 25" };

    try
    {
        await Map.Default.OpenAsync(location, options);
    }
    catch (Exception ex)
    {
        // No map application available to open
    }
}
```

TIP

The `Location` and `Placemark` types are in the `Microsoft.Maui.Devices.Sensors` namespace.

When you use a `Placemark` to open the map, more information is required. The information helps the map app search for the place you're looking for. The following information is required:

- `CountryName`
- `AdminArea`
- `Thoroughfare`
- `Locality`

```
public async Task NavigateToBuilding()
{
    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.Default.OpenAsync(placemark, options);
    }
    catch (Exception ex)
    {
        // No map application available to open or placemark can not be located
    }
}
```

Testing if the map opened

There's always the possibility that opening the map app failed, such as when there isn't a map app or your app doesn't have the correct permissions. For each `IMap.OpenAsync` method overload, there's a corresponding `IMap.TryOpenAsync` method, which returns a Boolean value indicating that the map app was successfully opened.

The following code example uses the `TryOpenAsync` method to open the map:

```
var location = new Location(47.645160, -122.1306032);
var options = new MapLaunchOptions { Name = "Microsoft Building 25" };

if (await Map.Default.TryOpenAsync(location, options) == false)
{
    // Map failed to open
}
```

Extension methods

As long as the `Microsoft.Maui.Devices.Sensors` namespace is imported, which a new .NET MAUI project automatically does, you can use the built-in extension method `OpenMapsAsync` to open the map:

```
public async Task NavigateToBuildingByPlacemark()
{
    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 placemark.OpenMapsAsync(options);
    }
    catch (Exception ex)
    {
        // No map application available to open or placemark can not be located
    }
}
```

Add navigation

When you open the map, you can calculate a route from the device's current location to the specified location.

Pass the `MapLaunchOptions` type to the `Map.OpenAsync` method, specifying the navigation mode. The following example opens the map app and specifies a driving navigation mode:

```
public async Task DriveToBuilding25()
{
    var location = new Location(47.645160, -122.1306032);
    var options = new MapLaunchOptions { Name = "Microsoft Building 25",
                                         NavigationMode = NavigationMode.Driving };

    try
    {
        await Map.Default.OpenAsync(location, options);
    }
    catch (Exception ex)
    {
        // No map application available to open
    }
}
```

Platform differences

This section describes the platform-specific differences with the maps API.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

`NavigationMode` supports Bicycling, Driving, and Walking.

Permissions

9/20/2022 • 5 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `Permissions` class. This class allows you to check and request permissions at run-time. The `Permissions` type is available in the `Microsoft.Maui.ApplicationModel` namespace.

Available permissions

.NET MAUI attempts to abstract as many permissions as possible. However, each operating system has a different set of permissions. Even though the API allows access to a common permission, there may be differences between operating systems related to that permission. The following table describes the available permissions:

The following table uses ✓ to indicate that the permission is supported and ✗ to indicate the permission isn't supported or isn't required:

PERMISSION	ANDROID	IOS	WINDOWS	TVOS
CalendarRead	✓	✓	✗	✗
CalendarWrite	✓	✓	✗	✗
Camera	✓	✓	✗	✗
ContactsRead	✓	✓	✓	✗
ContactsWrite	✓	✓	✓	✗
Flashlight	✓	✗	✗	✗
LocationWhenInUse	✓	✓	✓	✓
LocationAlways	✓	✓	✓	✗
Media	✗	✓	✗	✗
Microphone	✓	✓	✓	✗
Phone	✓	✓	✗	✗
Photos	✗	✓	✗	✓
Reminders	✗	✓	✗	✗
Sensors	✓	✓	✓	✗
Sms	✓	✓	✗	✗

PERMISSION	ANDROID	IOS	WINDOWS	TVOS
Speech	✓	✓	✗	✗
StorageRead	✓	✗	✗	✗
StorageWrite	✓	✗	✗	✗

If a permission is marked as **✗**, it will always return `Granted` when checked or requested.

Checking permissions

To check the current status of a permission, use the `Permissions.CheckStatusAsync` method along with the specific permission to get the status for. The following example checks the status of the `LocationWhenInUse` permission:

```
PermissionStatus status = await Permissions.CheckStatusAsync<Permissions.LocationWhenInUse>();
```

A `PermissionException` is thrown if the required permission isn't 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's 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. On Android, you can call `shouldShowRationale` to detect if the user has already denied the permission in the past.

Permission status

When using `CheckStatusAsync` or `RequestAsync`, a `PermissionStatus` is returned that can be used to determine the next steps:

- `Unknown`
The permission is in an unknown state, or on iOS, the user has never been prompted.
- `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.

Requesting permissions

To request a permission from the users, use the `Permissions.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` without showing a dialog to the user. The following example requests the `LocationWhenInUse` permission:

```
PermissionStatus status = await Permissions.RequestAsync<Permissions.LocationWhenInUse>();
```

A `PermissionException` is thrown if the required permission isn't declared.

IMPORTANT

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 then ask the user to manually turn it on.

Explain why permission is needed

It's best practice to explain to your user why your application needs a specific permission. On iOS, you must specify a string that is displayed to the user. Android doesn't 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's the first time the permission is being requested. The `ShouldShowRationale` method can be used to determine if an informative 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 always return `false` when calling this method.

Example

The following code presents the general usage pattern for determining whether a permission has been granted, and then requesting it if it hasn't.

```
public async Task<PermissionStatus> CheckAndRequestLocationPermission()
{
    PermissionStatus 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;
}
```

Extending permissions

The Permissions API was created to be flexible and extensible for applications that require more validation or permissions that aren't included in .NET MAUI. Create a class that inherits from `Permissions.BasePermission`, and implement the required abstract methods. The following example code demonstrates the basic abstract members, but without implementation:

```

public class MyPermission : Permissions.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();
    }

    // Indicates that the requestor should prompt the user as to why the app requires the permission,
    because the
    // user has previously denied this permission.
    public override bool ShouldShowRationale()
    {
        throw new NotImplementedException();
    }
}

```

When implementing a permission in a specific platform, the `BasePlatformPermission` class can be inherited from. This class provides extra platform helper methods to automatically check the permission declarations. This helps when creating custom permissions that do groupings, for example requesting both **Read** and **Write** access to storage on Android. The following code example demonstrates requesting **Read** and **Write** storage access:

```

public class ReadWriteStoragePerms : Permissions.BasePlatformPermission
{
    public override (string androidPermission, bool isRuntime)[] RequiredPermissions =>
        new List<(string androidPermission, bool isRuntime)>
    {
        (global::Android.Manifest.Permission.ReadExternalStorage, true),
        (global::Android.Manifest.Permission.WriteExternalStorage, true)
    }.ToArray();
}

```

You then check the permission in the same way as any other permission type provided by .NET MAUI:

```
PermissionStatus status = await Permissions.RequestAsync<ReadWriteStoragePerms>();
```

Platform differences

This section describes the platform-specific differences with the permissions API.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

Permissions must have the matching attributes set in the Android Manifest file. Permission status defaults to `Denied`.

Version tracking

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IVersionTracking` interface.

This interface lets you check the applications version and build numbers along with seeing additional information such as if it's the first time the application launched.

The default implementation of the `IVersionTracking` interface is available through the `VersionTracking.Default` property. Both the `IVersionTracking` interface and `VersionTracking` class are contained in the `Microsoft.Maui.ApplicationModel` namespace.

Get started

To enable version tracking in your app, invoke the `ConfigureEssentials` method on the `MauiAppBuilder` object in the *MauiProgram.cs* file. Then, on the `IEssentialsBuilder` object, call the `UseVersionTracking` method:

```
public static MauiApp CreateMauiApp()
{
    var builder = MauiApp.CreateBuilder();

    builder
        .UseMauiApp<App>()
        .ConfigureFonts(fonts =>
    {
        fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
        fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
    })
        .ConfigureEssentials(essentials =>
    {
        essentials.UseVersionTracking();
    });

    return builder.Build();
}
```

Check the version

The `IVersionTracking` interface provides many properties that describe the current version of the app and how it relates to the previous version. The following example writes the tracking information to labels on the page:

```
private void ReadVersion_Clicked(object sender, EventArgs e)
{
    labelIsFirst.Text = VersionTracking.Default isFirstLaunchEver.ToString();
    labelCurrentVersionIsFirst.Text = VersionTracking.Default isFirstLaunchForCurrentVersion.ToString();
    labelCurrentBuildIsFirst.Text = VersionTracking.Default isFirstLaunchForCurrentBuild.ToString();
    labelCurrentVersion.Text = VersionTracking.Default.CurrentVersion.ToString();
    labelCurrentBuild.Text = VersionTracking.Default.CurrentBuild.ToString();
    labelFirstInstalledVer.Text = VersionTracking.Default.FirstInstalledVersion.ToString();
    labelFirstInstalledBuild.Text = VersionTracking.Default.FirstInstalledBuild.ToString();
    labelVersionHistory.Text = String.Join(',', VersionTracking.Default.VersionHistory);
    labelBuildHistory.Text = String.Join(',', VersionTracking.Default.BuildHistory);

    // These two properties may be null if this is the first version
    labelPreviousVersion.Text = VersionTracking.Default.PreviousVersion?.ToString() ?? "none";
    labelPreviousBuild.Text = VersionTracking.Default.PreviousBuild?.ToString() ?? "none";
}
```

The first time the app is run after version tracking is enabled, the `IsFirstLaunchEver` property will return `true`. If you add version tracking in a newer version of an already released app, `IsFirstLaunchEver` may incorrectly report `true`. This property always returns `true` the first time version tracking is enabled and the user runs the app. You can't fully rely on this property if users have upgraded from older versions that weren't tracking the version.

Platform differences

All version information is stored using the [Preferences](#) API, and is stored with a filename of `[YOUR-APP-PACKAGE-ID].microsoft.maui.essentials.versiontracking` and follows the same data persistence outlined in the [Preferences](#) documentation.

Contacts

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IContacts` interface to select a contact and read information about it.

The default implementation of the `IContacts` interface is available through the `Contacts.Default` property. Both the `IContacts` interface and `Contacts` class are contained in the `Microsoft.Maui.ApplicationModel.Communication` namespace.

IMPORTANT

Because of a namespace conflict, the `Contacts` type must be fully qualified when targeting iOS or macOS:

`Microsoft.Maui.ApplicationModel.Communication.Contacts`. New projects automatically target these platforms, along with Android and Windows.

To write code that will compile for iOS and macOS, fully qualify the `Contacts` type. Alternatively, provide a `using` directive to map the `Communication` namespace:

```
using Communication = Microsoft.Maui.ApplicationModel.Communication;

// Code that uses the namespace:
var contact = await Communication.Contacts.Default.PickContactAsync();
```

Get started

To access the `Contacts` functionality the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

The `ReadContacts` permission is required and must be configured in the Android project. This can be added in the following ways:

- Add the assembly-based permission:

Open the `AssemblyInfo.cs` file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.ReadContacts)]
```

- OR -

- Update the Android Manifest:

Open the `AndroidManifest.xml` file under the **Properties** folder and add the following in the `manifest` node:

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

Pick a contact

You can request the user to pick a contact by calling the `PickContactAsync` method. A contact dialog will appear on the device allowing the user to select a contact. If the user doesn't select a contact, `null` is returned.

```
private async void SelectContactButton_Clicked(object sender, EventArgs e)
{
    try
    {
        var contact = await Contacts.Default.PickContactAsync();

        if (contact == null)
            return;

        string id = contact.Id;
        string namePrefix = contact.NamePrefix;
        string givenName = contact.GivenName;
        string middleName = contact.MiddleName;
        string familyName = contact.FamilyName;
        string nameSuffix = contact.NameSuffix;
        string displayName = contact.DisplayName;
        List<ContactPhone> phones = contact.Phones; // List of phone numbers
        List<ContactEmail> emails = contact.Emails; // List of email addresses
    }
    catch (Exception ex)
    {
        // Most likely permission denied
    }
}
```

Get all contacts

The `GetAllAsync` method returns a collection of contacts.

```
public async IAsyncEnumerable<string> GetContactNames()
{
    var contacts = await Contacts.Default.GetAllAsync();

    // No contacts
    if (contacts == null)
        yield break;

    foreach (var contact in contacts)
        yield return contact.DisplayName;
}
```

Platform differences

This section describes the platform-specific differences with the contacts API.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)
- The `cancellationToken` parameter in the `GetAllAsync` method isn't supported.

Email

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IEmail` interface to open the default email app. When the email app is loaded, it can be set to create a new email with the specified recipients, subject, and body.

The default implementation of the `IEmail` interface is available through the `Email.Default` property. Both the `IEmail` interface and `Email` class are contained in the `Microsoft.Maui.ApplicationModel.Communication` namespace.

Get started

To access the email functionality, the following platform specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

If your project's Target Android version is set to **Android 11 (R API 30)** or higher, you must update your *Android Manifest* with queries that use Android's [package visibility requirements](#).

In the *Platforms/Android/AndroidManifest.xml* file, add the following `queries/intent` nodes the `manifest` node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.SENDTO" />
    <data android:scheme="mailto" />
  </intent>
</queries>
```

Using Email

The Email functionality works by providing the email information as an argument to the `ComposeAsync` method. In this example, the `EmailMessage` type is used to represent the email information:

```
if (Email.Default.ComposeSupported)
{
    string subject = "Hello friends!";
    string body = "It was great to see you last weekend.";
    string[] recipients = new[] { "john@contoso.com", "jane@contoso.com" };

    var message = new EmailMessage
    {
        Subject = subject,
        Body = body,
        BodyFormat = EmailBodyFormat.PlainText,
        To = new List<string>(recipients)
    };

    await Email.Default.ComposeAsync(message);
}
```

File attachments

When creating the email provided to the email client, you can add file attachments. The file type (MIME) is automatically detected, so you don't need to specify it. Some mail clients may restrict the types of files you send, or possibly prevent attachments altogether.

Use the `EmailMessage.Attachments` collection to manage the files attached to an email.

The following example demonstrates adding arbitrary text to a file, and then adding it to the email.

```
if (Email.Default.IsComposeSupported)
{
    string subject = "Hello friends!";
    string body = "It was great to see you last weekend. I've attached a photo of our adventures together.";
    string[] recipients = new[] { "john@contoso.com", "jane@contoso.com" };

    var message = new EmailMessage
    {
        Subject = subject,
        Body = body,
        BodyFormat = EmailBodyFormat.PlainText,
        To = new List<string>(recipients)
    };

    string picturePath = Path.Combine(FileSystem.CacheDirectory, "memories.jpg");

    message.Attachments.Add(new EmailAttachment(picturePath));

    await Email.Default.ComposeAsync(message);
}
```

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

Not all email clients for Android support `EmailBodyFormat.Html`, since there is no way to detect this, we recommend using `EmailBodyFormat.PlainText` when sending emails.

Connectivity

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IConnectivity` interface to inspect the network accessibility of the device. The network connection may have access to the internet. Devices also contain different kinds of network connections, such as Bluetooth, cellular, or WiFi. The `IConnectivity` interface has an event to monitor changes in the devices connection state.

The default implementation of the `IConnectivity` interface is available through the `Connectivity.Current` property. Both the `IConnectivity` interface and `Connectivity` class are contained in the `Microsoft.Maui.Networking` namespace.

Get started

To access the **Connectivity** functionality, the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

The `AccessNetworkState` permission is required and must be configured in the Android project. This can be added in the following ways:

- Add the assembly-based permission:

Open the `Platforms/Android/MainApplication.cs` file and add the following assembly attributes after `using` directives:

```
[assembly: UsesPermission(Android.Manifest.Permission.AccessNetworkState)]
```

- OR -

- Update the Android Manifest:

Open the `Platforms/Android/AndroidManifest.xml` file and add the following in the `manifest` node:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Using Connectivity

You can determine the scope of the current network by checking the `NetworkAccess` property.

```
NetworkAccess accessType = Connectivity.Current.NetworkAccess;

if (accessType == NetworkAccess.Internet)
{
    // Connection to internet is available
}
```

Network access falls into the following categories:

- `Internet` — Local and internet access.
- `ConstrainedInternet` — Limited internet access. This value means that there's a captive portal, where local access to a web portal is provided. Once the portal is used to provide authentication credentials, internet access is granted.
- `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:

```
 IEnumerable<ConnectionProfile> profiles = Connectivity.Current.ConnectionProfiles;

if (profiles.Contains(ConnectionProfile.WiFi))
{
    // Active Wi-Fi connection.
}
```

Whenever the connection profile or network access changes, the `connectivityChanged` event is raised:

```

public class ConnectivityTest
{
    public ConnectivityTest() =>
        Connectivity.ConnectivityChanged += Connectivity_ConnectivityChanged;

    ~ConnectivityTest() =>
        Connectivity.ConnectivityChanged -= Connectivity_ConnectivityChanged;

    void Connectivity_ConnectivityChanged(object sender, ConnectivityChangedEventArgs e)
    {
        if (e.NetworkAccess == NetworkAccess.ConstrainedInternet)
            Console.WriteLine("Internet access is available but is limited.");

        else if (e.NetworkAccess != NetworkAccess.Internet)
            Console.WriteLine("Internet access has been lost.");

        // Log each active connection
        Console.Write("Connections active: ");

        foreach (var item in e.ConnectionProfiles)
        {
            switch (item)
            {
                case ConnectionProfile.Bluetooth:
                    Console.Write("Bluetooth");
                    break;
                case ConnectionProfile.Cellular:
                    Console.Write("Cell");
                    break;
                case ConnectionProfile.Ethernet:
                    Console.Write("Ethernet");
                    break;
                case ConnectionProfile.WiFi:
                    Console.Write("WiFi");
                    break;
                default:
                    break;
            }
        }

        Console.WriteLine();
    }
}

```

Limitations

It's important to know that it's possible that `Internet` is reported by `NetworkAccess` but full access to the web isn't available. Because of 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 isn't available.

Phone dialer

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IPhoneDialer` interface. This interface enables an application to open a phone number in the dialer.

The default implementation of the `IPhoneDialer` interface is available through the `PhoneDialer.Default` property. Both the `IPhoneDialer` interface and `PhoneDialer` class are contained in the `Microsoft.Maui.ApplicationModel.Communication` namespace.

Get started

To access the phone dialer functionality, the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

If your project's Target Android version is set to **Android 11 (R API 30)** or higher, you must update your *Android Manifest* with queries that use Android's [package visibility requirements](#).

In the *Platforms/Android/AndroidManifest.xml* file, add the following `queries/intent` nodes the `manifest` node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.DIAL" />
    <data android:scheme="tel"/>
  </intent>
</queries>
```

Open the phone dialer

The phone dialer functionality works by calling the `open` method with a phone number. When the phone dialer is opened, .NET MAUI will automatically attempt to format the number based on the country code, if specified.

```
if (PhoneDialer.Default.IsSupported)
    PhoneDialer.Default.Open("000-000-0000");
```

SMS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `ISms` interface to open the default SMS app and preload it with a message and recipient.

The default implementation of the `ISms` interface is available through the `Sms.Default` property. Both the `ISms` interface and `Sms` class are contained in the `Microsoft.Maui.ApplicationModel.Communication` namespace.

Get started

To access the SMS functionality, the following platform specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

If your project's Target Android version is set to **Android 11 (R API 30)** or higher, you must update your *Android Manifest* with queries that use Android's [package visibility requirements](#).

In the *Platforms/Android/AndroidManifest.xml* file, add the following `queries/intent` nodes the `manifest` node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="smsto"/>
  </intent>
</queries>
```

Create a message

The SMS functionality works by creating a new `SmsMessage` object, and calling the `ComposeAsync` method. You can optionally include a message and zero or more recipients.

```
if (Sms.Default.ComposeSupported)
{
    string[] recipients = new[] { "000-000-0000" };
    string text = "Hello, I'm interested in buying your vase.";

    var message = new SmsMessage(text, recipients);

    await Sms.Default.ComposeAsync(message);
}
```

Web authenticator

9/20/2022 • 7 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) the `IWebAuthenticator` interface. This interface lets you start browser-based authentication flows, which listen for a callback to a specific URL registered to the app.

The default implementation of the `IWebAuthenticator` interface is available through the `WebAuthenticator.Default` property. Both the `IWebAuthenticator` interface and `WebAuthenticator` class are contained in the `Microsoft.Maui.Authentication` namespace.

Overview

Many apps require adding user authentication, and this often means enabling your users to sign in to their existing Microsoft, Facebook, Google, or Apple Sign In account.

TIP

[Microsoft Authentication Library \(MSAL\)](#) provides an excellent turn-key solution to adding authentication to your app.

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 aren't a great place to store secrets and anything stored in a mobile app's code, binaries, or otherwise, is considered to be insecure.

The best practice here's 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 access the `WebAuthenticator` functionality the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

Android requires an **Intent Filter** setup to handle your callback URI. This is accomplished by inheriting from the `WebAuthenticatorCallbackActivity` class:

```

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

namespace YourNameSpace;

[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 :
    Microsoft.Maui.Authentication.WebAuthenticatorCallbackActivity
{
    const string CALLBACK_SCHEME = "myapp";

}

```

If your project's Target Android version is set to **Android 11 (R API 30)** or higher, you must update your *Android Manifest* with queries that use Android's [package visibility requirements](#).

In the *Platforms/Android/AndroidManifest.xml* file, add the following `queries/intent` nodes the `manifest` node:

```

<queries>
    <intent>
        <action android:name="android.support.customtabs.action.CustomTabsService" />
    </intent>
</queries>

```

Using WebAuthenticator

The API consists mainly of a single method, `AuthenticateAsync`, which takes two parameters:

1. The URL used to start the web browser flow.
2. The URI the flow is expected to ultimately call back to, that is registered to your app.

The result is a `WebAuthenticatorResult`, which includes any query parameters parsed from the callback URI:

```

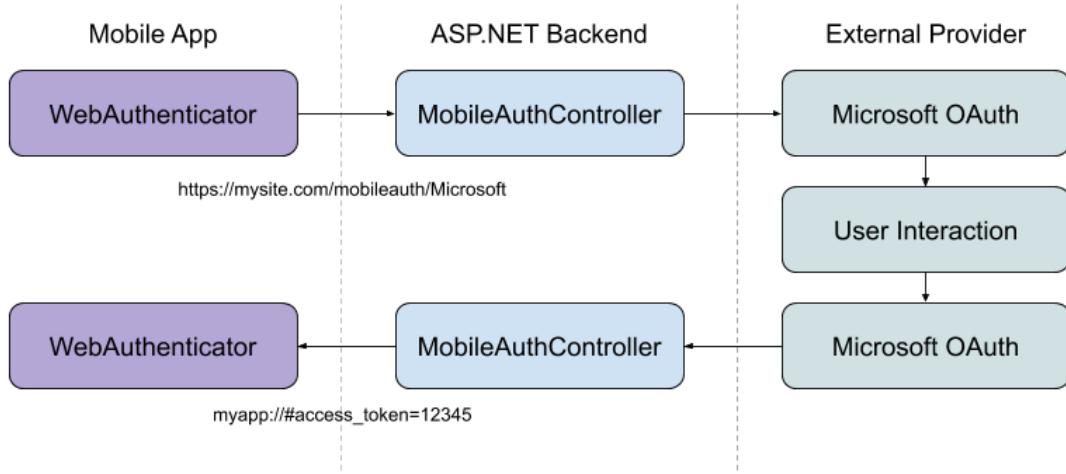
try
{
    WebAuthenticatorResult authResult = await WebAuthenticator.Default.AuthenticateAsync(
        new Uri("https://mysite.com/mobileauth/Microsoft"),
        new Uri("myapp://"));

    string accessToken = authResult?.AccessToken;

    // Do something with the token
}
catch (TaskCanceledException e)
{
    // Use stopped auth
}

```

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

`WebAuthenticatorOptions` parameter passed to the `AuthenticateAsync` method:

```

try
{
    WebAuthenticatorResult authResult = await WebAuthenticator.Default.AuthenticateAsync(
        new WebAuthenticatorOptions()
    {
        Url = new Uri("https://mysite.com/mobileauth/Microsoft"),
        CallbackUrl = new Uri("myapp://"),
        PrefersEphemeralWebBrowserSession = true
    });

    string accessToken = authResult?.AccessToken;

    // Do something with the token
}
catch (TaskCanceledException e)
{
    // Use stopped auth
}

```

Platform differences

This section describes the platform-specific differences with the web authentication API.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

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, call the `AppleSignInAuthenticator.AuthenticateAsync()` method. This will use automatically the native Apple Sign in APIs so your users get the best experience possible on these devices. For example, you can write your shared code to use the correct API at runtime:

```
var scheme = "..."; // Apple, Microsoft, Google, Facebook, etc.
var authUrlRoot = "https://mysite.com/mobileauth/";
WebAuthenticatorResult result = null;

if (scheme.Equals("Apple")
    && DeviceInfo.Platform == DevicePlatform.iOS
    && DeviceInfo.Version.Major >= 13)
{
    // Use Native Apple Sign In API's
    result = await AppleSignInAuthenticator.AuthenticateAsync();
}
else
{
    // Web Authentication flow
    var authUrl = new Uri($"{authUrlRoot}{scheme}");
    var callbackUrl = new Uri("myapp://");

    result = await WebAuthenticator.Default.AuthenticateAsync(authUrl, callbackUrl);
}

var authToken = string.Empty;

if (result.Properties.TryGetValue("name", out string name) && !string.IsNullOrEmpty(name))
    authToken += $"Name: {name}{Environment.NewLine}";

if (result.Properties.TryGetValue("email", out string email) && !string.IsNullOrEmpty(email))
    authToken += $"Email: {email}{Environment.NewLine}";

// Note that Apple Sign In has an IdToken and not an AccessToken
authToken += result?.AccessToken ?? result?.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 Windows 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, configure the web app with the following steps:

1. Set up your [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](#).

Add a custom mobile auth controller

With a mobile authentication flow, you usually start the flow directly to a provider the user has chosen. For example, clicking a "Microsoft" button on the sign-in screen of the app. It's also important 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) the app is requesting, and start 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](#).

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.

Battery

9/20/2022 • 3 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IBattery` interface to check the device's battery information and monitor for changes. This interface also provides information about the device's energy-saver status, which indicates if the device is running in a low-power mode.

The default implementation of the `IBattery` interface is available through the `Battery.Default` property. Both the `IBattery` interface and `Battery` class are contained in the `Microsoft.Maui.Devices` namespace.

Get started

To access the **Battery** functionality the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

The `Battery` permission is required and must be configured in the Android project. This can be added in the following ways:

- Add the assembly-based permission:

Open the `AssemblyInfo.cs` file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.BatteryStats)]
```

- OR -

- Update the Android Manifest:

In the **Solution Explorer**, open the `AndroidManifest.xml` file. This is typically located in the **Your-project > Platforms > Android** folder. Add the following node as a child to the `<manifest>` node:

```
<uses-permission android:name="android.permission.BATTERY_STATS" />
```

Check the battery status

The battery status can be checked by accessing the `Battery.Default` property, which is the default implementation of the `IBattery` interface. This interface defines various properties to provide information about the state of the battery. The `BatteryInfoChanged` event is also available, and is raised when the state of the battery changed.

The following example demonstrates how to use the monitor the `BatteryInfoChanged` event and report the battery status two `Label` controls:

```

private void BatterySwitch_Toggled(object sender, ToggledEventArgs e) =>
    WatchBattery();

private bool _isBatteryWatched;

private void WatchBattery()
{
    if (!_isBatteryWatched)
    {
        Battery.Default.BatteryInfoChanged += Battery_BatteryInfoChanged;
    }
    else
    {
        Battery.Default.BatteryInfoChanged -= Battery_BatteryInfoChanged;
    }

    _isBatteryWatched = !_isBatteryWatched;
}

private void Battery_BatteryInfoChanged(object sender, BatteryInfoChangedEventArgs e)
{
    BatteryStatusLabel.Text = e.State switch
    {
        BatteryState.Charging => "Battery is currently charging",
        BatteryState.Discharging => "Charger is not connected and the battery is discharging",
        BatteryState.Full => "Battery is full",
        BatteryState.NotCharging => "The battery isn't charging.",
        BatteryState.NotPresent => "Battery is not available.",
        BatteryState.Unknown => "Battery is unknown",
        _ => "Battery is unknown"
    };
    BatteryLevelLabel.Text = $"Battery is {e.ChargeLevel * 100}% charged.";
}

```

The `chargeLevel` property returns a value between 0.0 and 1.0, indicating the battery's charge level from empty to full, respectively.

Low-power energy-saver mode

Devices that run on batteries can be put into a low-power energy-saver mode. Sometimes devices are switched into this mode automatically, like 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.

IMPORTANT

Applications should avoid background processing if the device's energy-saver status is on.

The energy-saver status of the device can be read by accessing the `EnergySaverStatus` property, which is either `On`, `Off`, or `Unknown`. If the status is `On`, the application should avoid background processing or other activities that may consume a lot of power.

The battery will raise the `EnergySaverStatusChanged` event when the battery enters or leaves energy-saver mode. You can also obtain the current energy-saver status of the device using the `EnergySaverStatus` property:

The following code example monitors the energy-saver status and sets a property accordingly.

```

private bool _isBatteryLow = false;

private void BatterySaverSwitch_Toggled(object sender, ToggledEventArgs e)
{
    // Capture the initial state of the battery
    _isBatteryLow = Battery.Default.EnergySaverStatus == EnergySaverStatus.On;
    BatterySaverLabel.Text = _isBatteryLow.ToString();

    // Watch for any changes to the battery saver mode
    Battery.Default.EnergySaverStatusChanged += Battery_EnergySaverStatusChanged;
}

private void Battery_EnergySaverStatusChanged(object sender, EnergySaverStatusChangedEventArgs e)
{
    // Update the variable based on the state
    _isBatteryLow = Battery.Default.EnergySaverStatus == EnergySaverStatus.On;
    BatterySaverLabel.Text = _isBatteryLow.ToString();
}

```

Power source

The `PowerSource` property returns a `BatteryPowerSource` enumeration that indicates how the device is being charged, if at all. If it's not being charged, the status will be `Battery`. The `AC`, `Usb`, and `Wireless` values indicate that the battery is being charged.

The following code example sets the text of a `Label` control based on power source.

```

private void SetChargeModeLabel()
{
    BatteryPowerSourceLabel.Text = Battery.Default.PowerSource switch
    {
        BatteryPowerSource.Wireless => "Wireless charging",
        BatteryPowerSource.Usb => "USB cable charging",
        BatteryPowerSource.AC => "Device is plugged in to a power source",
        BatteryPowerSource.Battery => "Device isn't charging",
        _ => "Unknown"
    };
}

```

Platform differences

This section describes the platform-specific differences with the battery.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

No platform differences.

Device display information

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IDeviceDisplay` interface to read information about the device's screen metrics. This interface can be used to request the screen stays awake while the app is running.

The default implementation of the `IDeviceDisplay` interface is available through the `DeviceDisplay.Current` property. Both the `IDeviceDisplay` interface and `DeviceDisplay` class are contained in the `Microsoft.Maui.Devices` namespace.

Main display info

The `IDeviceDisplay.MainDisplayInfo` property returns information about the screen and orientation. The following code example uses the `Loaded` event of a page to read information about the current screen:

```
private void ReadDeviceDisplay()
{
    System.Text.StringBuilder sb = new System.Text.StringBuilder();

    sb.AppendLine($"Pixel width: {DeviceDisplay.Current.MainDisplayInfo.Width} / Pixel Height: {DeviceDisplay.Current.MainDisplayInfo.Height}");
    sb.AppendLine($"Density: {DeviceDisplay.Current.MainDisplayInfo.Density}");
    sb.AppendLine($"Orientation: {DeviceDisplay.Current.MainDisplayInfo.Orientation}");
    sb.AppendLine($"Rotation: {DeviceDisplay.Current.MainDisplayInfo.Rotation}");
    sb.AppendLine($"Refresh Rate: {DeviceDisplay.Current.MainDisplayInfo.RefreshRate}");

    DisplayDetailsLabel.Text = sb.ToString();
}
```

The `IDeviceDisplay` interface also provides the `MainDisplayInfoChanged` event that is raised when any screen metric changes, such as when the device orientation changes from `DisplayOrientation.Landscape` to `DisplayOrientation.Por`.

Keep the screen on

You can also prevent the device from locking or the screen turning off by setting the `IDeviceDisplay.KeepScreenOn` property to `true`. The following code example toggles the screen lock whenever the switch control is pressed:

```
private void AlwaysOnSwitch_Toggled(object sender, ToggledEventArgs e) =>
    DeviceDisplay.Current.KeepScreenOn = AlwaysOnSwitch.IsToggled;
```

Platform differences

This section describes the platform-specific differences with the device display.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

No platform differences.

Device information

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IDeviceInfo` interface to read information about the device the app is running on.

The default implementation of the `IDeviceInfo` interface is available through the `DeviceInfo.Current` property. Both the `IDeviceInfo` interface and `DeviceInfo` class are contained in the `Microsoft.Maui.Devices` namespace.

Read device info

The `IDeviceInfo` interface provides many properties that describe the device, such as the manufacturer and idiom. The following example demonstrates reading the device info properties:

```
private void ReadDeviceInfo()
{
    System.Text.StringBuilder sb = new System.Text.StringBuilder();

    sb.AppendLine($"Model: {DeviceInfo.Current.Model}");
    sb.AppendLine($"Manufacturer: {DeviceInfo.Current.Manufacturer}");
    sb.AppendLine($"Name: {DeviceInfo.Name}");
    sb.AppendLine($"OS Version: {DeviceInfo.VersionString}");
    sb.AppendLine($"Refresh Rate: {DeviceInfo.Current}");
    sb.AppendLine($"Idiom: {DeviceInfo.Current.Idiom}");
    sb.AppendLine($"Platform: {DeviceInfo.Current.Platform}");

    bool isVirtual = DeviceInfo.Current.DeviceType switch
    {
        DeviceType.Physical => false,
        DeviceType.Virtual => true,
        _ => false
    };

    sb.AppendLine($"Virtual device? {isVirtual}");

    DisplayDeviceLabel.Text = sb.ToString();
}
```

Get the device platform

The `IDeviceInfo.Platform` property represents the operating system the app is running on. The `DevicePlatform` type provides a property for each operating system:

- `DevicePlatform.Android`
- `DevicePlatform.iOS`
- `DevicePlatform.macOS`
- `DevicePlatform.MacCatalyst`
- `DevicePlatform.tvOS`
- `DevicePlatform.Tizen`
- `DevicePlatform.WinUI`
- `DevicePlatform.watchOS`
- `DevicePlatform.Unknown`

The following example demonstrates checking if the `DeviceInfo.Platform` property matches the `Android` operating system:

```
private bool IsAndroid() =>
    DeviceInfo.Current.Platform == DevicePlatform.Android;
```

Get the device type

The `DeviceInfo.Idiom` property represents the type of device the app is running on, such as a desktop computer or a tablet. The `DeviceIdiom` type provides a property for each type of device:

- `DeviceIdiom.Phone`
- `DeviceIdiom.Tablet`
- `DeviceIdiom.Desktop`
- `DeviceIdiom.TV`
- `DeviceIdiom.Watch`
- `DeviceIdiom.Unknown`

The following example demonstrates comparing the `DeviceInfo.Idiom` value to a `DeviceIdiom` property:

```
private void PrintIdiom()
{
    if (DeviceInfo.Current.Idiom == DeviceIdiom.Desktop)
        Console.WriteLine("The current device is a desktop");
    else if (DeviceInfo.Current.Idiom == DeviceIdiom.Phone)
        Console.WriteLine("The current device is a phone");
    else if (DeviceInfo.Current.Idiom == DeviceIdiom.Tablet)
        Console.WriteLine("The current device is a Tablet");
}
```

Device type

`IDeviceInfo.DeviceType` property an enumeration to determine if the application is running on a physical or virtual device. A virtual device is a simulator or emulator.

```
bool isVirtual = DeviceInfo.Current.DeviceType switch
{
    DeviceType.Physical => false,
    DeviceType.Virtual => true,
    _ => false
};
```

Platform differences

This section describes the platform-specific differences with the device information.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

No platform differences.

Accessing device sensors

9/20/2022 • 12 minutes to read • [Edit Online](#)

Devices have all sorts of sensors that are available to you. Some sensors can detect movement, others changes in the environment, such as light. Monitoring and reacting to these sensors makes your app dynamic in adapting to how the device is being used. You can also respond to changes in the sensors and alert the user. This article gives you a brief overview of the common sensors supported by .NET Multi-User Application (.NET MAUI).

Device sensor-related types are available in the `Microsoft.Maui.Devices.Sensors` namespace.

Sensor speed

Sensor speed sets the speed in which a sensor will return data to your app. When you start a sensor, you provide the desired sensor speed with the `SensorSpeed` enumeration.

- `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.

WARNING

Monitoring too many sensors at once may affect the rate sensor data is returned to your app.

Sensor event handlers

Event handlers added to sensors with either the `Game` or `Fastest` speeds aren't guaranteed to run on the UI thread. If the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

Accelerometer

The accelerometer sensor measures the acceleration of the device along its three axes. The data reported by the sensor represents how the user is moving the device.

The `IAccelerometer` interface provides access to the sensor, and is available through the `Accelerometer.Default` property. Both the `IAccelerometer` interface and `Accelerometer` class are contained in the `Microsoft.Maui.Devices.Sensors` namespace.

To start monitoring the accelerometer sensor, call the `IAccelerometer.Start` method. .NET MAUI sends accelerometer data changes to your app by raising the `IAccelerometer.RadingChanged` event. Use the `IAccelerometer.Stop` method to stop monitoring the sensor. You can detect the monitoring state of the accelerometer with the `IAccelerometer.IsMonitoring` property, which will be `true` if the accelerometer was started and is currently being monitored.

The following code example demonstrates monitoring the accelerometer for changes:

```
public void ToggleAccelerometer()
{
    if (Accelerometer.Default.IsSupported)
    {
        if (!Accelerometer.Default.IsMonitoring)
        {
            // Turn on accelerometer
            Accelerometer.Default.RadingChanged += Accelerometer_ReadingChanged;
            Accelerometer.Default.Start(SensorSpeed.UI);
        }
        else
        {
            // Turn off accelerometer
            Accelerometer.Default.Stop();
            Accelerometer.Default.RadingChanged -= Accelerometer_ReadingChanged;
        }
    }
}

private void Accelerometer_ReadingChanged(object sender, AccelerometerChangedEventArgs e)
{
    // Update UI Label with accelerometer state
    AccelLabel.TextColor = Colors.Green;
    AccelLabel.Text = $"Accel: {e.Rading}";
}
```

Accelerometer readings are reported back in G. A G is a unit of gravitation force equal to the gravity exerted by the earth's gravitational field \$(9.81 \text{ m/s}^2)\$.

The coordinate-system is defined relative to the screen of the device in its default orientation. The axes aren't 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 \text{ m/s}^2)\$ minus the force of gravity \$(-9.81 \text{ m/s}^2)\$ and normalized as in G.
- When the device lies flat on a table and is pushed toward the sky with an acceleration of A \$m/s^2\$, the acceleration value is equal to \$A+9.81\$ which corresponds to the acceleration of the device \$(+A \text{ m/s}^2)\$ minus the force of gravity \$(-9.81 \text{ m/s}^2)\$ and normalized in G.

Platform-specific information (Accelerometer)

There is no platform-specific information related to the accelerometer sensor.

Barometer

The barometer sensor measures the ambient air pressure. The data reported by the sensor represents the current air pressure. This data is reported the first time you start monitoring the sensor and then each time the pressure changes.

The `IBarometer` interface provides access to the sensor, and is available through the `Barometer.Default` property. Both the `IBarometer` interface and `Barometer` class are contained in the

`Microsoft.Maui.Devices.Sensors` namespace.

To start monitoring the barometer sensor, call the `IBarometer.Start` method. .NET MAUI sends air pressure readings to your app by raising the `IBarometer.ReadingChanged` event. Use the `IBarometer.Stop` method to stop monitoring the sensor. You can detect the monitoring state of the barometer with the `IBarometer.IsMonitoring` property, which will be `true` if the barometer is currently being monitored.

The pressure reading is represented in hectopascals.

The following code example demonstrates monitoring the barometer for changes:

```
public void ToggleBarometer()
{
    if (Barometer.Default.IsSupported)
    {
        if (!Barometer.Default.IsMonitoring)
        {
            // Turn on accelerometer
            Barometer.Default.ReadingChanged += Barometer_ReadingChanged;
            Barometer.Default.Start(SensorSpeed.UI);
        }
        else
        {
            // Turn off accelerometer
            Barometer.Default.Stop();
            Barometer.Default.ReadingChanged -= Barometer_ReadingChanged;
        }
    }
}

private void Barometer_ReadingChanged(object sender, BarometerChangedEventArgs e)
{
    // Update UI Label with barometer state
    BarometerLabel.TextColor = Colors.Green;
    BarometerLabel.Text = $"Barometer: {e.Reading}";
}
```

Platform-specific information (Barometer)

This section describes platform-specific implementation details related to the barometer sensor.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

No platform-specific implementation details.

Compass

The compass sensor monitors the device's magnetic north heading.

The `ICompass` interface provides access to the sensor, and is available through the `Compass.Default` property. Both the `ICompass` interface and `Compass` class are contained in the `Microsoft.Maui.Devices.Sensors` namespace.

To start monitoring the compass sensor, call the `ICompass.Start` method. .NET MAUI raises the `ICompass.ReadingChanged` event when the compass heading changes. Use the `ICompass.Stop` method to stop monitoring the sensor. You can detect the monitoring state of the compass with the `ICompass.IsMonitoring` property, which will be `true` if the compass is currently being monitored.

The following code example demonstrates monitoring the compass for changes:

```

private void ToggleCompass()
{
    if (Compass.Default.IsSupported)
    {
        if (!Compass.Default.IsMonitoring)
        {
            // Turn on compass
            Compass.Default.RadingChanged += Compass_ReadingChanged;
            Compass.Default.Start(SensorSpeed.UI);
        }
        else
        {
            // Turn off compass
            Compass.Default.Stop();
            Compass.Default.RadingChanged -= Compass_ReadingChanged;
        }
    }
}

private void Compass_ReadingChanged(object sender, CompassChangedEventArgs e)
{
    // Update UI Label with compass state
    CompassLabel.TextColor = Colors.Green;
    CompassLabel.Text = $"Compass: {e.Rading}";
}

```

Platform-specific information (Compass)

This section describes platform-specific implementation details related to the compass feature.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

Android doesn't provide an API for retrieving the compass heading. .NET MAUI uses the accelerometer and magnetometer sensors 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. Recalibrating the compass on Android varies by phone model and Android version. You'll need to search the internet on how to recalibrate the compass. Here are two links that may help in recalibrating the compass:

- [Google Help Center: Find and improve your location's accuracy](#)
- [Stack Exchange Android Enthusiasts: How can I calibrate the compass on my phone?](#)

Running multiple sensors from your app at the same time may impair the sensor speed.

Lowpass filter

Because of how the Android compass values are updated and calculated, there may be a need to smooth out the values. A *Lowpass 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.Default.Start(SensorSpeed.UI, applyLowPassFilter: true);
```

This is only applied on the Android platform, and the parameter is ignored on iOS and Windows. For more information, see [this GitHub issue comment](#).

Shake

Even though this article is listing **shake** as a sensor, it isn't. The [accelerometer](#) is used to detect when the device

is shaken.

The `IAccelerometer` interface provides access to the sensor, and is available through the `Accelerometer.Default` property. Both the `IAccelerometer` interface and `Accelerometer` class are contained in the `Microsoft.Maui.Devices.Sensors` namespace.

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 ($x^2+y^2+z^2$) readings from the accelerometer and comparing it to a specific threshold.

To start monitoring the accelerometer sensor, call the `IAccelerometer.Start` method. When a shake is detected, the `IAccelerometer.ShakeDetected` event is raised. Use the `IAccelerometer.Stop` method to stop monitoring the sensor. You can detect the monitoring state of the accelerometer with the `IAccelerometer.IsMonitoring` property, which will be `true` if the accelerometer was started and is currently being monitored.

It's recommended to use `Game` or faster for the `SensorSpeed`.

The following code example demonstrates monitoring the accelerometer for the `ShakeDetected` event:

```
private void ToggleShake()
{
    if (Accelerometer.Default.IsSupported)
    {
        if (!Accelerometer.Default.IsMonitoring)
        {
            // Turn on compass
            Accelerometer.Default.ShakeDetected += Accelerometer_ShakeDetected;
            Accelerometer.Default.Start(SensorSpeed.Game);
        }
        else
        {
            // Turn off compass
            Accelerometer.Default.Stop();
            Accelerometer.Default.ShakeDetected -= Accelerometer_ShakeDetected;
        }
    }
}

private void Accelerometer_ShakeDetected(object sender, EventArgs e)
{
    // Update UI Label with a "shaked detected" notice, in a randomized color
    ShakeLabel.TextColor = new Color(Random.Shared.Next(256), Random.Shared.Next(256),
    Random.Shared.Next(256));
    ShakeLabel.Text = $"Shake detected";
}
```

Platform-specific information (Shake)

There is no platform-specific information related to the accelerometer sensor.

Gyroscope

The gyroscope sensor measures the angular rotation speed around the device's three primary axes.

The `IGyroscope` interface provides access to the sensor, and is available through the `Gyroscope.Default` property. Both the `IGyroscope` interface and `Gyroscope` class are contained in the `Microsoft.Maui.Devices.Sensors` namespace.

To start monitoring the gyroscope sensor, call the `IGyroscope.Start` method. .NET MAUI sends gyroscope data changes to your app by raising the `IGyroscope.ReadingChanged` event. The data provided by this event is

measured in rad/s (radian per second). Use the `IGyroscope.Stop` method to stop monitoring the sensor. You can detect the monitoring state of the gyroscope with the `IGyroscope.IsMonitoring` property, which will be `true` if the gyroscope was started and is currently being monitored.

The following code example demonstrates monitoring the gyroscope:

```
private void ToggleGyroscope()
{
    if (Gyroscope.Default.IsSupported)
    {
        if (!Gyroscope.Default.IsMonitoring)
        {
            // Turn on compass
            Gyroscope.Default.ReadingChanged += Gyroscope_ReadingChanged;
            Gyroscope.Default.Start(SensorSpeed.UI);
        }
        else
        {
            // Turn off compass
            Gyroscope.Default.Stop();
            Gyroscope.Default.ReadingChanged -= Gyroscope_ReadingChanged;
        }
    }
}

private void Gyroscope_ReadingChanged(object sender, GyroscopeChangedEventArgs e)
{
    // Update UI Label with gyroscope state
    GyroscopeLabel.TextColor = Colors.Green;
    GyroscopeLabel.Text = $"Gyroscope: {e.Reading}";
}
```

Platform-specific information (Gyroscope)

There is no platform-specific information related to the gyroscope sensor.

Magnetometer

The magnetometer sensor indicates the device's orientation relative to Earth's magnetic field.

The `IMagnetometer` interface provides access to the sensor, and is available through the `Magnetometer.Default` property. Both the `IMagnetometer` interface and `Magnetometer` class are contained in the `Microsoft.Maui.Devices.Sensors` namespace.

To start monitoring the magnetometer sensor, call the `IMagnetometer.Start` method. .NET MAUI sends magnetometer data changes to your app by raising the `IMagnetometer.ReadingChanged` event. The data provided by this event is measured in μT (microteslas). Use the `IMagnetometer.Stop` method to stop monitoring the sensor. You can detect the monitoring state of the magnetometer with the `IMagnetometer.IsMonitoring` property, which will be `true` if the magnetometer was started and is currently being monitored.

The following code example demonstrates monitoring the magnetometer:

```

private void ToggleMagnetometer()
{
    if (Magnetometer.Default.IsSupported)
    {
        if (!Magnetometer.Default.IsMonitoring)
        {
            // Turn on compass
            Magnetometer.Default.RadingChanged += Magnetometer_ReadingChanged;
            Magnetometer.Default.Start(SensorSpeed.UI);
        }
        else
        {
            // Turn off compass
            Magnetometer.Default.Stop();
            Magnetometer.Default.RadingChanged -= Magnetometer_ReadingChanged;
        }
    }
}

private void Magnetometer_ReadingChanged(object sender, MagnetometerChangedEventArgs e)
{
    // Update UI Label with magnetometer state
    MagnetometerLabel.TextColor = Colors.Green;
    MagnetometerLabel.Text = $"Magnetometer: {e.Rading}";
}

```

Platform-specific information (Magnetometer)

There is no platform-specific information related to the magnetometer sensor.

Orientation

The orientation sensor monitors the orientation of a device in 3D space.

NOTE

This sensor isn't used for determining 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.

The `IOrientationSensor` interface provides access to the sensor, and is available through the `OrientationSensor.Default` property. Both the `IOrientationSensor` interface and `OrientationSensor` class are contained in the `Microsoft.Maui.Devices.Sensors` namespace.

To start monitoring the orientation sensor, call the `IOrientationSensor.Start` method. .NET MAUI sends orientation data changes to your app by raising the `IOrientationSensor.RadingChanged` event. Use the `IOrientationSensor.Stop` method to stop monitoring the sensor. You can detect the monitoring state of the orientation with the `IOrientationSensor.IsMonitoring` property, which will be `true` if the orientation was started and is currently being monitored.

The following code example demonstrates monitoring the orientation sensor:

```

private void ToggleOrientation()
{
    if (OrientationSensor.Default.IsSupported)
    {
        if (!OrientationSensor.Default.IsMonitoring)
        {
            // Turn on compass
            OrientationSensor.Default.RadingChanged += Orientation_ReadingChanged;
            OrientationSensor.Default.Start(SensorSpeed.UI);
        }
        else
        {
            // Turn off compass
            OrientationSensor.Default.Stop();
            OrientationSensor.Default.RadingChanged -= Orientation_ReadingChanged;
        }
    }
}

private void Orientation_ReadingChanged(object sender, OrientationSensorChangedEventArgs e)
{
    // Update UI Label with orientation state
    OrientationLabel.TextColor = Colors.Green;
    OrientationLabel.Text = $"Orientation: {e.Rading}";
}

```

`IOrientationSensor` 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 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:

$$(a_x \sin(\theta/2), a_y \sin(\theta/2), a_z \sin(\theta/2), \cos(\theta/2))$$

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).

Platform-specific information (Orientation)

There is no platform-specific information related to the orientation sensor.

Flashlight

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IFlashlight` interface. With this interface, you can toggle the device's camera flash on and off, to emulate a flashlight.

The default implementation of the `IFlashlight` interface is available through the `Flashlight.Default` property. Both the `IFlashlight` interface and `Flashlight` class are contained in the `Microsoft.Maui.Devices` namespace.

The `Flashlight` class is available in the `Microsoft.Maui.Devices` namespace.

Get started

To access the flashlight functionality the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

There are two permissions to configure in your project: `Flashlight` and `Camera`. These permissions can be set in the following ways:

- Add the assembly-based permission:

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 the Android Manifest:

Open the `AndroidManifest.xml` file under the **Properties** folder and add the following in the `manifest` node:

```
<uses-permission android:name="android.permission.FLASHLIGHT" />
<uses-permission android:name="android.permission.CAMERA" />
```

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)]
```

Use Flashlight

The flashlight can be turned on and off through the `TurnOnAsync` and `TurnOffAsync` methods. The following code example ties the flashlight's on or off state to a `Switch` control:

```
private async void FlashlightSwitch_Toggled(object sender, ToggledEventArgs e)
{
    try
    {
        if (FlashlightSwitch.IsToggled)
            await Flashlight.Default.TurnOnAsync();
        else
            await Flashlight.Default.TurnOffAsync();
    }
    catch (FeatureNotSupportedException ex)
    {
        // Handle not supported on device exception
    }
    catch (PermissionException ex)
    {
        // Handle permission exception
    }
    catch (Exception ex)
    {
        // Unable to turn on/off flashlight
    }
}
```

Platform differences

This section describes the platform-specific differences with the flashlight.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

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.

Geocoding

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IGeocoding` interface. This interface provides APIs to geocode a placemark to a positional coordinates and reverse geocode coordinates to a placemark. The `IGeocoding` interface is exposed through the `Geocoding.Default` property.

The default implementation of the `IGeocoding` interface is available through the `Geocoding.Default` property. Both the `IGeocoding` interface and `Geocoding` class are contained in the `Microsoft.Maui.Devices.Sensors` namespace.

Get started

To access the **Geocoding** functionality the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

No setup is required.

Use geocoding

The following example demonstrates how to get the location coordinates for an address:

```
string address = "Microsoft Building 25 Redmond WA USA";
IEnumerable<Location> locations = await Geocoding.Default.GetLocationsAsync(address);

Location location = locations?.FirstOrDefault();

if (location != null)
    Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude: {location.Altitude}");
```

The altitude isn't always available. If it isn't available, the `Altitude` property might be `null`, or the value might be `0`. If the altitude is available, the value is in meters above sea level.

Reverse geocoding

Reverse geocoding is the process of getting placemarks for an existing set of coordinates. The following example demonstrates getting placemarks:

```
private async Task<string> GetGeocodeReverseData(double latitude = 47.673988, double longitude = -122.121513)
{
    IEnumerable<Placemark> placemarks = await Geocoding.Default.GetPlacemarksAsync(latitude, longitude);

    Placemark placemark = placemarks?.FirstOrDefault();

    if (placemark != null)
    {
        return
            $"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";
    }

    return "";
}
```

Get the distance between two locations

The `Location` and `LocationExtensions` classes define methods to calculate the distance between two locations. For an example of getting the distance between two locations, see [Distance between two locations](#).

Geolocation

9/20/2022 • 6 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IGeolocation` interface. This interface provides APIs to retrieve the device's current geolocation coordinates.

The default implementation of the `IGeolocation` interface is available through the `Geolocation.Default` property. Both the `IGeolocation` interface and `Geolocation` class are contained in the `Microsoft.Maui.Devices.Sensors` namespace.

Get started

To access the **Geolocation** functionality, the following platform-specific setup is required:

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

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:

- Add the assembly-based permission:

Open the *Platforms/Android/MainApplication.cs* file and add the following assembly attributes after `using` directives:

```
[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)]
```

If your application is targeting Android 10 - Q (API Level 29 or higher) and is requesting `LocationAlways`, you must also add this permission request:

```
[assembly: UsesPermission(Manifest.Permission.AccessBackgroundLocation)]
```

- OR -

- Update the Android Manifest:

Open the *Platforms/Android/AndroidManifest.xml* file and add the following in 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" />
```

If your application is targeting Android 10 - Q (API Level 29 or higher) and is requesting `LocationAlways`,

you must also add this permission request:

```
<uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
```

TIP

Be sure to read the [Android documentation on background location updates](#), as there are many restrictions that need to be considered.

Get the last known location

The device may have cached the most recent location of the device. Use the `GetLastKnownLocationAsync` method to access the cached location, if available. This is often faster than doing a full location query, but can be less accurate. If no cached location exists, this method returns `null`.

NOTE

When necessary, the Geolocation API prompts the user for permissions.

The following code example demonstrates checking for a cached location:

```
public async Task<string> GetCachedLocation()
{
    try
    {
        Location location = await Geolocation.Default.GetLastKnownLocationAsync();

        if (location != null)
            return $"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
    }

    return "None";
}
```

Depending on the device, not all location values may be available. For example, the `Altitude` property might be `null`, have a value of 0, or have a positive value indicating the meters above sea level. Other values that may not be present include the `Speed` and `Course` properties.

Get the current location

While checking for the [last known location](#) of the device may be quicker, it can be inaccurate. Use the `GetLocationAsync` method to query the device for the current location. You can configure the accuracy and timeout of the query. It's best to use the method overload that uses the `GeolocationRequest` and `CancellationToken` parameters, since it may take some time to get the device's location.

NOTE

When necessary, the Geolocation API prompts the user for permissions.

The following code example demonstrates how to request the device's location, while supporting cancellation:

```
private CancellationTokenSource _cancelTokenSource;
private bool _isCheckingLocation;

public async Task GetCurrentLocation()
{
    try
    {
        _isCheckingLocation = true;

        GeolocationRequest request = new GeolocationRequest(GeolocationAccuracy.Medium,
TimeSpan.FromSeconds(10));

        _cancelTokenSource = new CancellationTokenSource();

        Location location = await Geolocation.Default.GetLocationAsync(request, _cancelTokenSource.Token);

        if (location != null)
            Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude: {location.Altitude}");
    }
    // Catch one of the following exceptions:
    //  FeatureNotSupportedException
    //  FeatureNotEnabledException
    //  PermissionException
    catch (Exception ex)
    {
        // Unable to get location
    }
    finally
    {
        _isCheckingLocation = false;
    }
}

public void CancelRequest()
{
    if (_isCheckingLocation && _cancelTokenSource != null && _cancelTokenSource.IsCancellationRequested == false)
        _cancelTokenSource.Cancel();
}
```

Not all location values may be available, depending on the device. For example, the `Altitude` property might be `null`, have a value of 0, or have a positive value indicating the meters above sea level. Other values that may not be present include `Speed` and `Course`.

Accuracy

The following sections outline the location accuracy distance, per platform:

IMPORTANT

iOS has some limitations regarding accuracy. For more information, see the [Platform differences](#) section.

Lowest

PLATFORM	DISTANCE (IN METERS)
Android	500
iOS	3000
Windows	1000 - 5000

Low

PLATFORM	DISTANCE (IN METERS)
Android	500
iOS	1000
Windows	300 - 3000

Medium (Default)

PLATFORM	DISTANCE (IN METERS)
Android	100 - 500
iOS	100
Windows	30-500

High

PLATFORM	DISTANCE (IN METERS)
Android	0 - 100
iOS	10
Windows	<= 10

Best

PLATFORM	DISTANCE (IN METERS)
Android	0 - 100
iOS	~0
Windows	<= 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`:

```
public async Task CheckMock()
{
    GeolocationRequest request = new GeolocationRequest(GeolocationAccuracy.Medium);
    Location location = await Geolocation.Default.GetLocationAsync(request);

    if (location != null && location.IsFromMockProvider)
    {
        // location is from a mock provider
    }
}
```

Distance between two locations

The `Location.CalculateDistance` method calculates the distance between two geographic locations. This calculated distance doesn't take roads or other pathways into account, and is merely the shortest distance between the two points along the surface of the Earth. This calculation is known as the *great-circle distance* calculation.

The following code calculates the distance between the United States of America cities of Boston and San Francisco:

```
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 accepts the latitude and longitude arguments, respectively. 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

This section describes the platform-specific differences with the geolocation API.

Altitude is calculated differently on each platform.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

On Android, `altitude`, if available, is returned in meters above the WGS 84 reference ellipsoid. If this location doesn't have an altitude, `0.0` is returned.

The `Location.ReducedAccuracy` property is only used by iOS and returns `false` on all other platforms.

Haptic feedback

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IHapticFeedback` interface to control haptic feedback on a device. Haptic feedback is generally manifested by a gentle vibration sensation provided by the device to give a response to the user. Some examples of haptic feedback are when a user types on a virtual keyboard or when they play a game where the player's character has an encounter with an enemy character.

The default implementation of the `IHapticFeedback` interface is available through the `HapticFeedback.Default` property. Both the `IHapticFeedback` interface and `HapticFeedback` class are contained in the `Microsoft.Maui.Devices` namespace.

Get started

To access the haptic feedback functionality, the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

The `vibrate` permission is required and must be configured in the Android project. This can be added in the following ways:

- Add the assembly-based permission:

Open the `AssemblyInfo.cs` file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.Vibrate)]
```

- or -

- Update the Android Manifest:

Open the `AndroidManifest.xml` file under the **Properties** folder and add the following in the `manifest` node:

```
<uses-permission android:name="android.permission.VIBRATE" />
```

Use haptic feedback

The haptic feedback functionality is performed in two modes: a short `Click` or a `LongPress`. The following code example initiates a `Click` or `LongPress` haptic feedback response to the user based on which `Button` they click:

```
private void HapticShortButton_Clicked(object sender, EventArgs e) =>
    HapticFeedback.Default.Perform(HapticFeedbackType.Click);

private void HapticLongButton_Clicked(object sender, EventArgs e) =>
    HapticFeedback.Default.Perform(HapticFeedbackType.LongPress);
```

Vibration

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IVibration` interface. This interface lets you start and stop the vibrate functionality for a desired amount of time.

The default implementation of the `IVibration` interface is available through the `Vibration.Default` property. Both the `IVibration` interface and `Vibration` class are contained in the `Microsoft.Maui.Devices` namespace.

Get started

To access the Vibration functionality, the following platform specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

The `VIBRATE` permission is required, and must be configured in the Android project. This can be added in the following ways:

- Add the assembly-based permission:

Open the `AssemblyInfo.cs` file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.Vibrate)]
```

- OR -

- Update the Android Manifest:

Open the `AndroidManifest.xml` file under the **Properties** folder and add the following in the `manifest` node:

```
<uses-permission android:name="android.permission.VIBRATE" />
```

Vibrate the device

The vibration functionality can be requested for a set amount of time or the default of 500 milliseconds. The following code example randomly vibrates the device between one and seven seconds:

```
private void VibrateStartButton_Clicked(object sender, EventArgs e)
{
    int secondsToVibrate = Random.Shared.Next(1, 7);
    TimeSpan vibrationLength = TimeSpan.FromSeconds(secondsToVibrate);

    Vibration.Default.Vibrate(vibrationLength);
}

private void VibrateStopButton_Clicked(object sender, EventArgs e) =>
    Vibration.Default.Cancel();
```

Platform differences

This section describes the platform-specific differences with the vibration API.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

No platform differences.

Media picker

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IMediaPicker` interface. This interface lets a user pick or take a photo or video on the device.

The default implementation of the `IMediaPicker` interface is available through the `MediaPicker.Default` property. Both the `IMediaPicker` interface and `MediaPicker` class are contained in the `Microsoft.Maui.Media` namespace.

Get started

To access the media picker functionality, the following platform-specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

The `CAMERA`, `WRITE_EXTERNAL_STORAGE`, `READ_EXTERNAL_STORAGE` permissions are required, and must be configured in the Android project. These can be added in the following ways:

- Add the assembly-based permission:

Open the `Platforms/Android/MainApplication.cs` file and add the following assembly attributes after `using` directives:

```
// 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 the Android Manifest:

Open the `Platforms/Android/AndroidManifest.xml` file and add the following in 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" />
```

If your project's Target Android version is set to **Android 11 (R API 30)** or higher, you must update your *Android Manifest* with queries that use Android's [package visibility requirements](#).

In the `Platforms/Android/AndroidManifest.xml` file, add the following `queries/intent` nodes the `manifest` node:

```
<queries>
<intent>
    <action android:name="android.media.action.IMAGE_CAPTURE" />
</intent>
</queries>
```

Using media picker

The `IMediaPicker` interface has the following methods that all return a `FileResult`, which can be used to get the file's location or read it.

- `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 type that allows the `Title` to be set on some operating systems, which is displayed to the user.

IMPORTANT

All methods must be called on the UI thread because permission checks and requests are automatically handled by .NET MAUI.

Take a photo

Call the `CapturePhotoAsync` method to open the camera and let the user take a photo. If the user takes a photo, the return value of the method will be a non-null value. The following code sample uses the media picker to take a photo and save it to the cache directory:

```
public async void TakePhoto()
{
    if (MediaPicker.Default.IsCaptureSupported)
    {
        FileResult photo = await MediaPicker.Default.CapturePhotoAsync();

        if (photo != null)
        {
            // save the file into local storage
            string localFilePath = Path.Combine(FileSystem.CacheDirectory, photo.FileName);

            using Stream sourceStream = await photo.OpenReadAsync();
            using FileStream localFileStream = File.OpenWrite(localFilePath);

            await sourceStream.CopyToAsync(localFileStream);
        }
    }
}
```

TIP

The `FullPath` property doesn't always return the physical path to the file. To get the file, use the `OpenReadAsync` method.

Screenshot

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IScreenshot` interface. This interface lets you take a capture of the current displayed screen of the app.

The default implementation of the `IScreenshot` interface is available through the `Screenshot.Default` property. Both the `IScreenshot` interface and `Screenshot` class are contained in the `Microsoft.Maui.Media` namespace.

Capture a screenshot

To capture a screenshot of the current app, use the `CaptureAsync` method. This method returns a `IScreenshotResult`, which contains information about the capture, such as the width and height of the screenshot. `IScreenshotResult` also includes a `Stream` property that is used to convert the screenshot into an image object for use by your app. The following example demonstrates a method that captures a screenshot and returns it as an `ImageSource`.

```
public async Task<ImageSource> TakeScreenshotAsync()
{
    if (Screenshot.Default.IsCaptureSupported)
    {
        IScreenshotResult screen = await Screenshot.Default.CaptureAsync();

        Stream stream = await screen.OpenReadAsync();

        return ImageSource.FromStream(() => stream);
    }

    return null;
}
```

Limitations

Not all views support being captured at a screen level, such as an OpenGL view.

Text-to-Speech

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `ITextToSpeech` interface. This interface enables an application to utilize the built-in text-to-speech engines to speak back text from the device. You can also use it to query for available languages.

The default implementation of the `ITextToSpeech` interface is available through the `TextToSpeech.Default` property. Both the `ITextToSpeech` interface and `TextToSpeech` class are contained in the `Microsoft.Maui.Media` namespace.

Using Text-to-Speech

Text-to-speech works by calling the `SpeakAsync` method with the text to speak, as the following code example demonstrates:

```
public async void Speak() =>
    await TextToSpeech.Default.SpeakAsync("Hello World");
```

This method takes in an optional `CancellationToken` to stop the utterance once it starts.

```
CancellationTokenSource cts;

public async Task SpeakNowDefaultSettingsAsync()
{
    cts = new CancellationTokenSource();
    await TextToSpeech.Default.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.WhenAll(
        TextToSpeech.Default.SpeakAsync("Hello World 1"),
        TextToSpeech.Default.SpeakAsync("Hello World 2"),
        TextToSpeech.Default.SpeakAsync("Hello World 3"))
        .ContinueWith((t) => { isBusy = false; }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

Settings

To control the volume, pitch, and locale of the voice, use the `SpeechOptions` class. Pass an instance of that class to the `SpeakAsync` method. the `GetLocalesAsync` method retrieves a collection of the locales provided by the operating system.

```
public async void SpeakSettings()
{
    IEnumerable<Locale> locales = await TextToSpeech.Default.GetLocalesAsync();

    SpeechOptions options = new SpeechOptions()
    {
        Pitch = 1.5f, // 0.0 - 2.0
        Volume = 0.75f, // 0.0 - 1.0
        Locale = locales.FirstOrDefault()
    };

    await TextToSpeech.Default.SpeakAsync("How nice to meet you!", options);
}
```

The following are supported values for these parameters:

PARAMETER	MINIMUM	MAXIMUM
Pitch	0	2.0
Volume	0	1.0

Limitations

- Utterance queueing is not guaranteed if called across multiple threads.
- Background audio playback is not officially supported.

Unit converters

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `UnitConverters` class. This class provides several unit converters to help developers convert from one unit of measurement to another.

Using unit converters

All unit converters are available by using the static `Microsoft.Maui.Media.UnitConverters` class. For example, you can convert Fahrenheit to Celsius with the `FahrenheitToCelsius` method:

```
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`

Android platform-specifics

9/20/2022 • 2 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) platform-specifics allow you to consume functionality that's only available on a specific platform, without customizing handlers.

The following platform-specific functionality is provided for .NET MAUI views 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 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 .NET MAUI pages on Android:

- Disabling transition animations when navigating through pages in a `TabPage`. For more information, see [TabPage page transition animations on Android](#).
- Enabling swiping between pages in a `TabPage`. For more information, see [TabPage page swiping on Android](#).
- Setting the toolbar placement and color on a `TabPage`. For more information, see [TabPage toolbar placement on android](#).

The following platform-specific functionality is provided for the .NET MAUI `Application` class on Android:

- Setting the operating mode of a soft keyboard. For more information, see [Soft keyboard input mode on Android](#).

Entry input method editor options on Android

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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 ...>
    <StackLayout ...>
        <Entry ... android:Entry.ImeOptions="Send" />
        ...
    </StackLayout>
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;
...
entry.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>().SetImeOptions(ImeFlags.Send);
```

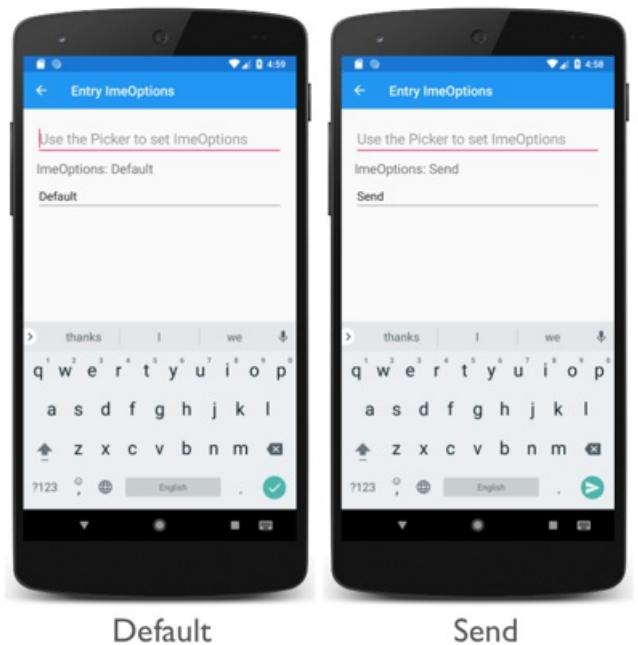
The `Entry.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>` method specifies that this platform-specific will only run on Android. The `Entry.SetImeOptions` method, in the

`Microsoft.Maui.Controls.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:



ListView fast scrolling on Android

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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 ...>
    <StackLayout ...
        xmlns:android="clr-namespace:Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;assembly=Microsoft.Maui.Controls">
        <ListView ItemsSource="{Binding GroupedEmployees}"
            GroupDisplayBinding="{Binding Key}"
            IsGroupingEnabled="true"
            android:ListView.IsFastScrollEnabled="true">
        </ListView>
    </StackLayout>
</ContentPage>
```

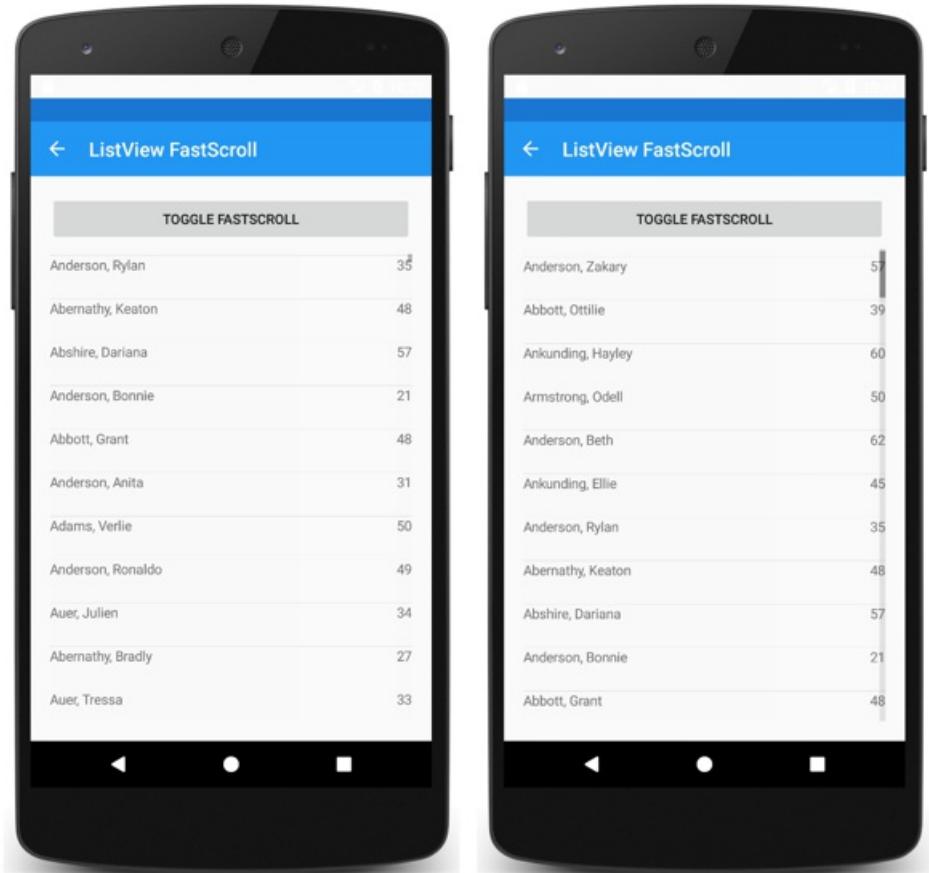
Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;
...
var listView = new Microsoft.Maui.Controls.ListView { IsGroupingEnabled = true, ... };
listView.SetBinding(ItemsView<Cell>.ItemsSourceProperty, "GroupedEmployees");
listView.GroupDisplayBinding = new Binding("Key");
listView.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>().SetIsFastScrollEnabled(true);
```

The `ListView.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>` method specifies that this platform-specific will only run on Android. The `ListView.SetIsFastScrollEnabled` method, in the `Microsoft.Maui.Controls.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<Microsoft.Maui.Controls.PlatformConfiguration.Android>()
    ().SetIsFastScrollEnabled(!listView.On<Microsoft.Maui.Controls.PlatformConfiguration.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

Soft keyboard input mode on Android

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:

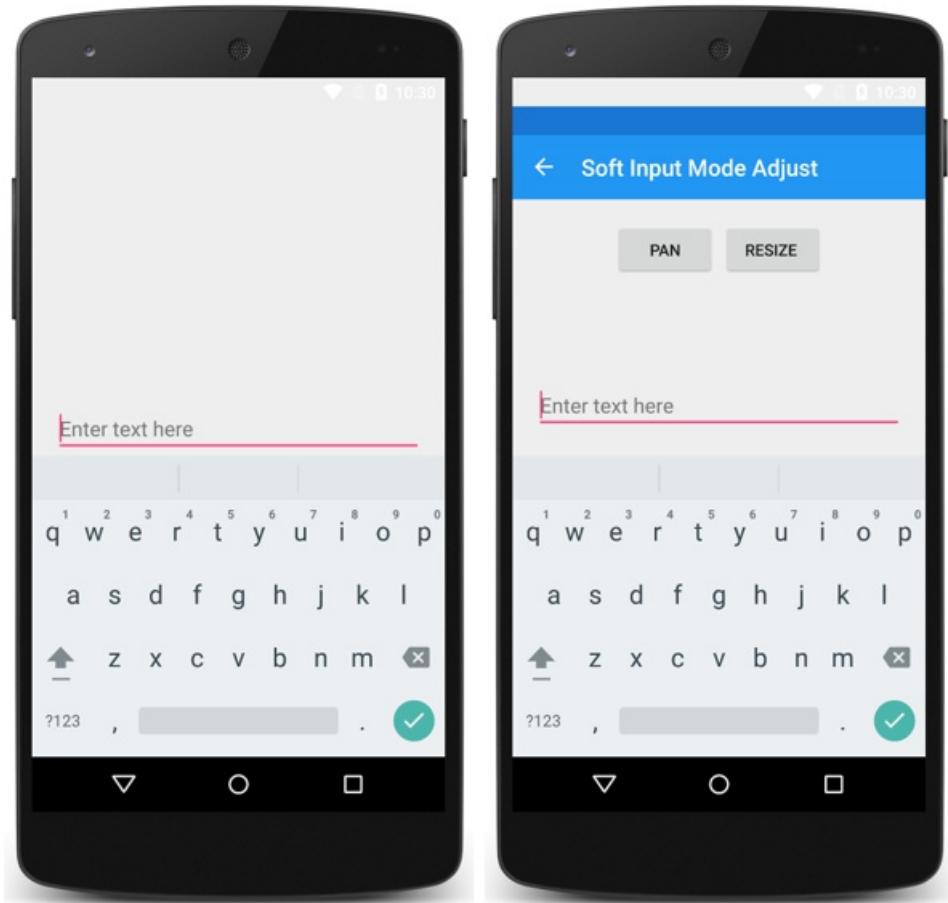
```
<Application ...  
    xmlns:android="clr-  
    namespace:Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;assembly=Microsoft.Maui.Controls"  
    android:Application.WindowSoftInputModeAdjust="Resize">  
    ...  
</Application>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;  
...  
App.Current.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>  
().UseWindowSoftInputModeAdjust(WindowSoftInputModeAdjust.Resize);
```

The `Application.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>` method specifies that this platform-specific will only run on Android. The `Application.UseWindowSoftInputModeAdjust` method, in the `Microsoft.Maui.Controls.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

SwipeView swipe transition mode on Android

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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 ...>
    <x:Bind xmlns:android="clr-namespace:Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;assembly=Microsoft.Maui.Controls" />
    <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:

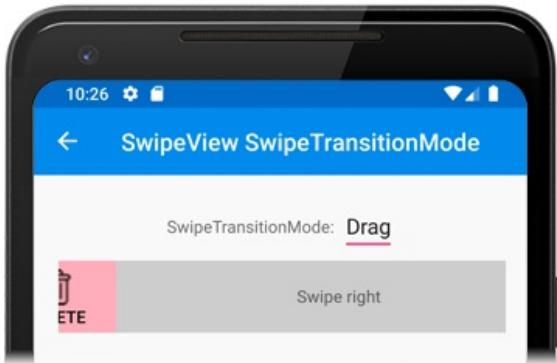
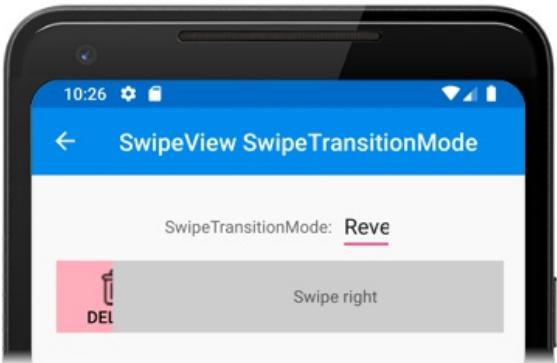
```
using Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;
...
SwipeView swipeView = new Microsoft.Maui.Controls.SwipeView();
swipeView.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>()
    ().SetSwipeTransitionMode(SwipeTransitionMode.Drag);
// ...
```

The `SwipeView.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>` method specifies that this platform-specific will only run on Android. The `SwipeView.SetSwipeTransitionMode` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific` 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`:



TabPage page swiping on Android

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:

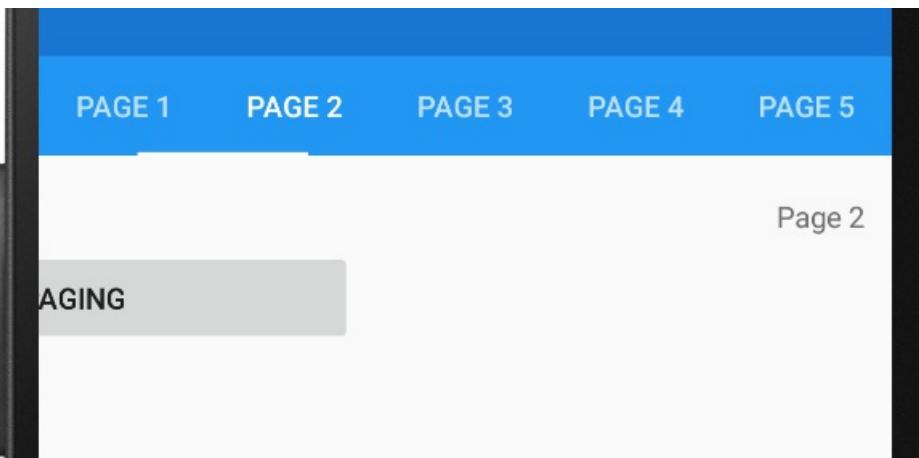
```
<TabPage ...  
    xmlns:android="clr-  
    namespace:Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;assembly=Microsoft.Maui.Controls"  
        android:TabPage.OffscreenPageLimit="2"  
        android:TabPage.IsSwipePagingEnabled="true">  
    ...  
</TabPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;  
...  
  
On<Microsoft.Maui.Controls.PlatformConfiguration.Android>()  
    .SetOffscreenPageLimit(2)  
    .SetIsSwipePagingEnabled(true);
```

The `TabPage.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>` method specifies that this platform-specific will only run on Android. The `TabPage.SetIsSwipePagingEnabled` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific` namespace, is used to enable swiping between pages in a `TabPage`. In addition, the `TabPage` class in the `Microsoft.Maui.Controls.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:



TabPage page transition animations on Android

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;assembly=Microsoft.Maui.Controls"  
    android:TabPage.IsSmoothScrollEnabled="false">  
    ...  
</TabPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;  
...  
On<Microsoft.Maui.Controls.PlatformConfiguration.Android>().SetIsSmoothScrollEnabled(false);
```

The `TabPage.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>` method specifies that this platform-specific will only run on Android. The `TabPage.SetIsSmoothScrollEnabled` method, in the `Microsoft.Maui.Controls.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 `Microsoft.Maui.Controls.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`.

TabPage toolbar placement on Android

9/20/2022 • 2 minutes to read • [Edit Online](#)

These .NET Multi-platform App UI (.NET MAUI) Android platform-specifics are used to set the placement 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:

```
<TabPage ...  
    xmlns:android="clr-  
    namespace:Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;assembly=Microsoft.Maui.Controls"  
    android:TabPage.ToolbarPlacement="Bottom">  
    ...  
</TabPage>
```

Alternatively, they can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;  
...  
On<Microsoft.Maui.Controls.PlatformConfiguration.Android>().SetToolbarPlacement(ToolbarPlacement.Bottom);
```

The `TabPage.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>` method specifies that this platform-specific will only run on Android. The `TabPage.SetToolbarPlacement` method, in the `Microsoft.Maui.Controls.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.

NOTE

The `GetToolbarPlacement` method can be used to retrieve the placement of the `TabPage` toolbar.

The result is that the toolbar placement can be set on a `TabPage`:



WebView mixed content on Android

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) Android platform-specific controls whether a `WebView` can display mixed content. Mixed content is content that's initially loaded over an HTTPS connection, but which loads resources (such as images, audio, video, stylesheets, scripts) over an HTTP connection. It's consumed in XAML by setting the `WebView.MixedContentMode` attached property to a value of the `MixedContentHandling` enumeration:

```
<ContentPage ...  
    xmlns:android="clr-  
    namespace:Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;assembly=Microsoft.Maui.Controls">  
    <WebView ... android:WebView.MixedContentMode="AlwaysAllow" />  
</ContentPage>
```

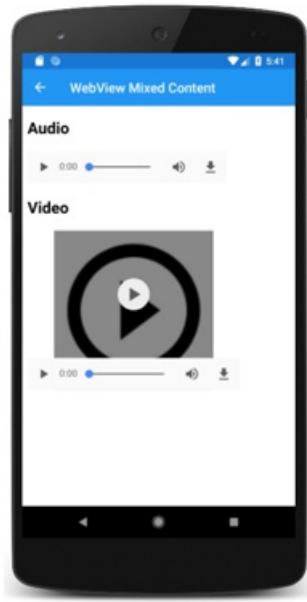
Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;  
...  
  
webView.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>  
().SetMixedContentMode(MixedContentHandling.AlwaysAllow);
```

The `WebView.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>` method specifies that this platform-specific will only run on Android. The `WebView.SetMixedContentMode` method, in the `Microsoft.Maui.Controls.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.

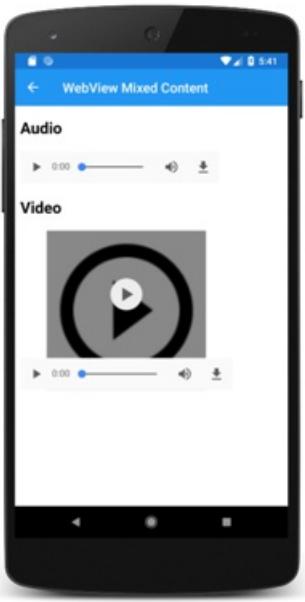
The result is that a specified `MixedContentHandling` value is applied to the `WebView`, which controls whether mixed content can be displayed:



AlwaysAllow



NeverAllow



CompatibilityMode

WebView zoom on Android

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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 ...>
    <!-- Other XAML code -->
    <WebView Source="https://www.microsoft.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 Microsoft.Maui.Controls.PlatformConfiguration.AndroidSpecific;
...
webView.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>()
    .EnableZoomControls(true)
    .DisplayZoomControls(true);
```

The `WebView.On<Microsoft.Maui.Controls.PlatformConfiguration.Android>` method specifies that this platform-specific will only run on Android. The `WebView.EnableZoomControls` method, in the `Microsoft.Maui.Controls.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`.

iOS platform-specifics

9/20/2022 • 2 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) platform-specifics allow you to consume functionality that's only available on a specific platform, without customizing handlers.

The following platform-specific functionality is provided for .NET MAUI 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 .NET MAUI 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](#).
- Controlling whether the navigation bar is translucent. For more information, see [Navigation bar translucency 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 presentation style of modal pages. For more information, see [Modal page presentation style](#).
- Setting the translucency mode of the tab bar on a `TabPage`. For more information, see [TabPage translucent TabBar on iOS](#).

The following platform-specific functionality is provided for .NET MAUI 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 .NET MAUI `Application` class 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](#).

Cell background color on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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`:

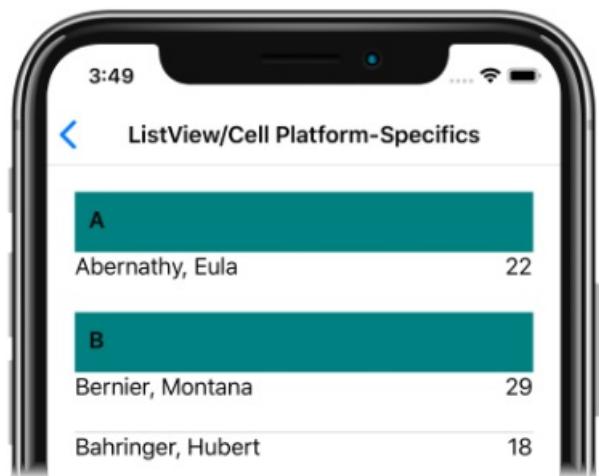
```
<ContentPage ...>
    xmlns:ios="clr-namespace:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">
        <StackLayout Margin="20">
            <ListView ItemsSource="{Binding GroupedEmployees}" IsGroupingEnabled="true">
                <ListView.GroupHeaderTemplate>
                    <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 Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;
...
ViewCell viewCell = new ViewCell { View = ... };
viewCell.On<iOS>().SetDefaultBackgroundColor(Colors.Teal);
```

The `ListView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Cell.SetDefaultBackgroundColor` method, in the `Microsoft.Maui.Controls.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`:



DatePicker item selection on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) iOS platform-specific controls when item selection occurs in a `DatePicker`, allowing you 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">
        <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 Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;
...
datePicker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
```

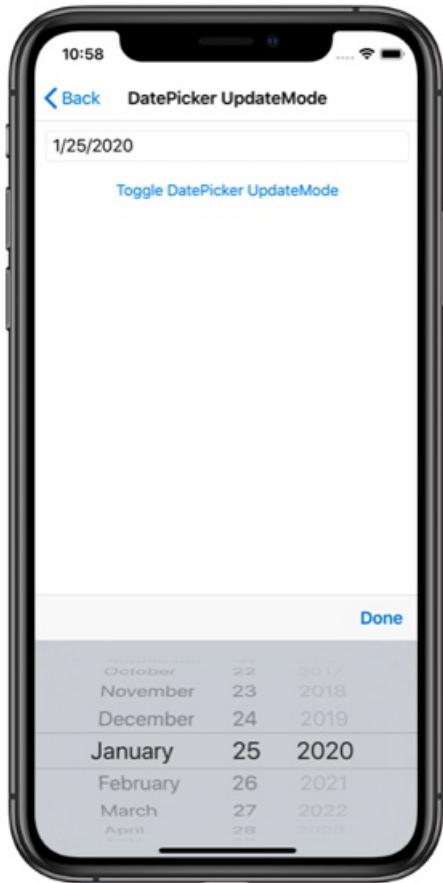
The `DatePicker.On<iOS>` method specifies that this platform-specific will only run on iOS. The `DatePicker.SetUpdateMode` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific` namespace, is used to control when item selection occurs, with the `UpdateMode` enumeration providing two possible values:

- `Immediately` – item selection occurs as the user browses items in the `DatePicker`. This is the default behavior.
- `WhenFinished` – item selection only occurs once the user has pressed the `Done` button in the `DatePicker`.

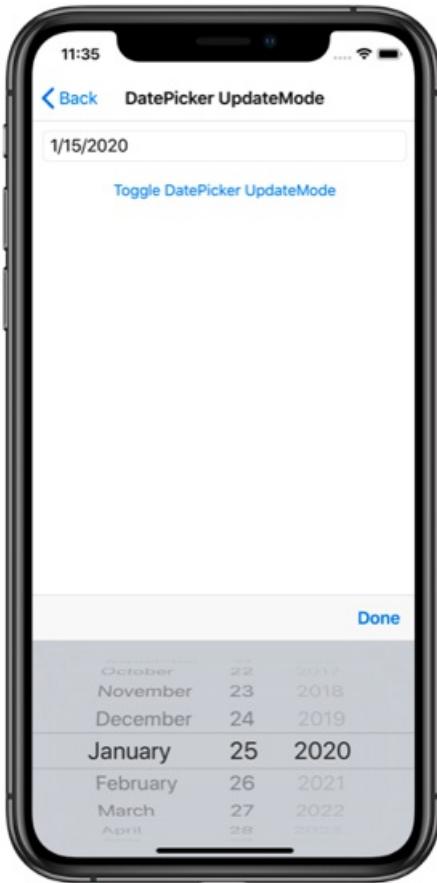
In addition, the `SetUpdateMode` method can be used to toggle the enumeration values by calling the `UpdateMode` method, which returns the current `UpdateMode`:

```
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

Entry Cursor Color on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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`:

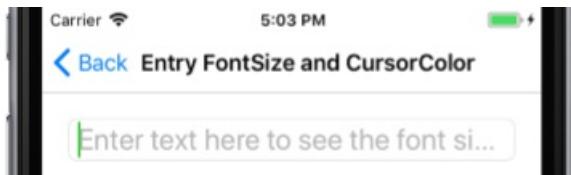
```
<ContentPage ...>
    <StackLayout>
        <Entry ... ios:Entry.CursorColor="LimeGreen" />
    </StackLayout>
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOS;
...
Entry entry = new Microsoft.Maui.Controls.Entry();
entry.On<iOS>().SetCursorColor(Colors.LimeGreen);
```

The `Entry.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Entry.SetCursorColor` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.iOS` 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`:



Entry font size on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"
    <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>
```

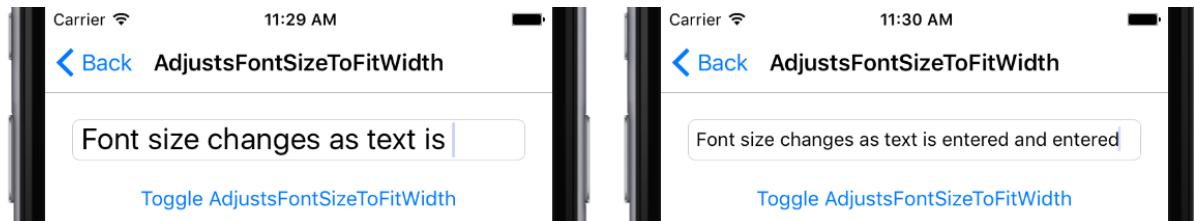
Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;
...
entry.On<iOS>().EnableAdjustsFontSizeToFitWidth();
```

The `Entry.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Entry.EnableAdjustsFontSizeToFitWidth` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific` namespace, is used to scale the font size of the inputted text to ensure that it fits in the `Entry`. In addition, the `Entry` class in the `Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific` namespace also has a `DisableAdjustsFontSizeToFitWidth` method that disables this platform-specific, and a `SetAdjustsFontSizeToFitWidth` method which can be used to toggle font size scaling by calling the `AdjustsFontSizeToFitWidth` method:

```
entry.On<iOS>().SetAdjustsFontSizeToFitWidth(!entry.On<iOS>().AdjustsFontSizeToFitWidth());
```

The result is that the font size of the `Entry` is scaled to ensure that the inputted text fits in the control:



FlyoutPage shadow on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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`:

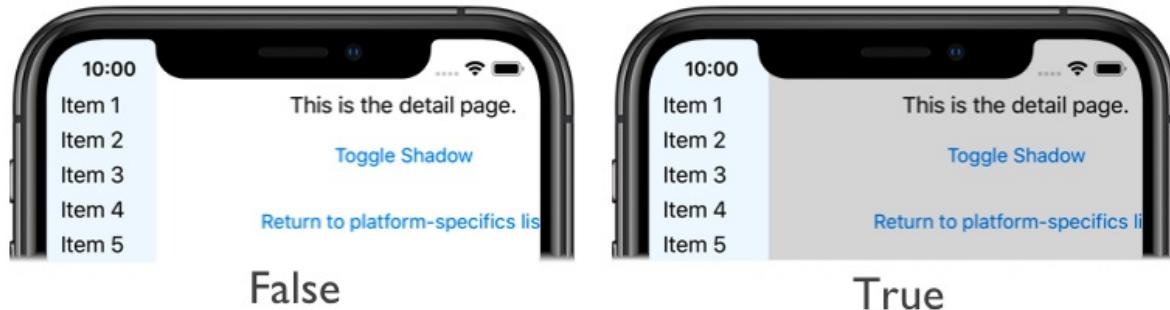
```
<FlyoutPage ...  
    xmlns:ios="clr-  
    namespace:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"  
    ios:FlyoutPage.ApplyShadow="true">  
    ...  
</FlyoutPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;  
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;  
...  
  
public class iOSFlyoutPageCode : FlyoutPage  
{  
    public iOSFlyoutPageCode()  
    {  
        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 `Microsoft.Maui.Controls.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:



Large Page Titles on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-
namespace:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"
    ...
    ios:NavigationPage.PrefersLargeTitles="true">
    ...
</NavigationPage>
```

Alternatively it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;
...

var navigationPage = new Microsoft.Maui.Controls.NavigationPage(new iOSLargeTitlePageCode());
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 `Microsoft.Maui.Controls.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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"
    Title="Large Title"
    ios:Page.LargeTitleDisplay="Never">
    ...
</ContentPage>
```

Alternatively, the page behavior can be overridden from C# using the fluent API:

```

using Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;
...
public class iOSLargeTitlePageCode : ContentPage
{
    public iOSLargeTitlePageCode()
    {
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Never);
    }
    ...
}

```

The `Page.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Page.SetLargeTitleDisplay` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific` namespace, controls the large title behavior on the `Page`, with the `LargeTitleDisplayMode` enumeration providing three possible values:

- `Always` – force the navigation bar and font size to use the large format.
- `Automatic` – use the same style (large or small) as the previous item in the navigation stack.
- `Never` – force the use of the regular, small format navigation bar.

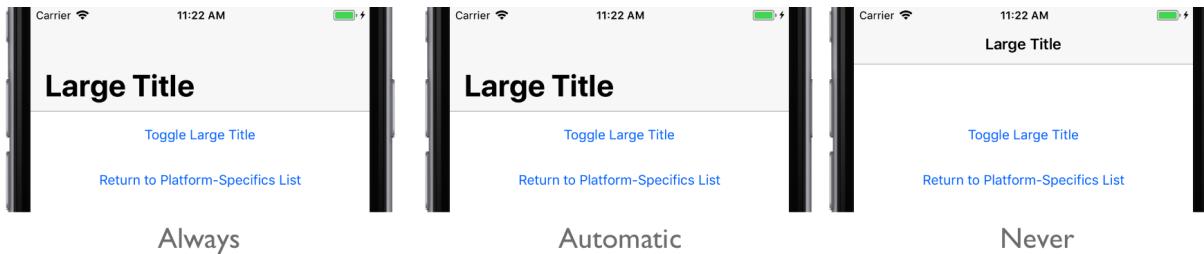
In addition, the `SetLargeTitleDisplay` method can be used to toggle the enumeration values by calling the `LargeTitleDisplay` method, which returns the current `LargeTitleDisplayMode`:

```

switch (On<iOS>().LargeTitleDisplay())
{
    case LargeTitleDisplayMode.Always:
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Automatic);
        break;
    case LargeTitleDisplayMode.Automatic:
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Never);
        break;
    case LargeTitleDisplayMode.Never:
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Always);
        break;
}

```

The result is that a specified `LargeTitleDisplayMode` is applied to the `Page`, which controls the large title behavior:



ListView group header style on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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 ...>
    <x:Bind xmlns:ios="clr-namespace:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">
        <StackLayout Margin="20">
            <ListView ... ios:ListView.GroupHeaderStyle="Grouped">
                ...
            </ListView>
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

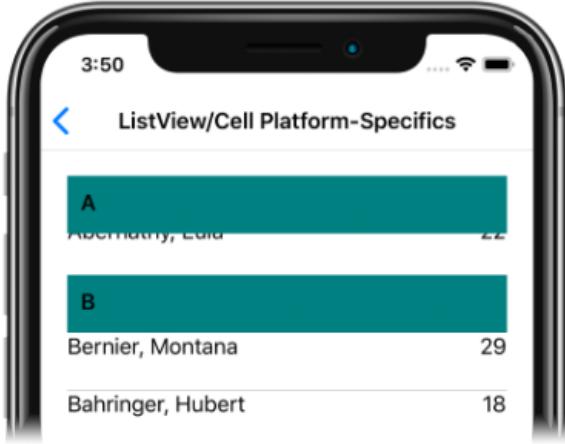
```
using Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;
...
listView.On<iOS>().SetGroupHeaderStyle(GroupHeaderStyle.Grouped);
```

The `ListView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `ListView.SetGroupHeaderStyle` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific` namespace, is used to control whether `ListView` header cells float during scrolling. The `GroupHeaderStyle` enumeration provides two possible values:

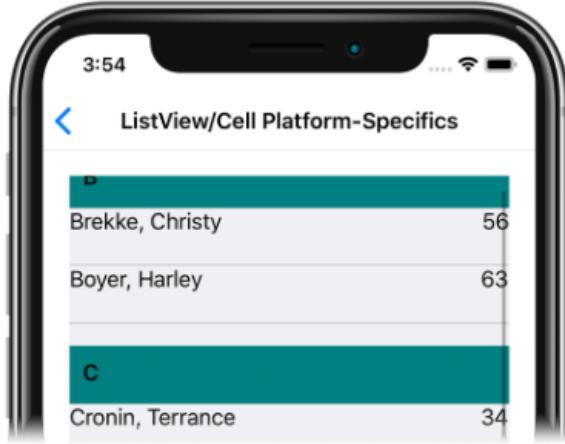
- `Plain` – indicates that header cells float when the `ListView` is scrolled (default).
- `Grouped` – indicates that header cells do not float when the `ListView` is scrolled.

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

ListView row animations on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">
        <StackLayout>
            <ListView ... ios:ListView.RowAnimationsEnabled="false">
                ...
            </ListView>
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.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 `Microsoft.Maui.Controls.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`.

ListView separator style on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">
        <StackLayout Margin="20">
            <ListView ... ios:ListView.SeparatorStyle="FullWidth">
                ...
            </ListView>
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.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 `Microsoft.Maui.Controls.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.
- `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.

Modal page presentation style on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"  
    ios:Page.ModalPresentationStyle="OverFullScreen">  
    ...  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

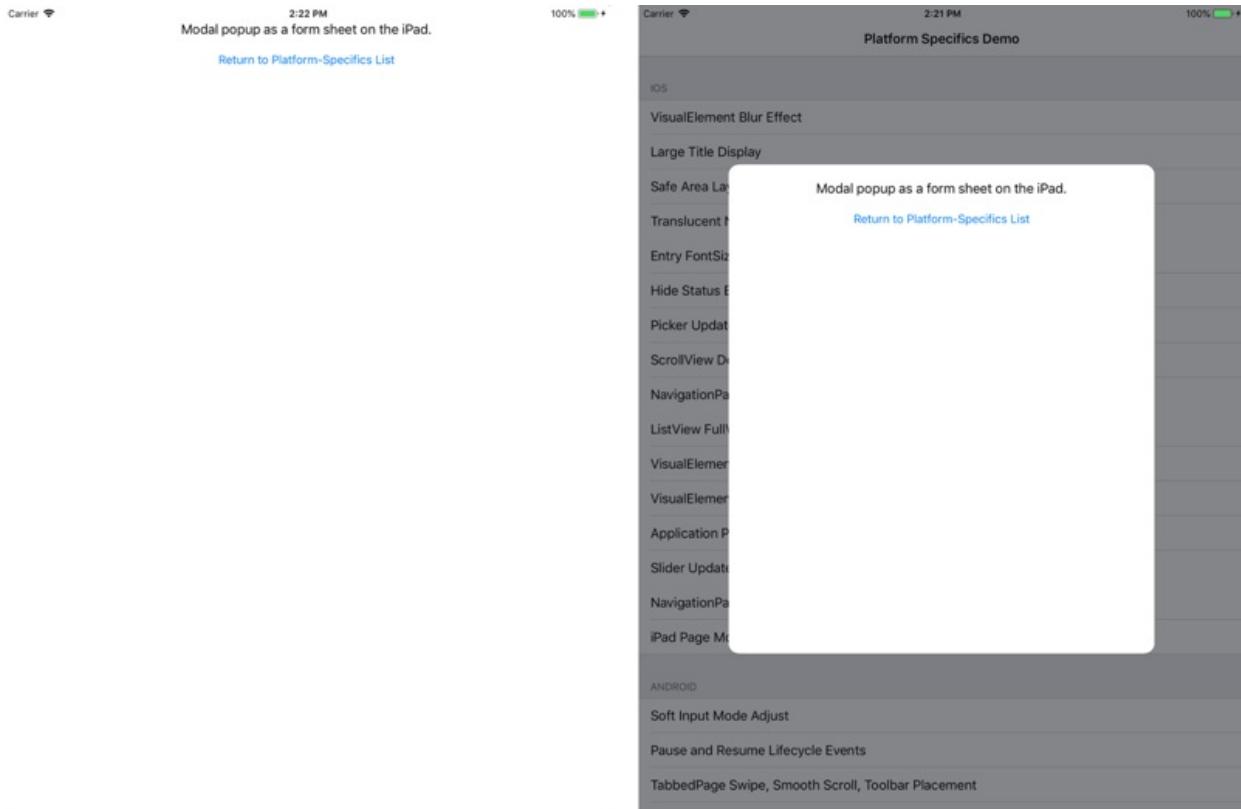
```
using Microsoft.Maui.Controls.PlatformConfiguration;  
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;  
...  
  
public class iOSModalFormSheetPageCode : ContentPage  
{  
    public iOSModalFormSheetPageCode()  
    {  
        On<iOS>().SetModalPresentationStyle(UIModalPresentationStyle.OverFullScreen);  
    }  
}
```

The `Page.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Page.SetModalPresentationStyle` method, in the `Microsoft.Maui.Controls.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 [Perform modal navigation](#).

NavigationPage bar translucency on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"  
    BackgroundColor="Blue"  
    ios:NavigationPage.IsNavigationBarTranslucent="true">  
    ...  
</NavigationPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;  
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;  
...  
  
(App.Current.MainPage as Microsoft.Maui.Controls.NavigationPage).BackgroundColor = Colors.Blue;  
(App.Current.MainPage as Microsoft.maui.Controls.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 `Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific` namespace, is used to make the navigation bar translucent. In addition, the `NavigationPage` class in the `Microsoft.Maui.Controls.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 Microsoft.Maui.Controls.NavigationPage)  
.On<iOS>()  
.SetIsNavigationBarTranslucent(!(App.Current.MainPage as Microsoft.Maui.Controls.NavigationPage).On<iOS>()  
.IsNavigationBarTranslucent());
```

The result is that the transparency of the navigation bar can be changed:



Picker item selection on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">
        <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 Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;
...
picker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
```

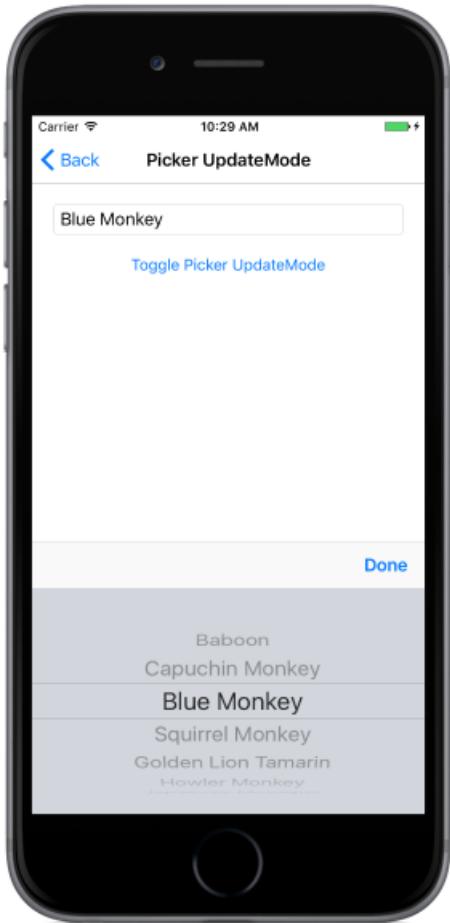
The `Picker.On<iOS>` method specifies that this platform-specific will only run on iOS. The `Picker.SetUpdateMode` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific` namespace, is used to control when item selection occurs, with the `UpdateMode` enumeration providing two possible values:

- `Immediately` – item selection occurs as the user browses items in the `Picker`. This is the default behavior in .NET MAUI.
- `WhenFinished` – item selection only occurs once the user has pressed the `Done` button in the `Picker`.

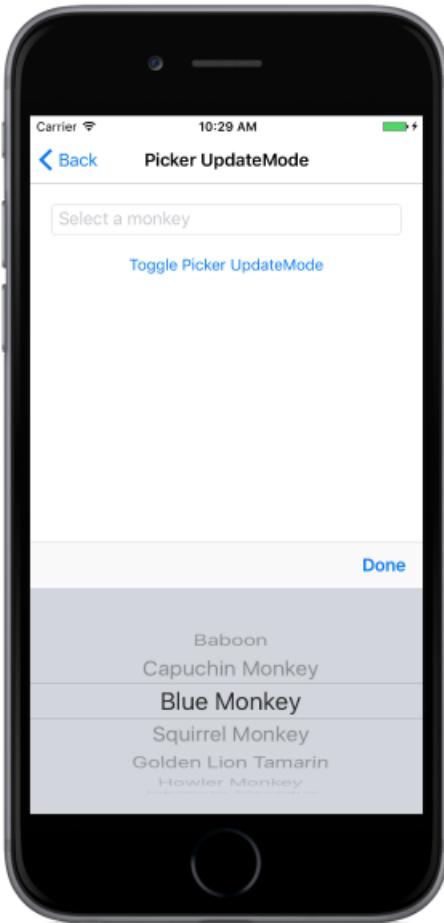
In addition, the `SetUpdateMode` method can be used to toggle the enumeration values by calling the `UpdateMode` method, which returns the current `UpdateMode`:

```
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

ScrollView content touches on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

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 .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">
        <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 Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.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 `Microsoft.Maui.Controls.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`:

Carrier

10:35 AM

100% 

[Toggle ScrollView DelayContentTouches](#)

ShouldDelayContentTouches = true

Carrier

10:36 AM

100% 

[Toggle ScrollView DelayContentTouches](#)

ShouldDelayContentTouches = false

SearchBar style on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">  
    <StackLayout>  
        <SearchBar ios:SearchBar.SearchBarStyle="Minimal"  
            Placeholder="Enter search term" />  
        ...  
    </StackLayout>  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

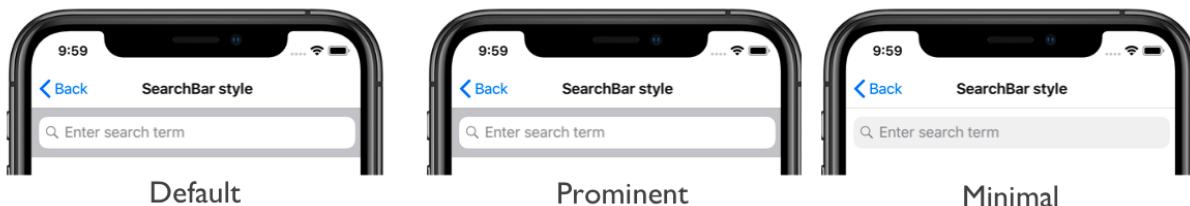
```
using Microsoft.Maui.Controls.PlatformConfiguration;  
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;  
...  
  
SearchBar searchBar = new SearchBar { Placeholder = "Enter search term" };  
searchBar.On<iOS>().SetSearchBarStyle(UISearchBarStyle.Minimal);
```

The `SearchBar.On<iOS>` method specifies that this platform-specific will only run on iOS. The `SearchBar.SetSearchBarStyle` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific` namespace, is used to control whether the `SearchBar` has a background. The `UISearchBarStyle` enumeration provides three possible values:

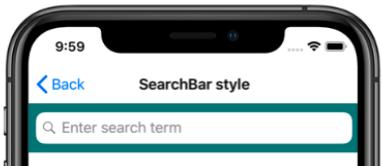
- `Default` indicates that the `SearchBar` has the default style. This is the default value of the `SearchBar.SearchBarStyle` bindable property.
- `Prominent` indicates that the `SearchBar` has a translucent background, and the search field is opaque.
- `Minimal` indicates that the `SearchBar` has no background, and the search field is translucent.

In addition, the `SearchBar.GetSearchBarStyle` method can be used to return the `UISearchBarStyle` that's applied to the `SearchBar`.

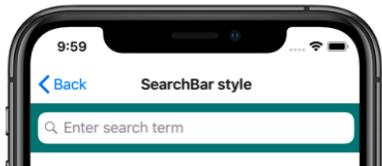
The result is that a specified `UISearchBarStyle` member is applied to a `SearchBar`, which controls whether the `SearchBar` has a background:



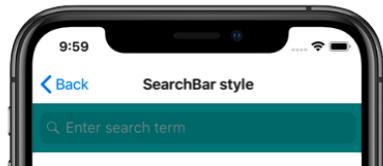
The following screenshot shows the `UISearchBarStyle` members applied to `SearchBar` objects that have their `BackgroundColor` property set:



Default



Prominent



Minimal

Simultaneous pan gesture recognition on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

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 .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"  
    ios:Application.PanGestureRecognizerShouldRecognizeSimultaneously="true">  
    ...  
</Application>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;  
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;  
...  
  
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 `Microsoft.Maui.Controls.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.

Slider thumb tap on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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`:

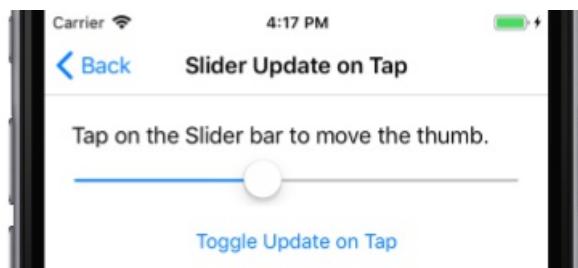
```
<ContentPage ...  
    xmlns:ios="clr-  
    namespace:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">  
    <StackLayout>  
        <Slider ... ios:Slider.UpdateOnTap="true" />  
        ...  
    </StackLayout>  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;  
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;  
...  
  
var slider = new Microsoft.Maui.Controls.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 `Microsoft.Maui.Controls.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:



SwipeView swipe transition mode on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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 ...>
    <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:

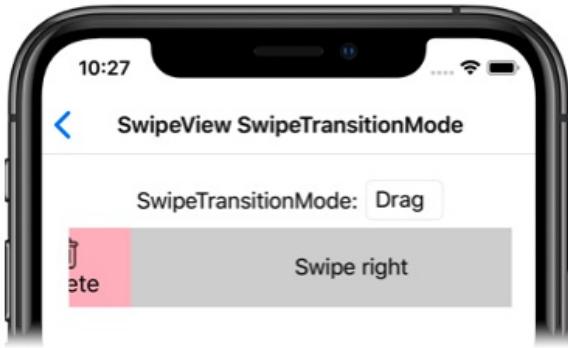
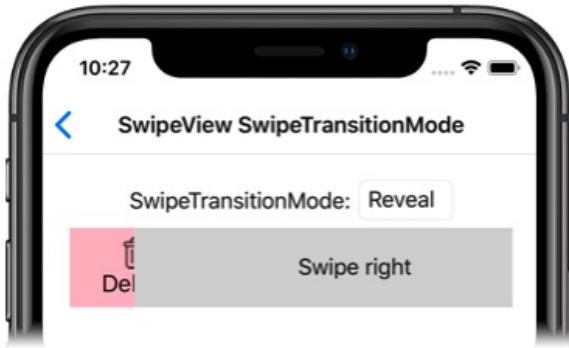
```
using Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;
...
var swipeView = new Microsoft.Maui.Controls.SwipeView();
swipeView.On<iOS>().SetSwipeTransitionMode(SwipeTransitionMode.Drag);
// ...
```

The `SwipeView.On<iOS>` method specifies that this platform-specific will only run on iOS. The `SwipeView.SetSwipeTransitionMode` method, in the `Microsoft.Maui.Controls.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`:



TabPage translucent tab bar on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"  
    ios:TabPage.TranslucencyMode="Opaque">  
    ...  
</TabPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;  
using Microsoft.Maui.Controls.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 `Microsoft.Maui.Controls.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:



TimePicker item selection on iOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) 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:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls">
    <StackLayout>
        <TimePicker Time="14:00:00"
            ios:TimePicker.UpdateMode="WhenFinished" />
        ...
    </StackLayout>
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration;
using Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;
...
timePicker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
```

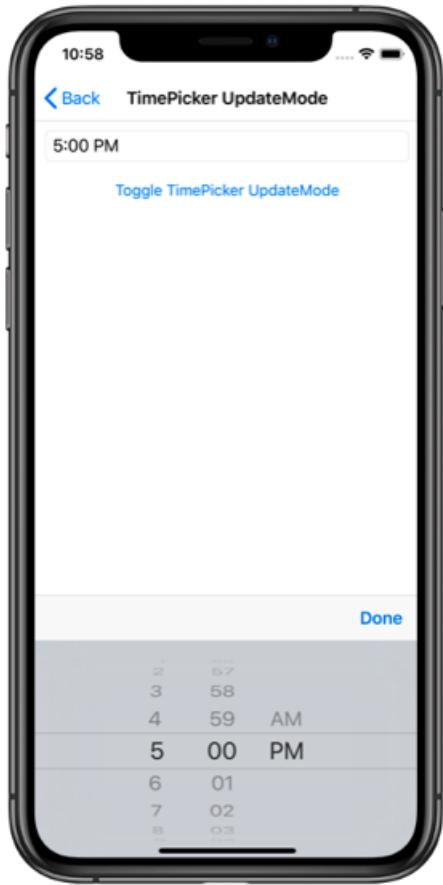
The `TimePicker.On<iOS>` method specifies that this platform-specific will only run on iOS. The `TimePicker.SetUpdateMode` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific` namespace, is used to control when item selection occurs, with the `UpdateMode` enumeration providing two possible values:

- `Immediately` – item selection occurs as the user browses items in the `TimePicker`. This is the default behavior.
- `WhenFinished` – item selection only occurs once the user has pressed the `Done` button in the `TimePicker`.

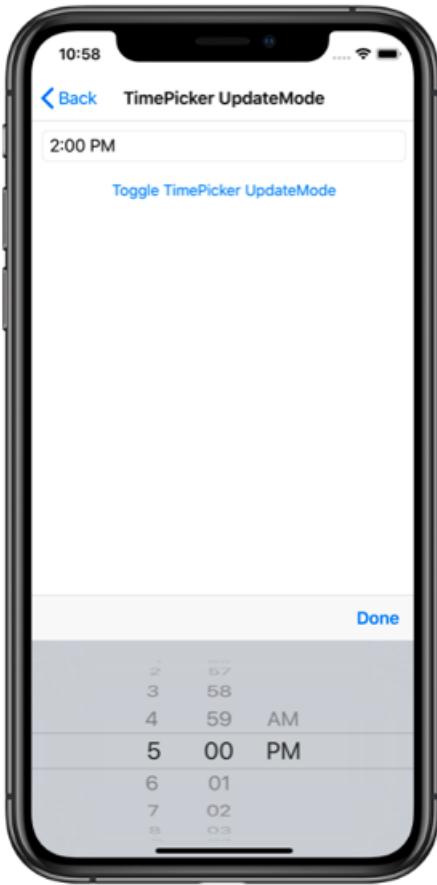
In addition, the `SetUpdateMode` method can be used to toggle the enumeration values by calling the `UpdateMode` method, which returns the current `updateMode`:

```
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:



UpdateTimeMode.Immediately



UpdateTimeMode.WhenFinished

Windows platform-specifics

9/20/2022 • 2 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) platform-specifics allow you to consume functionality that's only available on a specific platform, without customizing handlers.

The following platform-specific functionality is provided for .NET MAUI views, pages, and layouts on Windows:

- Setting an access key for a `VisualElement`. For more information, see [VisualElement Access Keys on Windows](#).

The following platform-specific functionality is provided for .NET MAUI views on Windows:

- 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 [ListViewSelectionMode 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](#).

The following platform-specific functionality is provided for the .NET MAUI `Application` class on Windows:

- Specifying the directory in the project that image assets will be loaded from. For more information, see [Default Image Directory on Windows](#).

Default image directory on Windows

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) Windows 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://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:windows="clr-
namespace:Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;assembly=Microsoft.Maui.Controls"
    ...
    windows:Application.ImageDirectory="Assets">
    ...
</Application>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;
...
Application.Current.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>().SetImageDirectory("Assets");
```

The `Application.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>` method specifies that this platform-specific will only run on Windows. The `Application.SetImageDirectory` method, in the `Microsoft.Maui.Controls.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 app image assets.

The result is that all images used in an app will be loaded from the specified project directory.

InputView reading order on Windows

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) Windows platform-specific enables the reading order (left-to-right or right-to-left) of bidirectional text in `Entry`, `Editor`, and `Label` objects to be detected dynamically. It's consumed in XAML by setting the `InputView.DetectReadingOrderFromContent` (for `Entry` and `Editor` objects) or `Label.DetectReadingOrderFromContent` attached property to a `boolean` value:

```
<ContentPage ...>
    xmlns:windows="clr-namespace:Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;assembly=Microsoft.Maui.Controls">
        <StackLayout>
            <Editor ... windows:InputView.DetectReadingOrderFromContent="true" />
            ...
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;
...
editor.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>().SetDetectReadingOrderFromContent(true);
```

The `Editor.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>` method specifies that this platform-specific will only run on Windows. The `InputView.SetDetectReadingOrderFromContent` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific` namespace, is used to control whether the reading order is detected from the content in the `InputView`. In addition, the `InputView.SetDetectReadingOrderFromContent` method can be used to toggle whether the reading order is detected from the content by calling the `InputView.GetDetectReadingOrderFromContent` method to return the current value:

```
editor.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>()
    .SetDetectReadingOrderFromContent(!editor.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>()
    .GetDetectReadingOrderFromContent());
```

The result is that `Entry`, `Editor`, and `Label` objects 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.

ListView SelectionMode on Windows

9/20/2022 • 2 minutes to read • [Edit Online](#)

On Windows, by default the .NET Multi-platform App UI (.NET MAUI) `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 .NET MAUI Windows 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:Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;assembly=Microsoft.Maui.Controls">  
    <StackLayout>  
        <ListView ... windows:ListView.SelectionMode="Inaccessible">  
            ...  
        </ListView>  
    </StackLayout>  
</ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

```
using Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;  
...  
  
listView.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>  
().SetSelectionMode(ListViewSelectionMode.Inaccessible);
```

The `ListView.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>` method specifies that this platform-specific will only run on Windows. The `ListView.SetSelectionMode` method, in the `Microsoft.Maui.Controls.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` objects on Windows.
- `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.

RefreshView pull direction on Windows

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) Windows 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:Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;assembly=Microsoft.Maui.Controls">
        <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:

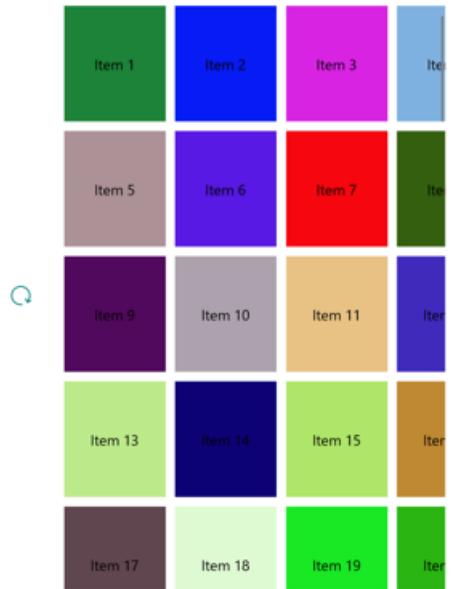
```
using Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;
...
refreshView.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>()
    .SetRefreshPullDirection(RefreshPullDirection.LeftToRight);
```

The `RefreshView.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>` method specifies that this platform-specific will only run on Windows. The `RefreshView.SetRefreshPullDirection` method, in the `Microsoft.Maui.Controls.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.

SearchBar spell check on Windows

9/20/2022 • 2 minutes to read • [Edit Online](#)

This .NET Multi-platform App UI (.NET MAUI) Windows 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:Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;assembly=Microsoft.Maui.Controls">
        <StackLayout>
            <SearchBar ... windows:SearchBar.IsSpellCheckEnabled="true" />
            ...
        </StackLayout>
    </ContentPage>
```

Alternatively, it can be consumed from C# using the fluent API:

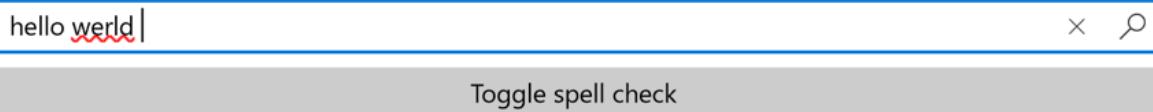
```
using Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;
...
searchBar.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>().SetIsSpellCheckEnabled(true);
```

The `SearchBar.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>` method specifies that this platform-specific will only run on Windows. The `SearchBar.SetIsSpellCheckEnabled` method, in the `Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific` namespace, turns the spell checker on and off. In addition, the `SearchBar.GetIsSpellCheckEnabled` 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<Microsoft.Maui.Controls.PlatformConfiguration.Windows>()
    .SetIsSpellCheckEnabled(!searchBar.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>()
    .GetIsSpellCheckEnabled());
```

The result is that text entered into the `SearchBar` can be spell checked, with incorrect spellings being indicated to the user:

earchBar Spell Check



NOTE

The `SearchBar` class in the `Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific` namespace also has `EnableSpellCheck` and `DisableSpellCheck` methods that can be used to enable and disable the spell checker on the `SearchBar`, respectively.

VisualElement access keys on Windows

9/20/2022 • 3 minutes to read • [Edit Online](#)

Access keys are keyboard shortcuts that improve the usability and accessibility of apps on Windows 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 .NET Multi-platform App UI (.NET MAUI) Windows platform-specific is used to specify an access key for a `VisualElement`. It's consumed in XAML by setting the `windows:VisualElement.AccessKey` attached property to an alphanumeric value, and by optionally setting the `windows:VisualElement.AccessKeyPlacement` attached property to a value of the `AccessKeyPlacement` enumeration, the `windows:VisualElement.AccessKeyHorizontalOffset` attached property to a `double`, and the `windows:VisualElement.AccessKeyVerticalOffset` attached property to a `double`:

```
<TabbedPage ...>
    xmlns:windows="clr-namespace:Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;assembly=Microsoft.Maui.Controls">
        <ContentPage Title="Page 1">
            windows:VisualElement.AccessKey="1"
            <StackLayout>
                ...
                <Switch windows:VisualElement.AccessKey="A" />
                <Entry Placeholder="Enter text here">
                    windows:VisualElement.AccessKey="B" />
                ...
                <Button Text="Access key F, placement top with offsets">
                    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 Microsoft.Maui.Controls.PlatformConfiguration.WindowsSpecific;
...
var page = new ContentPage { Title = "Page 1" };
page.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>().SetAccessKey("1");

var switchView = new Switch();
switchView.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>().SetAccessKey("A");
var entry = new Entry { Placeholder = "Enter text here" };
entry.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>().SetAccessKey("B");
...
var button4 = new Button { Text = "Access key F, placement top with offsets" };
button4.Clicked += OnButtonClicked;
button4.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>()
    .SetAccessKey("F")
    .SetAccessKeyPlacement(AccessKeyPlacement.Top)
    .SetAccessKeyHorizontalOffset(20)
    .SetAccessKeyVerticalOffset(20);
...

```

The `visualElement.On<Microsoft.Maui.Controls.PlatformConfiguration.Windows>` method specifies that this platform-specific will only run on Windows. The `VisualElement.SetAccessKey` method, in the `Microsoft.Maui.Controls.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:

Press the alt key once to see the access keys, then press the alt key and an access key.

A  Off

B

C Access key C

D Access key D, placement left

E Access key E, placement right

F Access key F, placement top with offsets

G Return to Platform-Specifics List

When a user activates an access key, by pressing the Alt key followed by the access key, the default action for the `VisualElement` will be executed. For example, when a user activates the access key on a `Switch`, the `Switch` is toggled. When a user activates the access key on an `Entry`, the `Entry` gains focus. When a user activates the access key on a `Button`, the event handler for the `Clicked` event is executed.

For more information about access keys, see [Access keys](#).

Clipboard

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IClipboard` interface. With this interface, you can copy and paste text to and from the system clipboard.

The default implementation of the `IClipboard` interface is available through the `Clipboard.Default` property. Both the `IClipboard` interface and `Clipboard` class are contained in the `Microsoft.Maui.ApplicationModel.DataTransfer` namespace.

TIP

Access to the clipboard must be done on the main user interface thread. For more information on how to invoke methods on the main user interface thread, see [MainThread](#).

Using Clipboard

Access to the clipboard is limited to string data. You can check if the clipboard contains data, set or clear the data, and read the data. The `ClipboardContentChanged` event is raised whenever the clipboard data changes.

The following code example demonstrates using a button to set the clipboard data:

```
private async void SetClipboardButton_Clicked(object sender, EventArgs e) =>
    await Clipboard.Default.SetTextAsync("This text was highlighted in the UI.");
```

The following code example demonstrates using a button to read the clipboard data. The code first checks if the clipboard has data, read that data, and then uses a `null` value with `SetTextAsync` to clear the clipboard:

```
private async void ReadClipboardButton_Clicked(object sender, EventArgs e)
{
    if (Clipboard.Default.HasText)
    {
        ClipboardOutputLabel.Text = await Clipboard.Default.GetTextAsync();
        await ClearClipboard();
    }
    else
        ClipboardOutputLabel.Text = "Clipboard is empty";
}

private async Task ClearClipboard() =>
    await Clipboard.Default.SetTextAsync(null);
```

Clear the clipboard

You can clear the clipboard by passing `null` to the `SetTextAsync` method, as the following code example demonstrates:

```
private async Task ClearClipboard() =>
    await Clipboard.Default.SetTextAsync(null);
```

Detecting clipboard changes

The `IClipboard` interface provides the `ClipboardContentChanged` event. When this event is raised, the clipboard content has changed. The following code example adds a handler to the event when the content page is loaded:

```
private void ContentPage_Loaded(object sender, EventArgs e)
{
    Clipboard.Default.ClipboardContentChanged += Clipboard_ClipboardContentChanged;
}

private async void Clipboard_ClipboardContentChanged(object sender, EventArgs e)
{
    ClipboardOutputLabel.Text = await Clipboard.Default.GetTextAsync();
}
```

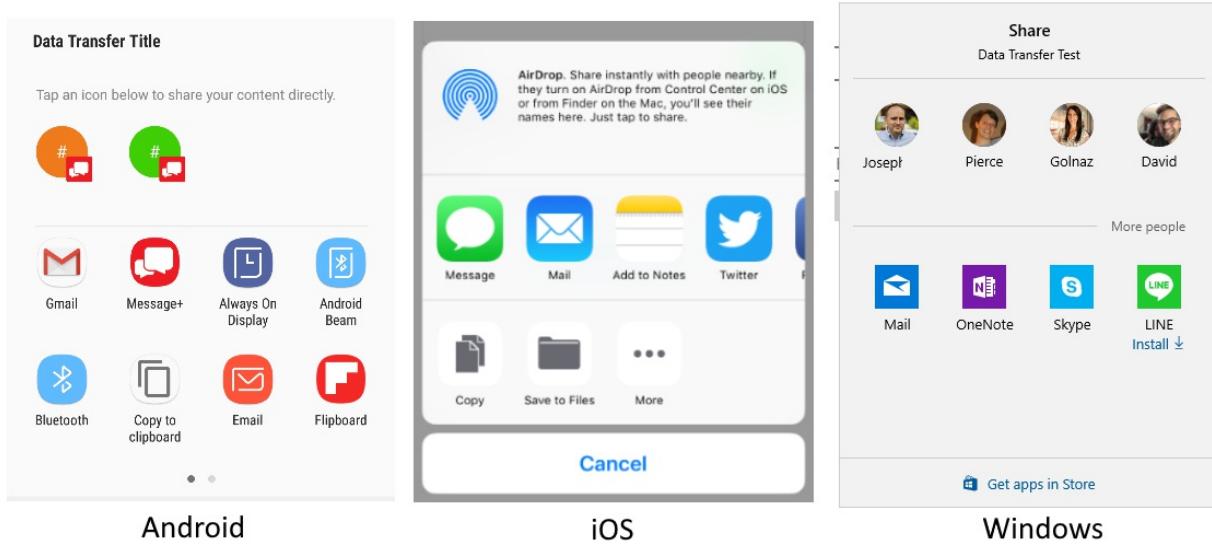
Share

9/20/2022 • 3 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IShare` interface. This interface provides an API to send data, such as text or web links, to the devices share function.

The default implementation of the `IShare` interface is available through the `Share.Default` property. Both the `IShare` interface and `Share` class are contained in the `Microsoft.Maui.ApplicationModel.DataTransfer` namespace.

When a share request is made, the device displays a share window, prompting the user to choose an app to share with:



Get started

To access the **Share** functionality, the following platform-specific setup is required:

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

No setup is required.

Share text and links

The share functionality works by calling the `RequestAsync` method with a data 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 async Task ShareText(string text)
{
    await Share.Default.RequestAsync(new ShareTextRequest
    {
        Text = text,
        Title = "Share Text"
    });
}

public async Task ShareUri(string uri, IShare share)
{
    await share.RequestAsync(new ShareTextRequest
    {
        Uri = uri,
        Title = "Share Web Link"
    });
}

```

Share a file

You can also share files to other applications on the device. .NET MAUI automatically detects the file type (MIME) and requests a share. However, operating systems may restrict which types of files can be shared.

The following code example writes a text file to the device, and then requests a share:

```

public async Task ShareFile()
{
    string fn = "Attachment.txt";
    string file = Path.Combine(FileSystem.CacheDirectory, fn);

    File.WriteAllText(file, "Hello World");

    await Share.Default.RequestAsync(new ShareFileRequest
    {
        Title = "Share text file",
        File = new ShareFile(file)
    });
}

```

Share multiple files

Sharing multiple files is slightly different from sharing a single file. Instead of using the `File` property of the share request, use the `Files` property:

```

public async Task ShareMultipleFiles()
{
    string file1 = Path.Combine(FileSystem.CacheDirectory, "Attachment1.txt");
    string file2 = Path.Combine(FileSystem.CacheDirectory, "Attachment2.txt");

    File.WriteAllText(file1, "Content 1");
    File.WriteAllText(file2, "Content 2");

    await Share.Default.RequestAsync(new ShareMultipleFilesRequest
    {
        Title = "Share multiple files",
        Files = new List<ShareFile> { new ShareFile(file1), new ShareFile(file2) }
    });
}

```

Presentation location

When requesting a share or opening launcher on iPadOS, you can present it in a popover. This specifies where the popover 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`.

Here are some extension methods that help calculate the bounds of a view:

```
public static class ViewHelpers
{
    public static Rectangle GetAbsoluteBounds(this Microsoft.Maui.Controls.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 Microsoft.Maui.Controls.View v)
            {
                absoluteX += v.X + v.Margin.Top;
                absoluteY += v.Y + v.Margin.Left;
            }
        }

        return new Rectangle(absoluteX, absoluteY, element.Width, element.Height);
    }
}
```

This can then be used when calling `RequestAsync`:

```
public Command<Microsoft.Maui.Controls.View> ShareCommand { get; } = new  
Command<Microsoft.Maui.Controls.View>(Share);  
  
async void Share(Microsoft.Maui.Controls.View element)  
{  
    try  
    {  
        await Share.Default.RequestAsync(new ShareTextRequest  
        {  
            PresentationSourceBounds = element.GetAbsoluteBounds(),  
            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}}"/>
```

For an example of the `ViewHelpers` class, see the [.NET MAUI Sample hosted on GitHub](#).

Platform differences

This section describes the platform-specific differences with the share API.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

- The `Subject` property is used for the desired subject of a message.

File picker

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IFilePicker` interface. With the `IFilePicker` interface, you can prompt the user to pick one or more files from the device.

The default implementation of the `IFilePicker` interface is available through the `FilePicker.Default` property. Both the `IFilePicker` interface and `FilePicker` class are contained in the `Microsoft.Maui.Storage` namespace.

Get started

To access the `FilePicker` functionality, the following platform specific setup is required.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

The `ReadExternalStorage` permission is required and must be configured in the Android project. This can be added in the following ways:

- Add the assembly-based permission:

Open the `Platforms/Android/MainApplication.cs` file and add the following assembly attributes after `using` directives:

```
[assembly: UsesPermission(Android.Manifest.Permission.ReadExternalStorage)]
```

- or -

- Update the Android Manifest:

Open the `Platforms/Android/AndroidManifest.xml` file and add the following in the `manifest` node:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

IMPORTANT

All methods must be called on the UI thread because permission checks and requests are automatically handled by .NET MAUI.

Pick a file

The `PickAsync` method prompts the user to pick a file from the device. Use the `PickOptions` type to specify the title and file types allowed with the picker. The following example demonstrates opening the picker and processing the selected image:

```

public async Task<FileResult> PickAndShow(PickOptions options)
{
    try
    {
        var result = await FilePicker.Default.PickAsync(options);
        if (result != null)
        {
            if (result.FileName.EndsWith("jpg", StringComparison.OrdinalIgnoreCase) ||
                result.FileName.EndsWith("png", StringComparison.OrdinalIgnoreCase))
            {
                using var stream = await result.OpenReadAsync();
                var 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 file types per platform, by creating an instance of the `FilePickerFileType` class. The constructor of this class takes a dictionary that is keyed by the `DevicePlatform` type to identify the platform. The value of the dictionary key is a collection of strings representing the file types. For example here's how you would specify specific comic file types:

```

var customFileType = new FilePickerFileType(
    new Dictionary<DevicePlatform, IEnumerable<string>>
    {
        { DevicePlatform.iOS, new[] { "public.my.comic.extension" } }, // or general UTType
values
        { DevicePlatform.Android, new[] { "application/comics" } },
        { DevicePlatform.WinUI, new[] { ".cbr", ".cbz" } },
        { DevicePlatform.Tizen, new[] { "*/*" } },
        { DevicePlatform.macOS, new[] { "cbr", "cbz" } }, // or general UTType values
    });

```

```

PickOptions options = new()
{
    PickerTitle = "Please select a comic file",
    FileTypes = customFileType,
};

```

Pick multiple files

If you want the user to pick multiple files, call the `FilePicker.PickMultipleAsync` method. This method also takes a `PickOptions` parameter to specify additional information. The results are the same as `PickAsync`, but instead of the `FileResult` type returned, an `IEnumerable<FileResult>` type is returned with all of the selected files.

TIP

The `FullPath` property doesn't always return the physical path to the file. To get the file, use the `OpenReadAsync` method.

Platform differences

This section describes the platform-specific differences with the file picker.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

The `PickOptions.PickerTitle` is displayed on the initial prompt to the user, but not in the picker dialog itself.

File system helpers

9/20/2022 • 2 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IFileSystem` interface. This interface provides helper methods that access the app's cache and data directories, and helps access files in the app package.

The default implementation of the `IFileSystem` interface is available through the `FileSystem.Current` property. Both the `IFileSystem` interface and `FileSystem` class are contained in the `Microsoft.Maui.Storage` namespace.

Using file system helpers

Each operating system will have unique paths to the app cache and app data directories. The `IFileSystem` interface provides a cross-platform API for accessing these directory paths.

Cache directory

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 shouldn't be data that is required to operate the app, as the operating system may clear this storage.

```
string cacheDir = FileSystem.Current.CacheDirectory;
```

App data directory

To get the app's top-level directory for any files that aren't user data files. These files are backed up with the operating system syncing framework.

```
string mainDir = FileSystem.Current.AppDataDirectory;
```

Bundled files

To open a file that is bundled into the app package, use the `OpenAppPackageFileAsync` method and pass the file name. This method returns a read-only `Stream` representing the file contents. The following example demonstrates using a method to read the text contents of a file:

```
public async Task<string> ReadTextFile(string filePath)
{
    using Stream fileStream = await FileSystem.Current.OpenAppPackageFileAsync(filePath);
    using StreamReader reader = new StreamReader(fileStream);

    return await reader.ReadToEndAsync();
}
```

Writing from a bundled file to the app data folder

You can't modify an app's bundled file. But you can read it first, then write it back to the **cache directory** or **app data directory**. The following example uses `OpenAppPackageFileAsync` to read a bundled file, alters it, and then writes it to the app data folder:

```
public async Task ConvertFileToUpperCase(string sourceFile, string targetFileName)
{
    // Read the source file
    using Stream fileStream = await FileSystem.Current.OpenAppPackageFileAsync(sourceFile);
    using StreamReader reader = new StreamReader(fileStream);

    string content = await reader.ReadToEndAsync();

    // Transform file content to upper case text
    content = content.ToUpperInvariant();

    // Write the file content to the app data directory
    string targetFile = System.IO.Path.Combine(FileSystem.Current.AppDataDirectory, targetFileName);

    using FileStream outputStream = System.IO.File.OpenWrite(targetFile);
    using StreamWriter streamWriter = new StreamWriter(outputStream);

    await streamWriter.WriteAsync(content);
}
```

Platform differences

This section describes the platform-specific differences with the file system helpers.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)
- [FileSystem.CacheDirectory](#)
Returns the [CacheDir](#) of the current context.
- [FileSystem.AppDataDirectory](#)
Returns the [FilesDir](#) of the current context, which are backed up using [Auto Backup](#) starting on API 23 and above.
- [FileSystem.OpenAppPackageFileAsync](#)
Files that were added to the project with the [Build Action](#) of **MauiAsset** can be opened with this method. .NET MAUI projects will process any file in the *Resources\Raw* folder as a **MauiAsset**.

Preferences

9/20/2022 • 3 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `IPreferences` interface. This interface helps store app preferences in a key/value store.

The default implementation of the `IPreferences` interface is available through the `Preferences.Default` property. Both the `IPreferences` interface and `Preferences` class are contained in the `Microsoft.Maui.Storage` namespace.

Storage types

Preferences are stored with a `String` key. The value of a preference must be one of the following data types:

- `Boolean`
- `Double`
- `Int32`
- `Single`
- `Int64`
- `String`
- `DateTime`

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.
- 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 isn't a Coordinated Universal Time (UTC) value.

Set preferences

Preferences are set by calling the `Preferences.Set` method, providing the key and value:

```
// Set a string value:  
Preferences.Default.Set("first_name", "John");  
  
// Set an numerical value:  
Preferences.Default.Set("age", 28);  
  
// Set a boolean value:  
Preferences.Default.Set("has_pets", true);
```

Get preferences

To retrieve a value from preferences, you pass the key of the preference, followed by the default value when the key doesn't exist:

```
string firstName = Preferences.Default.Get("first_name", "Unknown");
int age = Preferences.Default.Get("age", -1);
bool hasPets = Preferences.Default.Get("has_pets", false);
```

Check for a key

It may be useful to check if a key exists in the preferences or not. Even though `Get` has you set a default value when the key doesn't exist, there may be cases where the key existed, but the value of the key matched the default value. So you can't rely on the default value as an indicator that the key doesn't exist. Use the `ContainsKey` method to determine if a key exists:

```
bool hasKey = Preferences.Default.ContainsKey("my_key");
```

Remove one or all keys

Use the `Remove` method to remove a specific key from preferences:

```
Preferences.Default.Remove("first_name");
```

To remove all keys, use the `Clear` method:

```
Preferences.Default.Clear();
```

Shared keys

The preferences stored by your app are only visible to your app. However, you can also create a **shared** preference that can be used by other extensions or a watch app. When you set, remove, or retrieve a preference, an optional string parameter can be supplied to specify the name of the container the preference is stored in.

The following methods take a string parameter named `sharedName`:

- `Preferences.Set`
- `Preferences.Get`
- `Preferences.Remove`
- `Preferences.Clear`

IMPORTANT

Please read the platform implementation specifics, as shared preferences have behavior-specific implementations

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](#)
- Android: [Getting Started with Settings Screens](#)

Platform differences

This section describes the platform-specific differences with the preferences API.

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

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 causes all *preferences* to be removed, except when the app runs on Android 6.0 (API level 23) or later, while using the [Auto Backup](#) feature. This feature is on by default and preserves app data, including **Shared Preferences**, which is what the **Preferences** API uses. You can disable this by following Google's [Auto Backup documentation](#).

Limitations

Performance may be impacted if you store large amounts of text, as the API was designed to store small amounts of text.

Secure storage

9/20/2022 • 3 minutes to read • [Edit Online](#)

This article describes how you can use the .NET Multi-platform App UI (.NET MAUI) `ISecureStorage` interface. This interface helps securely store simple key/value pairs.

The default implementation of the `ISecureStorage` interface is available through the `SecureStorage.Default` property. Both the `ISecureStorage` interface and `SecureStorage` class are contained in the `Microsoft.Maui.Storage` namespace.

Get started

To access the `SecureStorage` functionality, the following platform-specific setup is required:

- [Android](#)
- [iOS\macOS](#)
- [Windows](#)

[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 reinstalled or installed on a new device. This can affect `SecureStorage`, which utilizes share preferences that are backed up and can't be decrypted when the restore occurs. .NET MAUI automatically handles this case by removing the key so it can be reset. Alternatively, you can disable Auto Backup.

Enable or disable backup

You can choose to disable Auto Backup for your entire application by setting `android:allowBackup` 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 `SecureStorage` 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`. 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}.mauiessential.xml"/>
</full-backup-content>
```

Use secure storage

The following code examples demonstrate how to use secure storage.

TIP

It's 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's best to handle this by removing and adding the setting back if possible.

Write a value

To save a value for a given *key* in secure storage:

```
await SecureStorage.Default.SetAsync("oauth_token", "secret-oauth-token-value");
```

Read a value

To retrieve a value from secure storage:

```
string oauthToken = await SecureStorage.Default.GetAsync("oauth_token");

if (oauthToken == null)
{
    // No value is associated with the key "oauth_token"
}
```

TIP

If there isn't a value associated with the key, `GetAsync` returns `null`.

Remove a value

To remove a specific value, remove the key:

```
bool success = SecureStorage.Default.Remove("oauth_token");
```

To remove all values, use the `RemoveAll` method:

```
SecureStorage.Default.RemoveAll();
```

Platform differences

This section describes the platform-specific differences with the secure storage API.

- [Android](#)
- [iOS\macOS](#)

- [Windows](#)

`SecureStorage` uses the [Preferences API](#) and follows the same data persistence outlined in the [Preferences documentation](#), with a filename of `[YOUR-APP-PACKAGE-ID].microsoft.maui.essentials.preferences`. However, data is encrypted with the Android `EncryptedSharedPreferences` class, from the Android Security library, which wraps the `SharedPreferences` class and automatically encrypts keys and values using a two-scheme approach:

- Keys are deterministically encrypted, so that the key can be encrypted and properly looked up.
- Values are non-deterministically encrypted using AES-256 GCM.

For more information about the Android Security library, see [Work with data more securely](#) on developer.android.com.

Limitations

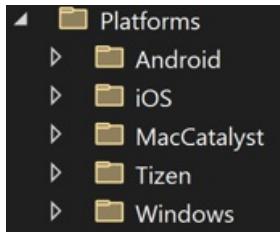
Performance may be impacted if you store large amounts of text, as the API was designed to store small amounts of text.

Configure multi-targeting

9/20/2022 • 5 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) apps use multi-targeting to target multiple platforms from a single project.

The project for a .NET MAUI app contains a *Platforms* folder, with each child folder representing a platform that .NET MAUI can target:



The folders for each target platform contain platform-specific code that starts the app on each platform, plus any additional platform code you add. At build time, the build system only includes the code from each folder when building for that specific platform. For example, when you build for Android the files in the *Platforms > Android* folder will be built into the app package, but the files in the other *Platforms* folders won't be.

In addition to this default multi-targeting approach, .NET MAUI apps can also be multi-targeted based on your own filename and folder criteria. This enables you to structure your .NET MAUI app project so that you don't have to place your platform code into sub-folders of the *Platforms* folder.

Configure filename-based multi-targeting

A standard multi-targeting pattern is to include the platform as an extension in the filename for the platform code. For example, *MyService.Android.cs* would represent an Android-specific implementation of the `MyService` class. The build system can be configured to use this pattern by adding the following XML to your .NET MAUI app project (.csproj) file as children of the `<Project>` node:

```

<!-- Android -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-android')) != true">
  <Compile Remove="**\**\*.Android.cs" />
  <None Include="**\**\*.Android.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Both iOS and Mac Catalyst -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-ios')) != true AND
$(TargetFramework.StartsWith('net6.0-maccatalyst')) != true">
  <Compile Remove="**\**\*.MaciOS.cs" />
  <None Include="**\**\*.MaciOS.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- iOS -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-ios')) != true">
  <Compile Remove="**\**\*.iOS.cs" />
  <None Include="**\**\*.iOS.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Mac Catalyst -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-maccatalyst')) != true">
  <Compile Remove="**\**\*.MacCatalyst.cs" />
  <None Include="**\**\*.MacCatalyst.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Windows -->
<ItemGroup Condition="$(TargetFramework.Contains('-windows')) != true">
  <Compile Remove="**\*.Windows.cs" />
  <None Include="**\*.Windows.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

```

This XML configures the build system to remove platform-based filename patterns under specific conditions:

- Don't compile C# code whose filename ends with *.Android.cs*, if you aren't building for Android.
- Don't compile C# code whose filename ends with *.MaciOS.cs*, if you aren't building for iOS and Mac Catalyst.
- Don't compile C# code whose filename ends with *.iOS.cs*, if you aren't building for iOS.
- Don't compile C# code whose filename ends with *.MacCatalyst.cs*, if you aren't building for Mac Catalyst.
- Don't compile C# code whose filename ends with *.Windows.cs*, if you aren't building for Windows.

IMPORTANT

Filename-based multi-targeting can be combined with folder-based multi-targeting. For more information, see [Combine filename and folder multi-targeting](#).

Configure folder-based multi-targeting

Another standard multi-targeting pattern is to include the platform as a folder name. For example, a folder named *Android* would contain Android-specific code. The build system can be configured to use this pattern by adding the following XML to your .NET MAUI app project (.csproj) file as children of the `<Project>` node:

```

<!-- Android -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-android')) != true">
  <Compile Remove="**\Android\**\*.cs" />
  <None Include="**\Android\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Both iOS and Mac Catalyst -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-ios')) != true AND
$(TargetFramework.StartsWith('net6.0-maccatalyst')) != true">
  <Compile Remove="**\MaciOS\**\*.cs" />
  <None Include="**\MaciOS\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- iOS -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-ios')) != true">
  <Compile Remove="**\iOS\**\*.cs" />
  <None Include="**\iOS\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Mac Catalyst -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-maccatalyst')) != true">
  <Compile Remove="**\MacCatalyst\**\*.cs" />
  <None Include="**\MacCatalyst\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Windows -->
<ItemGroup Condition="$(TargetFramework.Contains('-windows')) != true">
  <Compile Remove="**\Windows\**\*.cs" />
  <None Include="**\Windows\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

```

This XML configures the build system to remove platform-based folder patterns under specific conditions:

- Don't compile C# code that's located in the *Android* folder, or sub-folder of the *Android* folder, if you aren't building for Android.
- Don't compile C# code that's located in the *MacOS* folder, or sub-folder of the *MacOS* folder, if you aren't building for iOS and Mac Catalyst.
- Don't compile C# code that's located in the *iOS* folder, or sub-folder of the *iOS* folder, if you aren't building for iOS.
- Don't compile C# code that's located in the *MacCatalyst* folder, or sub-folder of the *MacCatalyst* folder, if you aren't building for Mac Catalyst.
- Don't compile C# code that's located in the *Windows* folder, or sub-folder of the *Windows* folder, if you aren't building for Windows.

IMPORTANT

Folder-based multi-targeting can be combined with filename-based multi-targeting. For more information, see [Combine filename and folder multi-targeting](#).

Combine filename and folder multi-targeting

Filename-based multi-targeting can be combined with folder-based multi-targeting if required. The build system can be configured to use this pattern by adding the following XML to your .NET MAUI app project (.csproj) file as children of the `<Project>` node:

```

<!-- Android -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-android')) != true">
  <Compile Remove="**\**\*.Android.cs" />
  <None Include="**\**\*.Android.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
  <Compile Remove="**\Android\**\*.cs" />
  <None Include="**\Android\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Both iOS and Mac Catalyst -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-ios')) != true AND
$(TargetFramework.StartsWith('net6.0-maccatalyst')) != true">
  <Compile Remove="**\**\*.MaciOS.cs" />
  <None Include="**\**\*.MaciOS.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
  <Compile Remove="**\MaciOS\**\*.cs" />
  <None Include="**\MaciOS\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- iOS -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-ios')) != true">
  <Compile Remove="**\**\*.iOS.cs" />
  <None Include="**\**\*.iOS.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
  <Compile Remove="**\iOS\**\*.cs" />
  <None Include="**\iOS\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Mac Catalyst -->
<ItemGroup Condition="$(TargetFramework.StartsWith('net6.0-maccatalyst')) != true">
  <Compile Remove="**\**\*.MacCatalyst.cs" />
  <None Include="**\**\*.MacCatalyst.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
  <Compile Remove="**\MacCatalyst\**\*.cs" />
  <None Include="**\MacCatalyst\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

<!-- Windows -->
<ItemGroup Condition="$(TargetFramework.Contains('-windows')) != true">
  <Compile Remove="**\*.Windows.cs" />
  <None Include="**\*.Windows.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
  <Compile Remove="**\Windows\**\*.cs" />
  <None Include="**\Windows\**\*.cs" Exclude="$(DefaultItemExcludes);$(DefaultExcludesInProjectFolder)" />
</ItemGroup>

```

This XML configures the build system to remove platform-based filename and folder patterns under specific conditions:

- Don't compile C# code whose filename ends with *.Android.cs*, or that's located in the *Android* folder or sub-folder of the *Android* folder, if you aren't building for Android.
- Don't compile C# code whose filename ends with *.MaciOS.cs*, or that's located in the *MaciOS* folder or sub-folder of the *MaciOS* folder, if you aren't building for iOS and Mac Catalyst.
- Don't compile C# code whose filename ends with *.iOS.cs*, or that's located in the *iOS* folder or sub-folder of the *iOS* folder, if you aren't building for iOS.
- Don't compile C# code whose filename ends with *.MacCatalyst.cs*, or that's located in the *MacCatalyst* folder or sub-folder of the *MacCatalyst* folder, if you aren't building for Mac Catalyst.
- Don't compile C# code whose filename ends with *.Windows.cs*, or that's located in the *Windows* folder or sub-folder of the *Windows* folder, if you aren't building for Windows.

Invoke platform code

9/20/2022 • 5 minutes to read • [Edit Online](#)

 [Browse the sample](#)

In situations where .NET Multi-platform App UI (.NET MAUI) doesn't provide any APIs for accessing specific platform APIs, you can write your own code to access the required platform APIs. This requires knowledge of [Apple's iOS and MacCatalyst APIs](#), [Google's Android APIs](#), and [Microsoft's Windows App SDK APIs](#).

Platform code can be invoked from cross-platform code by using conditional compilation, or by using partial classes and partial methods.

Conditional compilation

Platform code can be invoked from cross-platform code by using conditional compilation to target different platforms.

The following example shows the `DeviceOrientation` enumeration, which will be used to specify the orientation of your device:

```
namespace InvokePlatformCodeDemos.Services
{
    public enum DeviceOrientation
    {
        Undefined,
        Landscape,
        Portrait
    }
}
```

Retrieving the orientation of your device requires writing platform code. This can be accomplished by writing a method that uses conditional compilation to target different platforms:

```

#if ANDROID
using Android.Content;
using Android.Views;
using Android.Runtime;
#elif IOS
using UIKit;
#endif

using InvokePlatformCodeDemos.Services;

namespace InvokePlatformCodeDemos.Services.ConditionalCompilation
{
    public class DeviceOrientationService
    {
        public DeviceOrientation GetOrientation()
        {
#if ANDROID
            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.Portrait;
#elif IOS
            UIInterfaceOrientation orientation = UIApplication.SharedApplication.StatusBarOrientation;
            bool isPortrait = orientation == UIInterfaceOrientation.Portrait || orientation ==
UIInterfaceOrientation.PortraitUpsideDown;
            return isPortrait ? DeviceOrientation.Portrait : DeviceOrientation.Landscape;
#else
            return DeviceOrientation.Undefined;
#endif
        }
    }
}

```

In this example, platform implementations of the `GetOrientation` method are provided for Android and iOS. On other platforms, `DeviceOrientation.Undefined` is returned. Alternatively, rather than returning `DeviceOrientation.Undefined` you could throw a `PlatformNotSupportedException` that specifies the platforms that implementations are provided for:

```
throw new PlatformNotSupportedException("GetOrientation is only supported on Android and iOS.");
```

The `DeviceOrientationService.GetOrientation` method can then be invoked from cross-platform code by creating an object instance and invoking its operation:

```

using InvokePlatformCodeDemos.Services;
using InvokePlatformCodeDemos.Services.ConditionalCompilation;
...

DeviceOrientationService deviceOrientationService = new DeviceOrientationService();
DeviceOrientation orientation = deviceOrientationService.GetOrientation();

```

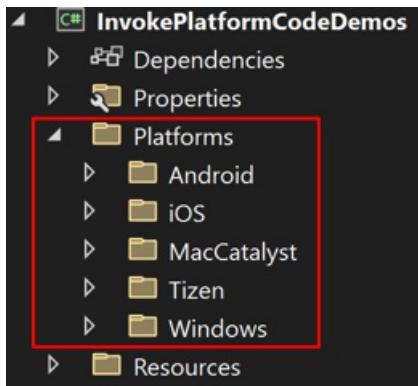
At build time the build system uses conditional compilation to target Android and iOS platform code to the correct platform.

For more information about conditional compilation, see [Conditional compilation](#).

Partial classes and methods

A .NET MAUI app project contains a *Platforms* folder, with each child folder representing a platform that .NET

MAUI can target:



The folders for each target platform contain platform-specific code that starts the app on each platform, plus any additional platform code you add. At build time, the build system only includes the code from each folder when building for that specific platform. For example, when you build for Android the files in the *Platforms > Android* folder will be built into the app package, but the files in the other *Platforms* folders won't be. This approach uses a feature called multi-targeting to target multiple platforms from a single project.

Multi-targeting can be combined with partial classes and partial methods to invoke platform functionality from cross-platform code. The process for doing this is to:

1. Define the cross-platform API as a partial class that defines partial method signatures for any operations you want to invoke on each platform. For more information, see [Define the cross-platform API](#).
2. Implement the cross-platform API per platform, by defining the same partial class and the same partial method signatures, while also providing the method implementations. For more information, see [Implement the API per platform](#).
3. Invoke the cross-platform API by creating an instance of the partial class and invoking its methods as required. For more information, see [Invoke the cross-platform API](#).

Define the cross-platform API

To invoke platform code from cross-platform code, the first step is to define the cross-platform API as a [partial class](#) that defines [partial method](#) signatures for any operations you want to invoke on each platform.

The following example shows the `DeviceOrientation` enumeration, which will be used to specify the orientation of your device:

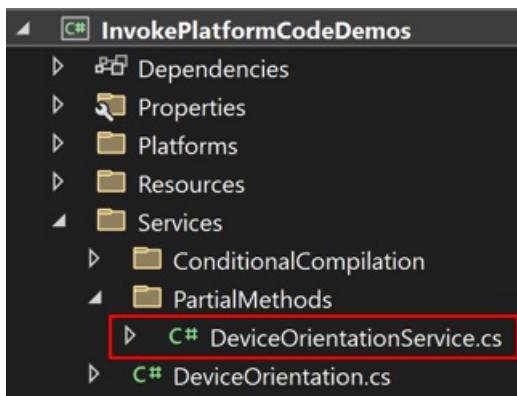
```
namespace InvokePlatformCodeDemos.Services
{
    public enum DeviceOrientation
    {
        Undefined,
        Landscape,
        Portrait
    }
}
```

The following example shows a cross-platform API that can be used to retrieve the orientation of a device:

```
namespace InvokePlatformCodeDemos.Services.PartialMethods
{
    public partial class DeviceOrientationService
    {
        public partial DeviceOrientation GetOrientation();
    }
}
```

The partial class is named `DeviceOrientationService`, which includes a partial method named `GetOrientation`.

The code file for this class must be outside of the *Platforms* folder:



Implement the API per platform

After defining the cross-platform API, it must be implemented on all platforms you're targeting by defining the same partial class and the same partial method signatures, while also providing the method implementations.

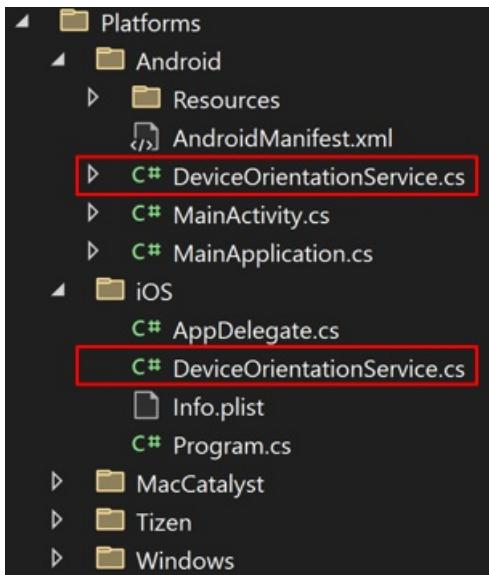
Platform implementations should be placed in the correct *Platforms* child folders to ensure that the build system only attempts to build platform code when building for the specific platform. The following table lists the default folder locations for platform implementations:

PLATFORM	FOLDER
Android	<i>Platforms > Android</i>
iOS	<i>Platforms > iOS</i>
MacCatalyst	<i>Platforms > MacCatalyst</i>
Tizen	<i>Platforms > Tizen</i>
Windows	<i>Platforms > Windows</i>

IMPORTANT

Platform implementations must be in the same namespace and same class that the cross-platform API was defined in.

The following screenshot shows the `DeviceOrientationService` classes in the *Android* and *iOS* folders:



Alternatively, multi-targeting can be performed based on your own filename and folder criteria, rather than using the *Platforms* folders. For more information, see [Configure multi-targeting](#).

Android

The following example shows the implementation of the `GetOrientation` method on Android:

```
using Android.Content;
using Android.Runtime;
using Android.Views;

namespace InvokePlatformCodeDemos.Services.PartialMethods;

public partial class DeviceOrientationService
{
    public partial 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.Portrait;
    }
}
```

iOS

The following example shows the implementation of the `GetOrientation` method on iOS:

```
using UIKit;

namespace InvokePlatformCodeDemos.Services.PartialMethods;

public partial class DeviceOrientationService
{
    public partial DeviceOrientation GetOrientation()
    {
        UIInterfaceOrientation orientation = UIApplication.SharedApplication.StatusBarOrientation;
        bool isPortrait = orientation == UIInterfaceOrientation.Portrait || orientation ==
UIInterfaceOrientation.PortraitUpsideDown;
        return isPortrait ? DeviceOrientation.Portrait : DeviceOrientation.Landscape;
    }
}
```

After providing the platform implementations, the API can be invoked from cross-platform code by creating an object instance and invoking its operation:

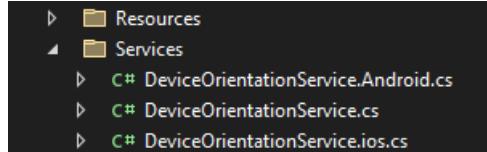
```
using InvokePlatformCodeDemos.Services;
using InvokePlatformCodeDemos.Services.PartialMethods;
...
DeviceOrientationService deviceOrientationService = new DeviceOrientationService();
DeviceOrientation orientation = deviceOrientationService.GetOrientation();
```

At build time the build system will use multi-targeting to combine the cross-platform partial class with the partial class for the target platform, and build it into the app package.

Configure multi-targeting

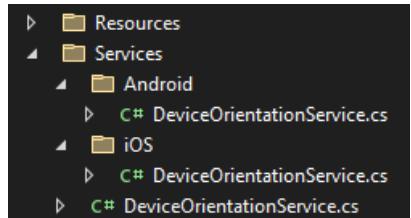
.NET MAUI apps can also be multi-targeted based on your own filename and folder criteria. This enables you to structure your .NET MAUI app project so that you don't have to place your platform code into child folders of the *Platforms* folder.

For example, a standard multi-targeting pattern is to include the platform as an extension in the filename for the platform code. The build system can be configured to combine cross-platform partial classes with platform partial classes based on this pattern:



```
▶ └── Resources
└── Services
  └── C# DeviceOrientationService.Android.cs
  └── C# DeviceOrientationService.cs
  └── C# DeviceOrientationService.iOS.cs
```

Another standard multi-targeting pattern is to include the platform as a folder name. The build system can be configured to combine cross-platform partial classes with platform partial classes based on this pattern:



```
▶ └── Resources
└── Services
  └── Android
    └── C# DeviceOrientationService.cs
  └── iOS
    └── C# DeviceOrientationService.cs
  └── C# DeviceOrientationService.cs
```

For more information, see [Configure multi-targeting](#).

.NET MAUI local databases

9/20/2022 • 6 minutes to read • [Edit Online](#)

 [Browse the sample](#)

The SQLite database engine allows .NET Multi-platform App UI (.NET MAUI) apps to load and save data objects in shared code. You can integrate SQLite.NET into .NET MAUI apps, to store and retrieve information in a local database, by following these steps:

1. [Install the NuGet package.](#)
2. [Configure constants.](#)
3. [Create a database access class.](#)
4. [Access data.](#)
5. [Advanced configuration.](#)

The sample app uses an SQLite database table to store todo items.

Install the SQLite NuGet package

Use the NuGet package manager to search for the `sqlite-net-pcl` package and add the latest version to your .NET MAUI app 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 MAUI projects.

Install SQLitePCLRaw.bundle_green

In addition to `sqlite-net-pcl`, you *temporarily* need to install the underlying dependency that exposes SQLite on each platform:

- **ID:** SQLitePCLRaw.bundle_green
- **Version:** 2.1.2
- **Authors:** Eric Sink
- **Owners:** Eric Sink
- **NuGet link:** [SQLitePCLRaw.bundle_green](#)

Configure app constants

Configuration data, such as database filename and path, can be stored as constants in your app. 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 =>
        Path.Combine(FileSystem.AppDataDirectory, DatabaseFilename);
}

```

In this example, 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 sample app defines a `TodoItemDatabase` class for this purpose.

Lazy initialization

The `TodoItemDatabase` uses asynchronous lazy initialization to delay initialization of the database until it's first accessed, with a simple `Init` method that gets called by each method in the class:

```

public class TodoItemDatabase
{
    SQLiteAsyncConnection Database;

    public TodoItemDatabase()
    {
    }

    async Task Init()
    {
        if (Database is not null)
            return;

        Database = new SQLiteAsyncConnection(Constants.DatabasePath, Constants.Flags);
        var result = await Database.CreateTableAsync<TodoItem>();
    }
    ...
}

```

In order to start database initialization, avoid blocking execution, and have the opportunity to catch exceptions, the sample app 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.

The following example shows the data manipulation methods in the sample app:

```

public class TodoItemDatabase
{
    ...
    public async Task<List<TodoItem>> GetItemsAsync()
    {
        await Init();
        return await Database.Table<TodoItem>().ToListAsync();
    }

    public async Task<List<TodoItem>> GetItemsNotDoneAsync()
    {
        await Init();
        return await Database.Table<TodoItem>().Where(t => t.Done).ToListAsync();

        // SQL queries are also possible
        //return await Database.QueryAsync<TodoItem>("SELECT * FROM [TodoItem] WHERE [Done] = 0");
    }

    public async Task<TodoItem> GetItemAsync(int id)
    {
        await Init();
        return await Database.Table<TodoItem>().Where(i => i.ID == id).FirstOrDefaultAsync();
    }

    public async Task<int> SaveItemAsync(TodoItem item)
    {
        await Init();
        if (item.ID != 0)
            return await Database.UpdateAsync(item);
        else
            return await Database.InsertAsync(item);
    }

    public async Task<int> DeleteItemAsync(TodoItem item)
    {
        await Init();
        return await Database.DeleteAsync(item);
    }
}

```

Access data

The `TodoItemDatabase` class can be registered as a singleton that can be used throughout the app if you are using dependency injection. For example, you can register your pages and the database access class as services on the `IServiceCollection` object, in `MauiProgram.cs`, with the `AddSingleton` and `AddTransient` methods:

```

builder.Services.AddSingleton<TodoListPage>();
builder.Services.AddTransient<TodoItemPage>();

builder.Services.AddSingleton<TodoItemDatabase>();

```

These services can then be automatically injected into class constructors, and accessed:

```

TodoItemDatabase database;

public TodoItemPage(TodoItemDatabase todoItemDatabase)
{
    InitializeComponent();
    database = todoItemDatabase;
}

async void OnSaveClicked(object sender, EventArgs e)
{
    if (string.IsNullOrWhiteSpace(Item.Name))
    {
        await DisplayAlert("Name Required", "Please enter a name for the todo item.", "OK");
        return;
    }

    await database.SaveItemAsync(Item);
    await Shell.Current.GoToAsync("../");
}

```

Alternatively, new instances of the database access class can be created:

```

TodoItemDatabase database;

public TodoItemPage()
{
    InitializeComponent();
    database = new TodoItemDatabase();
}

```

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.

Consume a REST-based web service

9/20/2022 • 7 minutes to read • [Edit Online](#)

 [Browse the sample](#)

Representational State Transfer (REST) is an architectural style for building web services. REST requests are typically made over HTTPS 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.
- **DELETE** – this operation is used to delete an item of data on the web service.

Web service APIs that adhere to REST 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).

REST-based 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 simplicity of REST has helped make it the primary method for accessing web services in mobile apps.

Web service operations

The example REST service is written using ASP.NET Core and provides the following operations:

OPERATION	HTTP METHOD	RELATIVE URI	PARAMETERS
Get a list of todo items	GET	/api/todoitems/	
Create a new todo item	POST	/api/todoitems/	A JSON formatted TodoItem
Update a todo item	PUT	/api/todoitems/	A JSON formatted TodoItem
Delete a todo item	DELETE	/api/todoitems/{id}	

The .NET MAUI app and web service uses the `TodoItem` class to model the data that is displayed and sent to the web service for storage:

```
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` object, and is used by the web service to identify data to be updated or deleted. For example, to delete the `TodoItem` whose ID is

`6bb8a868-dba1-4f1a-93b7-24ebce87e243`, the .NET MAUI app sends a DELETE request to

`https://hostname/api/todoitems/6bb8a868-dba1-4f1a-93b7-24ebce87e243`.

When the Web API framework receives a request, it routes the request to an action. These actions are public methods in the `TodoItemsController` class. The Web API framework uses routing middleware to match the URLs of incoming requests and map them to actions. REST APIs should use attribute routing to model 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](#).

Create the HttpClient object

A .NET Multi-platform App UI (.NET MAUI) app can consume a REST-based web service by sending requests to the web service with the `HttpClient` class. This class provides functionality for sending HTTP requests and receiving HTTP responses from a URI identified resource. Each request is sent as an asynchronous operation.

The `HttpClient` object should be declared at the class-level so that it lives for as long as the app needs to make HTTP requests:

```
public class RestService : IRestService
{
    HttpClient _client;
    JsonSerializerOptions _serializerOptions;

    public List<TodoItem> Items { get; private set; }

    public RestService()
    {
        _client = new HttpClient();
        _serializerOptions = new JsonSerializerOptions
        {
            PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
            WriteIndented = true
        };
    }
    ...
}
```

The `JsonSerializerOptions` object is used to configure the formatting of the JSON payload that's received from and sent to the web service. For more information, see [How to instantiate JsonSerializerOptions instances with System.Text.Json](#).

Retrieve data

The `HttpClient.GetAsync` method is used to send a GET request to the web service specified by the URI, and then receive the response from the web service:

```

public async Task<List<TodoItem>> RefreshDataAsync()
{
    Items = new List<TodoItem>();

    Uri uri = new Uri(string.Format(Constants.RestUrl, string.Empty));
    try
    {
        HttpResponseMessage response = await _client.GetAsync(uri);
        if (response.IsSuccessStatusCode)
        {
            string content = await response.Content.ReadAsStringAsync();
            Items = JsonSerializer.Deserialize<List<TodoItem>>(content, _serializerOptions);
        }
    }
    catch (Exception ex)
    {
        Debug.WriteLine(@"\tERROR {0}", ex.Message);
    }

    return Items;
}

```

Data is received from the web service as a `HttpResponseMessage` object. It contains information about the response, including the status code, headers, and any body. The REST service sends an HTTP status code in its response, which can be obtained from 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. The `HttpResponseMessage.Content` property represents the content of the response, and is of type `HttpContent`. The `HttpContent` class represents the HTTP body and content headers, such as `Content-Type` and `Content-Encoding`. The content is then read into a `string` using the `HttpContent.ReadAsStringAsync` method. The `string` is then deserialized from JSON to a `List` of `TodoItem` objects.

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 a POST request to the web service specified by the URI, and then to receive the response from the web service:

```

public async Task SaveTodoItemAsync(TodoItem item, bool isNewItem = false)
{
    Uri uri = new Uri(string.Format(Constants.RestUrl, string.Empty));

    try
    {
        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);
        else
            response = await _client.PutAsync(uri, content);

        if (response.IsSuccessStatusCode)
            Debug.WriteLine(@"\tTodoItem successfully saved.");
    }
    catch (Exception ex)
    {
        Debug.WriteLine(@"\tERROR {0}", ex.Message);
    }
}

```

In this example, 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 its response, which can be obtained from the `HttpResponseMessage.IsSuccessStatusCode` property, to indicate whether the HTTP request succeeded or failed.

The typical 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 a PUT request to the web service specified by the URI, and then receive the response from the web service:

```

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 its response, which can be obtained from the `HttpResponseMessage.IsSuccessStatusCode` property, to indicate whether the HTTP request succeeded or failed.

The typical 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 a DELETE request to the web service specified by the URI, and then receive the response from the web service:

```
public async Task DeleteTodoItemAsync(string id)
{
    Uri uri = new Uri(string.Format(Constants.RestUrl, id));

    try
    {
        HttpResponseMessage response = await _client.DeleteAsync(uri);
        if (response.IsSuccessStatusCode)
            Debug.WriteLine(@"\tTodoItem successfully deleted.");
    }
    catch (Exception ex)
    {
        Debug.WriteLine(@"\tERROR {0}", ex.Message);
    }
}
```

The REST service sends an HTTP status code in its response, which can be obtained from the `HttpResponseMessage.IsSuccessStatusCode` property, to indicate whether the HTTP request succeeded or failed. The typical 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're developing a REST web service locally with a framework such as ASP.NET Core Web API, you can debug your web service and .NET MAUI app at the same time. In this scenario, to consume your web service over HTTP from Android emulators and iOS simulators, you must enable clear-text HTTP traffic in your .NET MAUI app. For more information, see [Connect to local web services from Android emulators and iOS simulators](#).

Connect to local web services from Android emulators and iOS simulators

9/20/2022 • 8 minutes to read • [Edit Online](#)

 [Browse the sample](#)

Many mobile and desktop apps consume web services. During the software development phase, it's common to deploy a web service locally and consume it from an app running in the Android emulator or iOS simulator. This avoids having to deploy the web service to a hosted endpoint, and enables a straightforward debugging experience because both the app and web service are running locally.

.NET Multi-platform App UI (.NET MAUI) apps that run on Windows or MacCatalyst can consume ASP.NET Core web services that are running locally over HTTP or HTTPS without any additional work, provided that you've [trusted your development certificate](#). However, additional work is required when the app is running in the Android emulator or iOS simulator, and the process is different depending on whether the web service is running over HTTP or HTTPS.

Local machine address

The Android emulator and iOS simulator both provide access to web services running over HTTP or HTTPS on your local machine. However, the local machine address is different for each.

Android

Each instance of the Android emulator is isolated from your development machine network interfaces, and runs behind a virtual router. Therefore, an emulated device can't see your development machine or other emulator instances on the network.

However, the virtual router for each emulator manages a special network space that includes pre-allocated addresses, with the `10.0.2.2` address being an alias to your host loopback interface (127.0.0.1 on your development machine). Therefore, given a local web service that exposes a GET operation via the `/api/todoitems/` relative URI, an app running on the Android emulator can consume the operation by sending a GET request to `http://10.0.2.2:<port>/api/todoitems/` or `https://10.0.2.2:<port>/api/todoitems/`.

iOS

The iOS simulator uses the host machine network. Therefore, apps running in the simulator can connect to web services running on your local machine via the machines IP address or via the `localhost` hostname. For example, given a local web service that exposes a GET operation via the `/api/todoitems/` relative URI, an app running on the iOS simulator can consume the operation by sending a GET request to `http://localhost:<port>/api/todoitems/` or `https://localhost:<port>/api/todoitems/`.

NOTE

When running a .NET MAUI app in the iOS simulator from Windows, the app is displayed in the [remote iOS simulator for Windows](#). However, the app is running on the paired Mac. Therefore, there's no localhost access to a web service running in Windows for an iOS app running on a Mac.

Local web services running over HTTP

A .NET MAUI app running in the Android emulator or iOS simulator can consume an ASP.NET Core web service

that's running locally over HTTP. This can be achieved by configuring your .NET MAUI app project and your ASP.NET Core web service project to allow clear-text HTTP traffic.

In the code that defines the URL of your local web service in your .NET MAUI app, ensure that the web service URL specifies the HTTP scheme, and the correct hostname. The `DeviceInfo` class can be used to detect the platform the app is running on. The correct hostname can then be set as follows:

```
public static string BaseAddress =  
    DeviceInfo.Platform == DevicePlatform.Android ? "http://10.0.2.2:5000" : "http://localhost:5000";  
public static string TodoItemsUrl = $"{BaseAddress}/api/todoitems/";
```

For more information about the `DeviceInfo` class, see [Device information](#).

In addition, to run your app on Android you must add the required network security configuration, and to run your app on iOS you must opt-out of Apple Transport Security (ATS). For more information, see [Android network security configuration](#) and [iOS ATS configuration](#).

You must also ensure that your ASP.NET Core web service is configured to allow HTTP traffic. This can be achieved by adding a HTTP profile to the `profiles` section of `launchSettings.json` in your ASP.NET Core web service project:

```
{  
  ...  
  "profiles": {  
    "http": {  
      "commandName": "Project",  
      "dotnetRunMessages": true,  
      "launchBrowser": true,  
      "launchUrl": "api/todoitems",  
      "applicationUrl": "http://localhost:5000",  
      "environmentVariables": {  
        "ASPNETCORE_ENVIRONMENT": "Development"  
      }  
    },  
    ...  
  }  
}
```

A .NET MAUI app running in the Android emulator or iOS simulator can then consume an ASP.NET Core web service that's running locally over HTTP, provided that web service is launched with the `http` profile.

Android network security configuration

To enable clear-text local traffic on Android you must create a network security configuration file. This can be achieved by adding a new XML file named `network_security_config.xml` to the `Platforms\Android\Resources\xml` folder in your .NET MAUI app project. The XML file should specify the following configuration:

```
<?xml version="1.0" encoding="utf-8"?>  
<network-security-config>  
  <domain-config cleartextTrafficPermitted="true">  
    <domain includeSubdomains="true">10.0.2.2</domain>  
  </domain-config>  
</network-security-config>
```

Then, configure the `networkSecurityConfig` property on the `application` node in the `Platforms\Android\AndroidManifest.xml` file in your .NET MAUI app project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
    <application android:networkSecurityConfig="@xml/network_security_config" ...>
        ...
    </application>
</manifest>
```

For more information about network security configuration files, see [Network security configuration](#) on developer.android.com.

iOS ATS configuration

To enable clear-text local traffic on iOS you should opt-out of Apple Transport Security (ATS) in your .NET MAUI app. This can be achieved by adding the following configuration to the *Platforms\iOS\Info.plist* file in your .NET MAUI app project:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsLocalNetworking</key>
    <true/>
</dict>
```

For more information about ATS, see [Preventing Insecure Network Connections](#) on developer.apple.com.

Local web services running over HTTPS

A .NET MAUI app running in the Android emulator or iOS simulator can consume an ASP.NET Core web service that's running locally over HTTPS. The process to enable this is as follows:

1. Trust the self-signed development certificate on your machine. For more information, see [Trust your development certificate](#).
2. Specify the address of your local machine. For more information, see [Specify the local machine address](#).
3. Bypass the local development certificate security check. For more information, see [Bypass the certificate security check](#).

Each item will be discussed in turn.

Trust your development certificate

Installing the .NET Core SDK installs the ASP.NET Core HTTPS development certificate to your local user certificate store. However, while the certificate has been installed, it's not trusted. To trust the certificate, perform the following one-time step to run the dotnet `dev-certs` tool:

```
dotnet dev-certs https --trust
```

The following command provides help on the `dev-certs` tool:

```
dotnet dev-certs https --help
```

Alternatively, when you run an ASP.NET Core 2.1 project (or above), that uses HTTPS, Visual Studio will detect if the development certificate is missing and will offer to install it and trust it.

NOTE

The ASP.NET Core HTTPS development certificate is self-signed.

For more information about enabling local HTTPS on your machine, see [Enable local HTTPS](#).

Specify the local machine address

In the code that defines the URL of your local web service in your .NET MAUI app, ensure that the web service URL specifies the HTTPS scheme, and the correct hostname. The `DeviceInfo` class can be used to detect the platform the app is running on. The correct hostname can then be set as follows:

```
public static string BaseAddress =  
    DeviceInfo.Platform == DevicePlatform.Android ? "https://10.0.2.2:5001" : "https://localhost:5001";  
public static string TodoItemsUrl = $"{BaseAddress}/api/todoitems/";
```

For more information about the `DeviceInfo` class, see [Device information](#).

Bypass the certificate security check

Attempting to invoke a local secure web service from a .NET MAUI app running in an Android emulator will result in a `java.security.cert.CertPathValidatorException` being thrown, with a message indicating that the trust anchor for the certification path hasn't been found. Similarly, attempting to invoke a local secure web service from a .NET MAUI app running in an iOS simulator will result in an `NSURLErrorDomain` error with a message indicating that the certificate for the server is invalid. These errors occur because the local HTTPS development certificate is self-signed, and self-signed certificates aren't trusted by Android or iOS. Therefore, it's necessary to ignore SSL errors when an app consumes a local secure web service.

This can be accomplished by passing configured versions of the native `HttpMessageHandler` classes to the `HttpClient` constructor, which instruct the `HttpClient` class to trust localhost communication over HTTPS. The `HttpMessageHandler` class is an abstract class, whose implementation on Android is provided by the `AndroidMessageHandler` class, and whose implementation on iOS is provided by the `NSURLSessionHandler` class.

The following example shows a class that configures the `AndroidMessageHandler` class on Android and the `NSURLSessionHandler` class on iOS to trust localhost communication over HTTPS:

```

public class HttpClientHandlerService
{
    public HttpMessageHandler GetPlatformMessageHandler()
    {
#if ANDROID
        var handler = new CustomAndroidMessageHandler();
        handler.ServerCertificateCustomValidationCallback = (message, cert, chain, errors) =>
        {
            if (cert != null && cert.Issuer.Equals("CN=localhost"))
                return true;
            return errors == System.Net.Security.SslPolicyErrors.None;
        };
        return handler;
#elif IOS
        var handler = new NSUrlSessionHandler
        {
            TrustOverrideForUrl = IsHttpsLocalhost
        };
        return handler;
#else
        throw new PlatformNotSupportedException("Only Android and iOS supported.");
#endif
    }

#if ANDROID
    internal sealed class CustomAndroidMessageHandler : Xamarin.Android.Net.AndroidMessageHandler
    {
        protected override Java.Net.Ssl.IHostnameVerifier
GetSSLHostnameVerifier(Java.Net.Ssl.HttpsURLConnection connection)
        => new CustomHostnameVerifier();

        private sealed class CustomHostnameVerifier : Java.Lang.Object, Java.Net.Ssl.IHostnameVerifier
        {
            public bool Verify(string hostname, Java.Net.Ssl.ISSLSocket session)
            {
                return Java.Net.Ssl.HttpsURLConnection.DefaultHostnameVerifier.Verify(hostname, session) ||
                hostname == "10.0.2.2" && session.PeerPrincipal?.Name == "CN=localhost";
            }
        }
    }
#elif IOS
    public bool IsHttpsLocalhost(NSUrlSessionHandler sender, string url, Security.SecTrust trust)
    {
        if (url.StartsWith("https://localhost"))
            return true;
        return false;
    }
#endif
}

```

On Android, the `GetPlatformMessageHandler` method returns a `CustomAndroidMessageHandler` object that derives from `AndroidMessageHandler`. The `GetPlatformMessageHandler` method sets the `ServerCertificateCustomValidationCallback` property on a `CustomAndroidMessageHandler` object to a callback that ignores the result of the certificate security check for the local HTTPS development certificate.

On iOS, the `GetPlatformMessageHandler` method returns a `NSUrlSessionHandler` object that sets its `TrustOverrideForUrl` property to a delegate named `IsHttpsLocalHost` that matches the signature of the `NSUrlSessionHandler.NSUrlSessionHandlerTrustOverrideForUrlCallback` delegate. The `IsHttpsLocalHost` delegate returns `true` when the URL starts with `https://localhost`.

The resulting `HttpClientHandler` object can then be passed as an argument to the `HttpClient` constructor for debug builds:

```
#if DEBUG
    HttpClientHandlerService handler = new HttpClientHandlerService();
    HttpClient client = new HttpClient(handler.GetPlatformMessageHandler());
#else
    client = new HttpClient();
#endif
```

A .NET MAUI app running in the Android emulator or iOS simulator can then consume an ASP.NET Core web service that's running locally over HTTPS.

Deployment

9/20/2022 • 2 minutes to read • [Edit Online](#)

.NET Multi-platform App UI (.NET MAUI) uses a single-project system to manage the configuration of your cross-platform app. This configuration includes properties and settings that control building and packaging the app on each platform. For more information, see [Project configuration for .NET MAUI apps](#).

Hot restart enables you to quickly deploy an iOS app to a 64-bit local device, from Visual Studio 2022, without requiring a Mac build host. It removes the need for a full package rebuild by pushing new changes to the existing app bundle that's already present on your locally connected iOS device. It supports changes to code files, resources, and project references, enabling you to quickly test changes to your app during its development. For more information, see [Deploy an iOS app to a local device using hot restart](#).

Publishing

When distributing your .NET MAUI app for Android, you generate an *apk* (Android Package) or an *aab* (Android App Bundle) file. The *apk* is used for installing your app to an Android device, and the *aab* is used to publish your app to an Android store. With just a few configuration changes to your project, your app can be packaged for distribution. For more information, see [Publish a .NET MAUI app for Android](#).

When distributing your .NET MAUI app for iOS, you generate an *.ipa* file. An *.ipa* file is an iOS app archive file that stores an iOS app. Distributing a .NET MAUI app on iOS requires that the app is provisioned using a provisioning profile. Provisioning profiles are files that contain code signing information, as well as the identity of the app and its intended distribution mechanism. For more information about provisioning, see [Provision an iOS app for app store distribution](#). For more information about publishing an .NET MAUI app for iOS, see [Publish a .NET MAUI app for iOS](#).

When distributing your .NET MAUI app for macOS, you generate an *.app* or a *.pkg* file. An *.app* file is a self-contained app that can be run without installation, whereas a *.pkg* is an app packaged in an installer. For more information, see [Publish a .NET MAUI app for macOS](#).

When distributing your .NET MAUI app for Windows, you can publish the app and its dependencies to a folder for deployment to another system. You can also package the app into an MSIX package, which has numerous benefits for the users installing your app. For more information, see [Publish a .NET MAUI app for Windows](#).

Deploy an iOS app using hot restart

9/20/2022 • 6 minutes to read • [Edit Online](#)

Typically when building an app, your code is compiled and combined with other project resources to build an app bundle that's deployed to your simulator or device. With this model, when you make a change to your app, a new app bundle has to be built and deployed. While incremental builds can help to reduce compilation time, deployments usually take the same amount of time regardless of the size of the change.

.NET Multi-platform App UI (.NET MAUI) hot restart enables you to quickly deploy a .NET MAUI app to a 64-bit local iOS device, from Visual Studio 2022, without requiring a Mac build host. It removes the need for a full app bundle rebuild by pushing changes to the existing app bundle that's already present on your locally connected iOS device. It supports changes to code files, resources, and project references, enabling you to quickly test changes to your app during its development.

IMPORTANT

Hot restart can only be used to deploy apps that use the debug build configuration. You'll still need a Mac build host to build, sign, and deploy your app for production purposes.

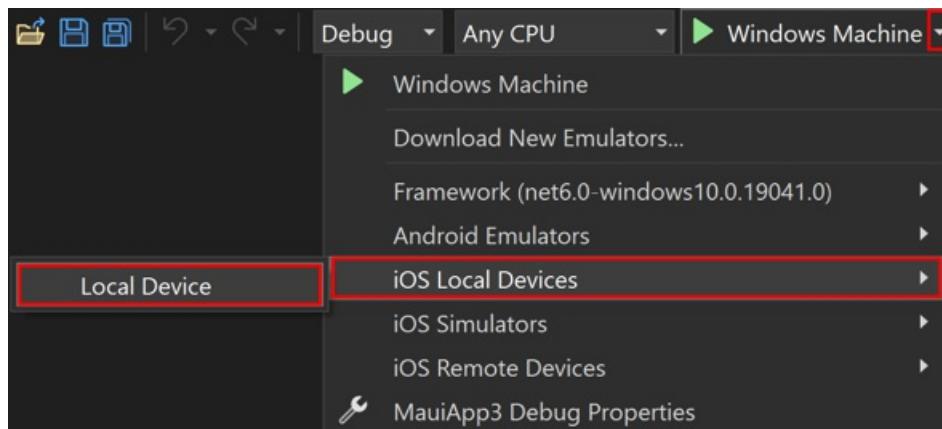
There are a number of requirements that must be met to use hot restart to deploy a .NET MAUI app to a locally connected iOS device:

- You must be using Visual Studio 2022 version 17.3 or greater.
- You must have iTunes (Microsoft Store or 64-bit version) installed on your development machine.
- You must have an [Apple Developer account](#) and paid [Apple Developer Program](#) enrollment.

Setup

Perform the following steps to set up hot restart:

1. In the Visual Studio toolbar, use the **Debug Target** drop down to select **iOS Local Devices** and then the **Local Device** entry:

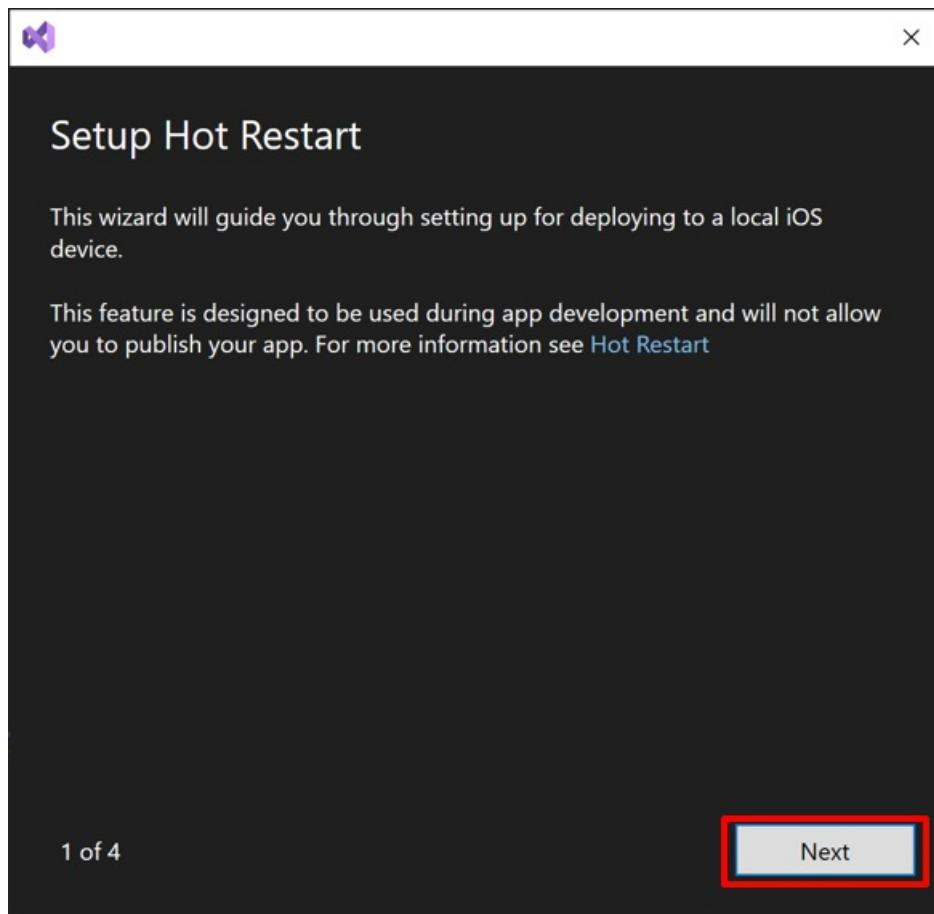


2. In the Visual Studio toolbar, select **Local Device**:

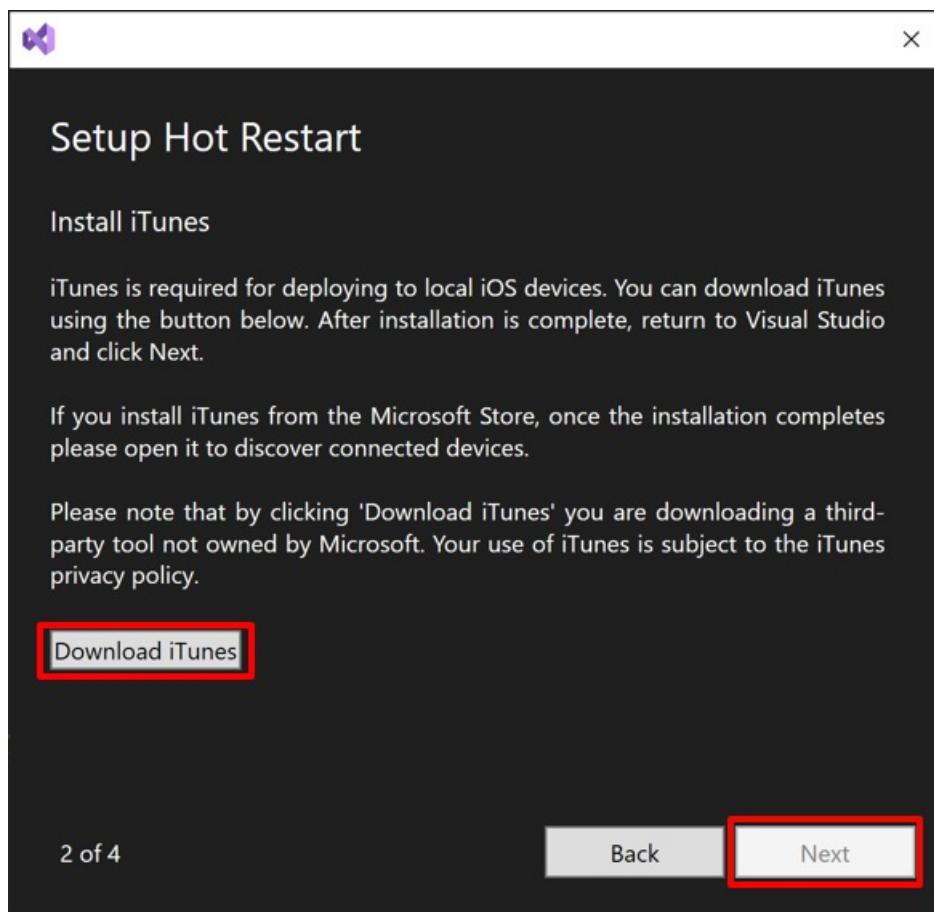


The **Setup Hot Restart** setup wizard will appear, which will guide you through setting up a local iOS device for hot restart deployment.

3. In the Setup Hot Restart setup wizard, select **Next**:



4. If you don't have iTunes installed, the setup wizard will prompt you to install it. In the **Setup Hot Restart** setup wizard, select **Download iTunes**:

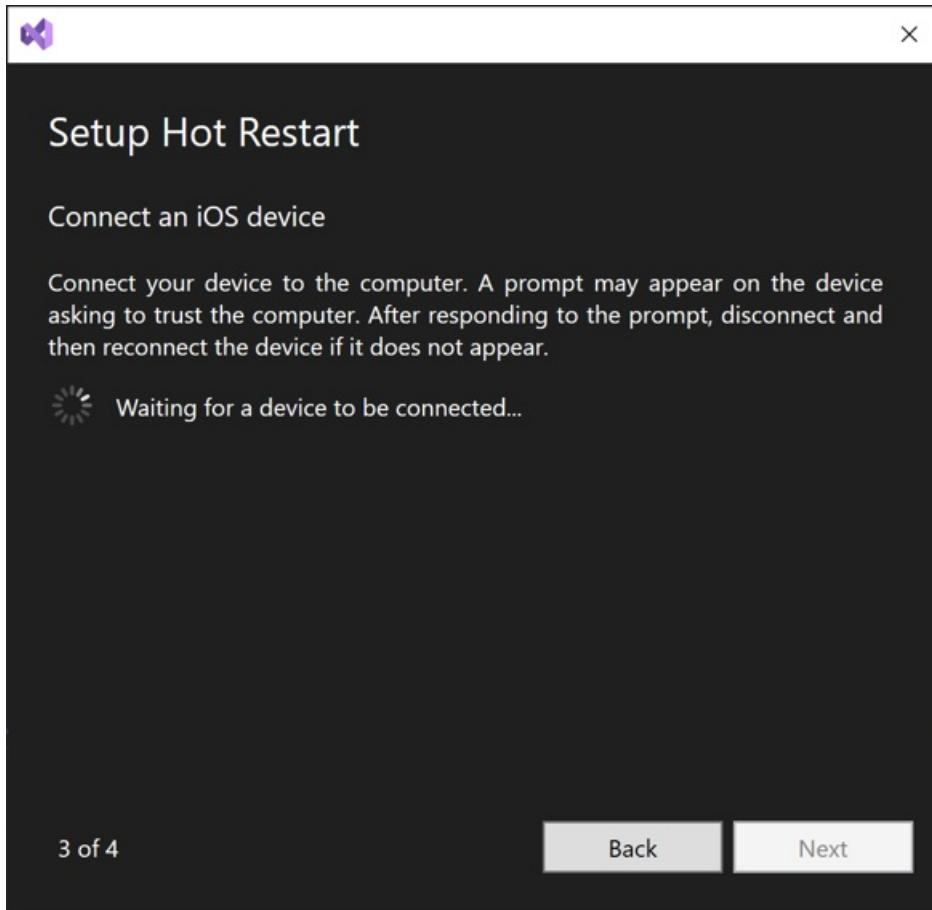


NOTE

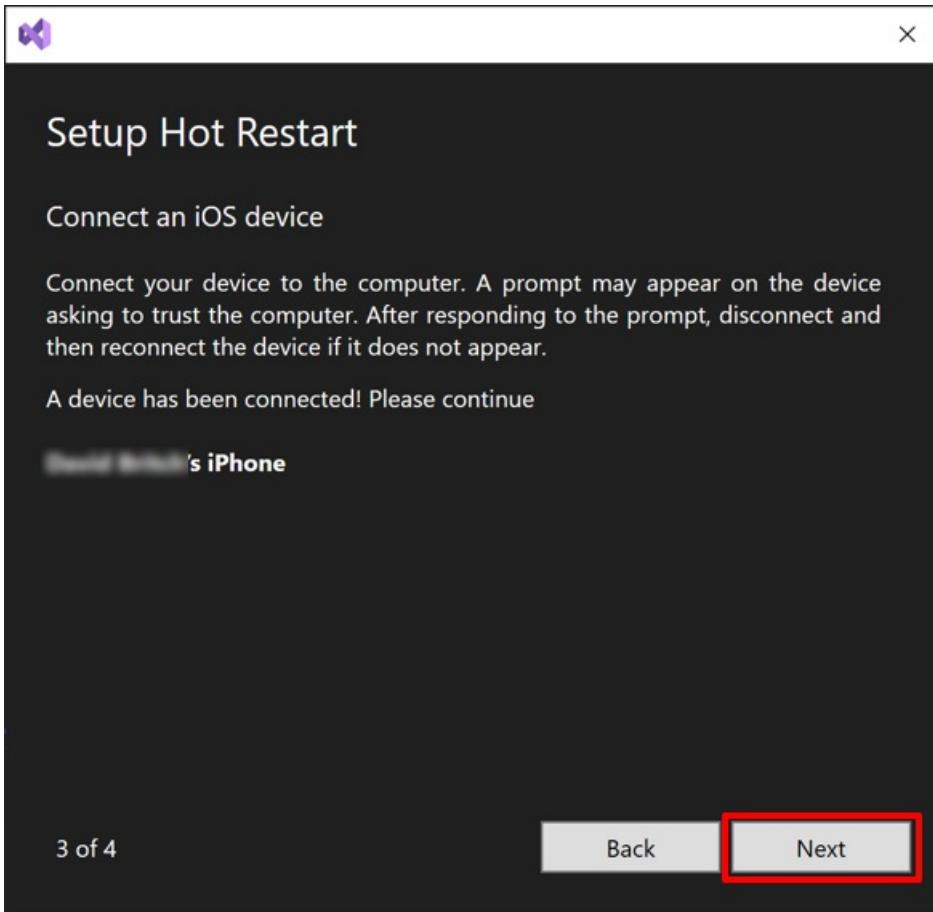
iTunes can either be installed from the Microsoft Store, or by downloading it from [Apple](#).

Wait for iTunes to download and then install it. If you install it from the Microsoft Store, once the installation completes please open it, then follow additional prompts to enable it to discover locally connected devices.

5. In the **Setup Hot Restart** setup wizard, select **Next** to move to the next step of the wizard that will prompt you to connect a local iOS device:



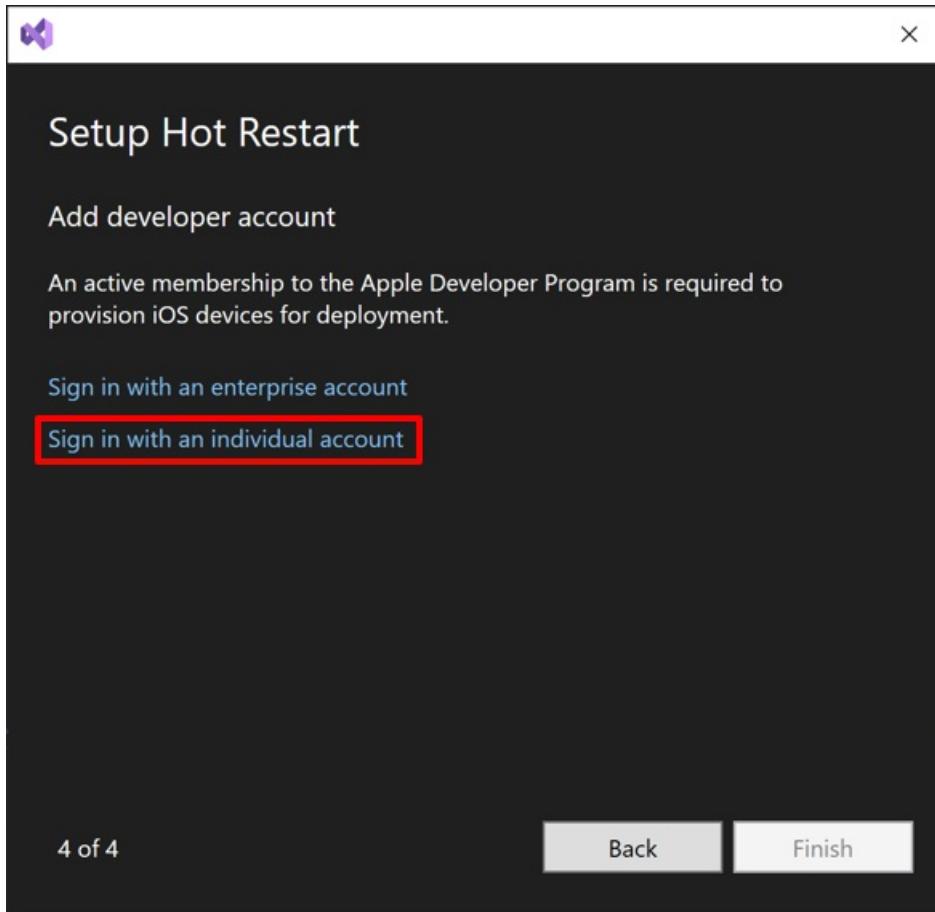
6. Connect your iOS device to your development machine via a USB cable. A prompt may appear on your device asking you to trust your development machine. On your device, click **Trust** and follow any additional device prompts.
7. In the **Setup Hot Restart** setup wizard, select **Next** once your local iOS device is detected:



NOTE

If the setup wizard fails to detect your local iOS device, disconnect then reconnect your local iOS device from your development machine. In addition, ensure that iTunes recognizes your local iOS device.

8. In the **Setup Hot Restart** setup wizard, click the [Sign in with an individual account](#) hyperlink to configure hot restart to use your individual Apple Developer Program account:

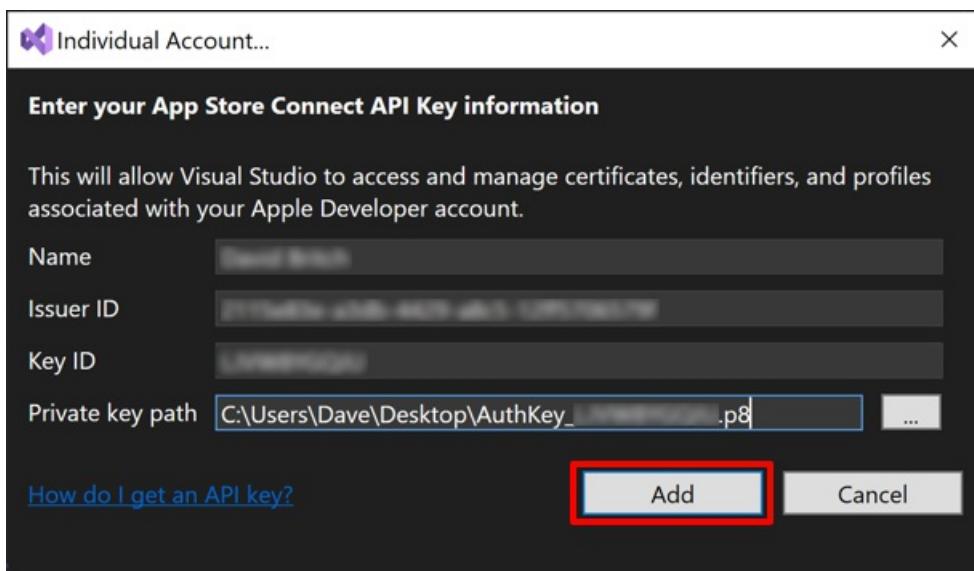


The **Individual account** dialog appears.

NOTE

Alternatively, to configure hot restart to use an enterprise Apple Developer account, click the **Sign in with an enterprise account** hyperlink and enter your credentials in the dialog that appears. Then proceed to step 12.

9. Create an App Store Connect API key. This will require you to have an [Apple Developer account](#) and paid [Apple Developer Program](#) enrollment. For information about creating an App Store Connect API key, see [Creating API Keys for App Store Connect API](#) on developer.apple.com.
10. In the **Individual account** dialog, enter your App Store Connect API key data:



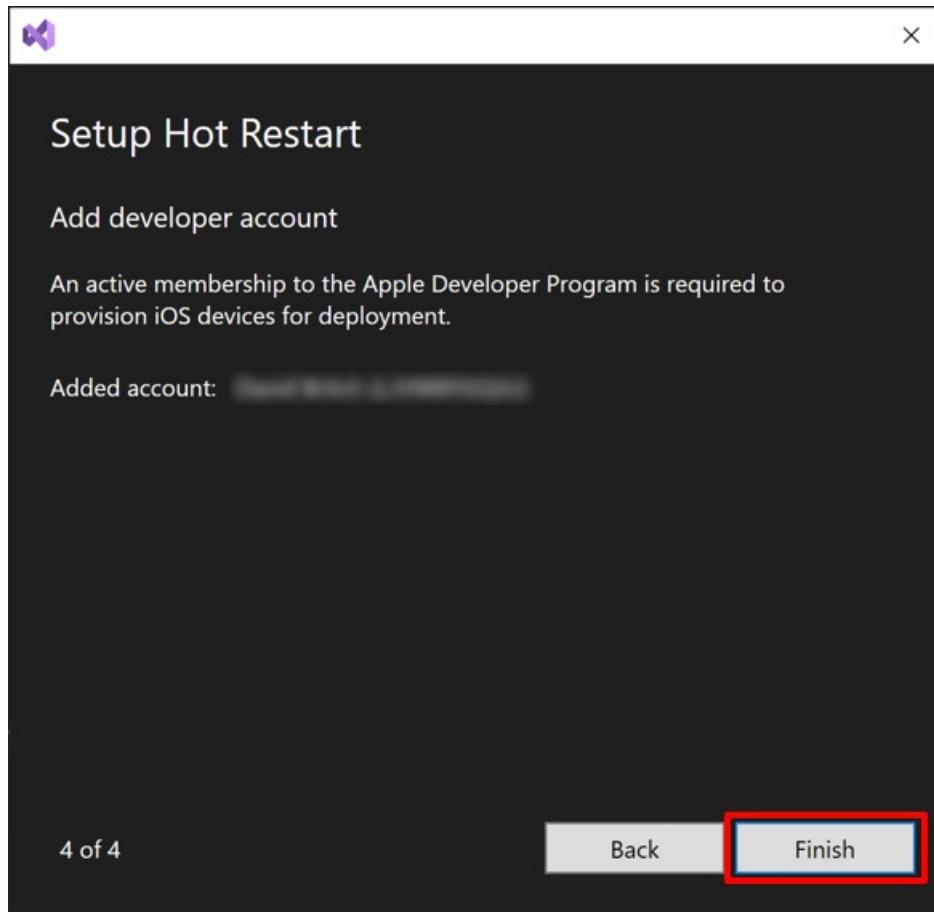
The **Name**, **Issuer ID**, and **Key ID** data can be found in [App Store Connect](#) by selecting **Users** and

Access and then the **Keys** tab. The **Private key** can also be downloaded from this location:

The screenshot shows the 'Keys' tab in the Apple Developer Program interface. It displays a table of keys. One key is listed under 'Active (1)'. The columns are NAME, GENERATED BY, KEY ID, LAST USED, and ACCESS. The 'NAME' column has a red box around it. The 'KEY ID' column has a red box around it. The 'ACCESS' column shows 'Admin' and a 'Download API Key' button, which is also highlighted with a red box.

11. In the **Individual account** dialog, click the **Add** button. The **Individual account** dialog will close.

12. In the **Setup Hot Restart** setup wizard, click the **Finish** button:



Your Apple Developer Program account will be added to Visual Studio and the **Setup Hot Restart** setup wizard will close.

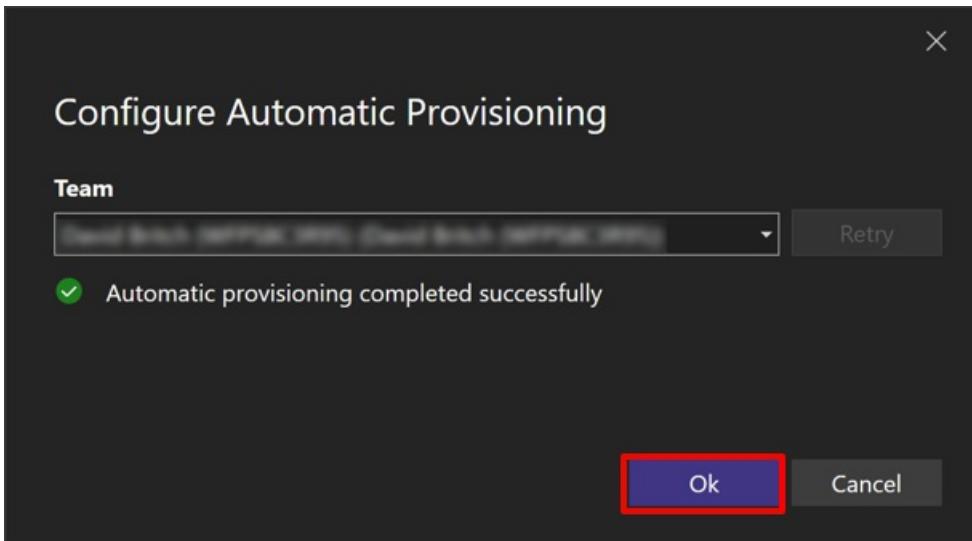
13. In **Solution Explorer**, right-click on your project and select **Properties**.

14. In the project properties, expand **iOS** and select **Bundle Signing**. Use the **Scheme** drop down to select **Automatic Provisioning** and then click the **Configure Automatic Provisioning** hyperlink:



The **Configure Automatic Provisioning** dialog will appear.

15. In the **Configure Automatic Provisioning** dialog, select the team for your Connect API key:



Visual Studio will complete the automatic provisioning process. Then, click the **Ok** button to dismiss the **Configure Automatic Provisioning** dialog.

NOTE

Using automatic provisioning is recommended so that additional iOS devices can be easily configured for deployment. However, you can use manual provisioning if the correct provisioning profiles are present on your machine.

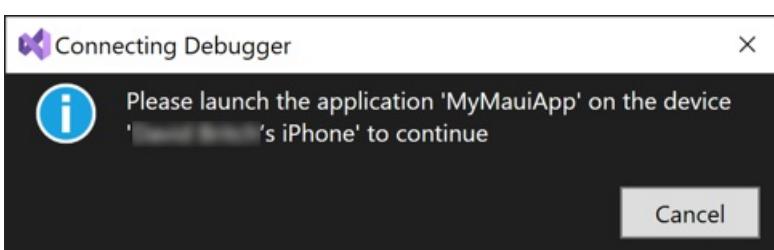
Deploy and debug using hot restart

After performing the initial setup, your local connected iOS device will appear in the debug target drop-down menu. To deploy and debug your app:

1. Ensure that your local connected iOS device is unlocked.
2. In the Visual Studio toolbar, select your local connected iOS device in the debug target drop down, and click the **Run** button to build your app and deploy it to your local iOS device:



3. After deploying your app, Visual Studio will display the **Connecting Debugger** dialog:



Launch the app on your device and Visual Studio will connect the debugger to your running app, and the **Connecting Debugger** dialog will be dismissed.

While you're debugging your app, you can edit your C# code and press the restart button in the Visual Studio toolbar to restart your debug session with the new changes applied:



Prevent code from executing

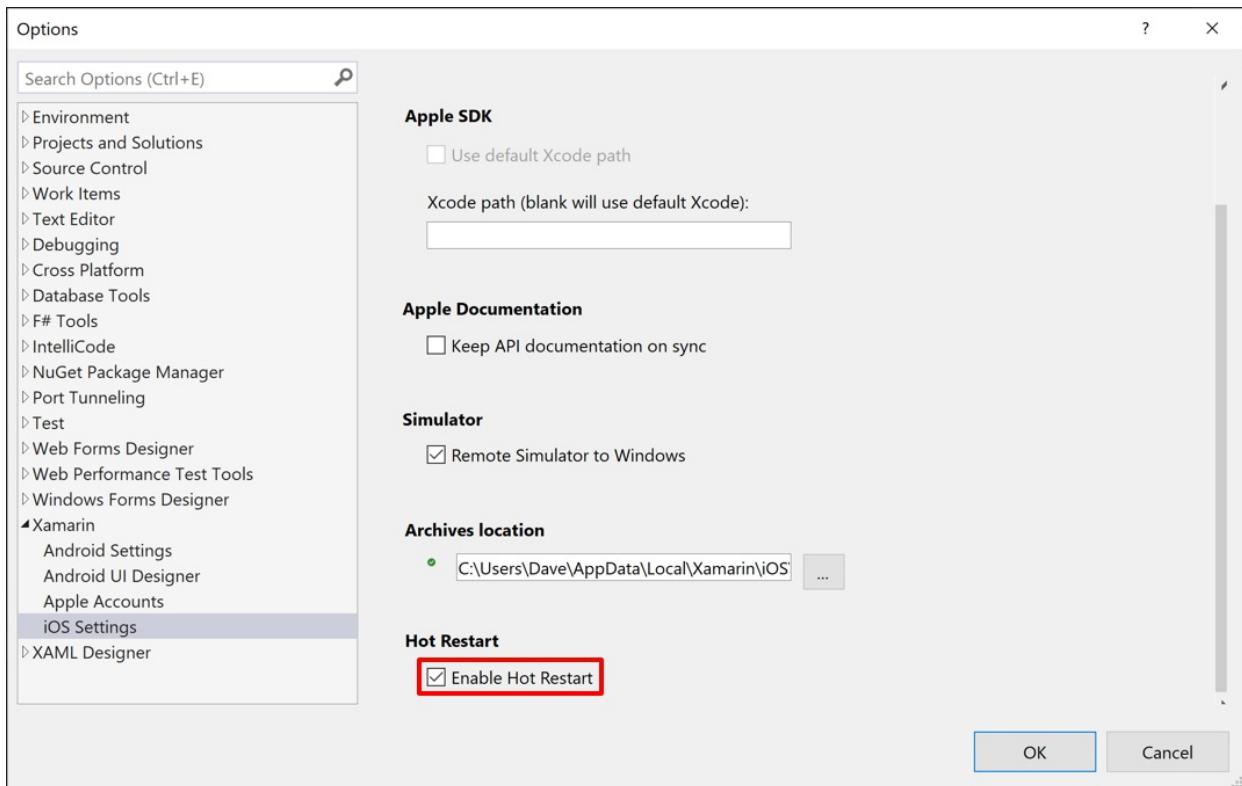
Storyboard and XIB files aren't supported when using hot restart, and your app may crash if it attempts to load these at runtime. Similarly, static iOS libraries and frameworks aren't supported and you may see runtime errors or crashes if your app attempts to load these.

In these scenarios, the `HOTRESTART` preprocessor symbol can be used to prevent code from executing when debugging with hot restart:

```
#if !HOTRESTART
    // Code here won't be executed when debugging with hot restart
#endif
```

Enable hot restart

Hot restart is enabled by default in Visual Studio 2022. If it's been previously disabled, it can be enabled by selecting **Tools > Options** from the Visual Studio menu bar. Next, in the **Options** dialog box, expand **Xamarin** and select **iOS Settings**. Then, ensure that **Enable Hot Restart** is checked:



Troubleshoot

.NET MAUI apps that use iOS asset catalogs are currently unsupported by hot restart.

iOS uses a watchdog that monitors app launch times and responsiveness, and terminates unresponsive apps. For example, the watchdog terminates apps that block the main thread for a significant time. On old iOS devices, the watchdog may terminate an app that's been deployed using hot restart before the debugger has connected to it. The workaround is to reduce the amount of processing performed in the app's startup path, and to use a more recent iOS device.

To report additional issues, please use the feedback tool at [Help > Send Feedback > Report a Problem](#).

Project configuration for .NET MAUI apps

9/20/2022 • 17 minutes to read • [Edit Online](#)

.NET MAUI uses a [single-project system](#) to manage the configuration of your cross-platform app. Project configuration in .NET MAUI is similar to other projects in Visual Studio, right-click on the project in the **Solution Explorer**, and select **Properties**.

Application

The **Application** section describes some settings related to which platforms your app targets, as well as the output file and default namespace.

- **General**

Describes some basic settings about your app.

SETTING	DEFAULT VALUE	DESCRIPTION
Assembly name	<code>\$(MSBuildProjectName)</code>	Specifies the name of the output file that will hold the assembly manifest.
Default namespace	Varies.	Specifies the base namespace for files added to your project. This generally defaults to the name of your project or a value you specified when you created the project.

- **iOS Targets**

If you're going to target iOS and macOS (using Mac Catalyst), these settings describe the target iOS version.

SETTING	DEFAULT VALUE	DESCRIPTION
Target the iOS platform	Checked	Specifies that this project will target the iOS platform.
Target iOS Framework	<code>net6.0-ios</code>	The Target Framework Moniker used to target iOS.
Minimum Target iOS Framework	<code>14.2</code>	The minimum version of iOS your app targets.

- **Android Targets**

If you're going to target Android, these settings describe the target Android version.

SETTING	DEFAULT VALUE	DESCRIPTION

SETTING	DEFAULT VALUE	DESCRIPTION
Target the Android platform	Checked	When checked, the .NET MAUI project will target and build an Android version of your app. Uncheck to disable the Android target.
Target Android Framework	<code>net6.0-android</code>	The Target Framework Moniker used to target Android.
Minimum Target Android Framework	<code>21.0</code>	The minimum version of Android your app targets.

- **Windows Targets**

If you're going to target Windows, these settings describe the target Windows version.

SETTING	DEFAULT VALUE	DESCRIPTION
Target the Windows platform	Checked	When checked, the .NET MAUI project will target and build a Windows version of your app. Uncheck to disable the Windows target.
Target Windows Framework	<code>net6.0-windows10.0.19041.0</code>	The Target Framework Moniker used to target Windows.
Minimum Target Windows Framework	<code>10.0.17763.0</code>	The minimum version of Windows your app targets.

Build

The Build section describes settings related to compiling your app.

General

Settings related to target platforms.

- **Conditional compilation symbols**

Specifies symbols on which to perform conditional compilation. Separate symbols with a semicolon `;`. Symbols can be broken up into target platforms. For more information, see [Conditional compilation](#).

- **Platform target**

Specifies the processor to be targeted by the output file. Choose `Any CPU` to specify that any processor is acceptable, allowing the application to run on the broadest range of hardware.

Typically this is set to `Any CPU` and the runtime identifier setting is used to target a CPU platform.

OPTION	DESCRIPTION
<code>Any CPU</code>	(Default) Compiles your assembly to run on any platform. Your application runs as a 64-bit process whenever possible and falls back to 32-bit when only that mode is available.

OPTION	DESCRIPTION
x86	Compiles your assembly to be run by the 32-bit, x86-compatible runtime.
x64	Compiles your assembly to be run by the 64-bit runtime on a computer that supports the AMD64 or EM64T instruction set.
ARM32	Compiles your assembly to run on a computer that has an Advanced RISC Machine (ARM) processor.
ARM64	Compiles your assembly to run by the 64-bit runtime on a computer that has an Advanced RISC Machine (ARM) processor that supports the A64 instruction set.

- **Nullable**

Specifies the project-wide C# nullable context. For more information, see [Nullable References](#).

OPTION	DESCRIPTION
Unset	(Default) If this setting isn't set, the default is <code>Disable</code> .
Disable	Nullable warnings are disabled. All reference type variables are nullable reference types.
Enable	The compiler enables all null reference analysis and all language features.
Warnings	The compiler performs all null analysis and emits warnings when code might dereference null.
Annotations	The compiler doesn't perform null analysis or emit warnings when code might dereference null.

- **Implicit global usings**

Enables implicit global usings to be declared by the project SDK. This is enabled by default and imports many of the .NET MAUI namespaces automatically to all code files. Code files don't need to add `using` statements for common .NET MAUI namespaces. For more information, see [MSBuild properties - ImplicitUsings](#).

- **Unsafe code**

Allow code that uses the `unsafe` keyword to compile. This is disabled by default.

- **Optimize code**

Enable compiler optimizations for smaller, faster, and more efficient output. There is an option for each target platform, in Debug or Release mode. Generally, this is enabled for Release mode, as the code is optimized for speed at the expense of helpful debugging information.

- **Debug symbols**

Specifies the kind of debug symbols produced during build.

Errors and warnings

Settings related to how errors and warnings are treated and reported during compilation.

- **Warning level**

Specifies the level to display for compiler warnings.

- **Suppress specific warnings**

Blocks the compiler from generating the specified warnings. Separate multiple warning numbers with a comma or a semicolon .

- **Treat warnings as errors**

When enabled, instructs the compiler to treat warnings as errors. This is disabled by default.

- **Treat specific warnings as errors**

Specifies which warnings are treated as errors. Separate multiple warning numbers with a comma or a semicolon .

Output

Settings related to generating the output file.

- **Base output path**

Specifies the base location for the project's output during build. Subfolders will be appended to this path to differentiate project configuration.

Defaults to .\bin\ .

- **Base intermediate output path**

Specifies the base location for the project's intermediate output during build. Subfolders will be appended to the path to differentiate project configuration.

Defaults to .\obj\ .

- **Reference assembly**

When enabled, produces a reference assembly containing the public API of the project. This is disabled by default.

- **Documentation file**

When enabled, generates a file containing API documentation. This is disabled by default.

Events

In this section you can add commands that run during the build.

- **Pre-build event**

Specifies commands that run before the build starts. Does not run if the project is up-to-date. A non-zero exit code will fail the build before it runs.

- **Post-build event**

Specifies commands that run before the build starts. Does not run if the project is up-to-date. A non-zero exit code will fail the build before it runs.

- **When to run the post-build event**

Specifies under which condition the post-build even will be run.

Strong naming

Settings related to signing the assembly.

- **Sign the assembly**

When enabled, signs the output assembly to give it a strong name.

Advanced

Additional settings related to the build.

- **Language version**

The version of the language available to the code in the project. Defaults to `10.0`.

- **Check for arithmetic overflow**

Throw exceptions when integer arithmetic produces out of range values. This setting is available for each platform. The default is disabled for each platform.

- **Deterministic**

Produce identical compilation output for identical inputs. This setting is available for each platform. The default is enabled for each platform.

- **Internal compiler error reporting**

Send internal compiler error reports to Microsoft. Defaults to `Prompt before sending`.

- **File alignment**

Specifies, in bytes, where to align the sections of the output file. This setting is available for each platform. The default is `512` for each platform.

Package

The **Package** section describes settings related to generating a NuGet package.

General

Settings related to generating a NuGet package.

- **Generate NuGet package on build**

When enabled, produces a NuGet package file during build operations. This is disabled by default.

- **Package ID**

The case-insensitive package identifier, which must be unique across the NuGet package gallery, such as nuget.org. IDs may not contain spaces or characters that aren't valid for a URL, and generally follow .NET namespace rules.

Defaults to the MSBuild value of `$(AssemblyName)`.

- **Title**

A human-friendly title of the package, typically used in UI displays as on nuget.org and the Package Manager in Visual Studio.

- **Package Version**

The version of the package, following the `major.minor.patch` pattern. Version numbers may include a pre-release suffix.

Defaults to the MSBuild value of `$(ApplicationDisplayVersion)`.

- **Authors**

A comma-separated list of authors, matching the profile names on nuget.org. These are displayed in the NuGet Gallery on nuget.org and are used to cross-reference packages by the same authors.

Defaults to the MSBuild value of `$(AssemblyName)`.

- **Company**

The name of the company associated with the NuGet package.

Defaults to the MSBuild value of `$(Authors)`.

- **Product**

The name of the product associated with the NuGet package.

Defaults to the MSBuild value of `$(AssemblyName)`.

- **Description**

A description of the package for UI display.

- **Copyright**

Copyright details for the package.

- **Project URL**

A URL for the package's home page, often shown in UI displays as well as nuget.org.

- **Icon**

The icon image for the package. Image file size is limited to 1 MB. Supported file formats include JPEG and PNG. An image resolution of 128x128 is recommended.

- **README**

The README document for the package. Must be a Markdown (.md) file.

- **Repository URL**

Specifies the URL for the repository where the source code for the package resides and/or from which it's being built. For linking to the project page, use the 'Project URL' field, instead.

- **Repository type**

Specifies the type of the repository. Default is 'git'.

- **Tags**

A semicolon-delimited list of tags and keywords that describe the package and aid discoverability of the packages through search and filtering.

- **Release notes**

A description of the changes made in the release of the package, often used in UI like the Updates tab of the Visual Studio Package Manager in place of the package description.

- **Pack as a .NET tool**

When enabled, packs the project as a special package that contains a console application that may be installed via the "dotnet tool" command. This is disabled by default.

- **Package Output Path**

Determines the output path in which the package will be dropped.

Defaults to the MSBuild value of `$(OutputPath)`.

- **Assembly neutral language**

Which language code is considered the neutral language. Defaults to unset.

- **Assembly version**

The version of the assembly, defaults to `1.0.0.0` if not set.

- **File version**

The version associated with the file, defaults to `1.0.0.0` if not set.

License

- **Package License**

Specify a license for the project's package. Defaults to `None`.

- **Symbols**

- **Produce a symbol package**

When enabled, creates an additional symbol package when the project is packaged. This is disabled by default.

Code Analysis

Settings related to code analysis.

All analyzers

Settings related to when analysis runs.

- **Run on build**

When enabled, runs code analysis on build. Defaults to enabled.

- **Run on live analysis**

When enabled, runs code analysis live in the editor as you type. Defaults to enabled.

.NET analysis

Settings related to .NET analyzers.

- **Enforce code style on build (experimental)**

When enabled, produces diagnostics about code style on build. This is disabled by default.

- **Enable .NET analyzers**

When enabled, runs .NET analyzers to help with API usage. Defaults to enabled.

- **Analysis level**

The set of analyzers that should be run in the project. Defaults to `Latest`. For more information, see [MSBuild: AnalysisLevel](#).

MAUI Shared

These are project settings for .NET MAUI that are shared across all target platforms.

General

General settings related to .NET MAUI.

- **Application Title**

The display name of the application.

- **Application ID**

The identifier of the application in reverse domain name format, for example: `com.microsoft.maui`.

- **Application ID (GUID)**

The identifier of the application in GUID format.

- **Application Display Version**

The version of the application. This should be a single digit integer. Defaults to `1`.

Android

These are Android-specific .NET MAUI settings.

Manifest

Settings related to the Android manifest.

- **Application name**

The string that's displayed as the name of the application. This is the name that's shown in the app's title bar. If not set, the label of the app's MainActivity is used as the application name. The default setting is `@string/app_name`, which refers to the string resource `app_name` location in `Resources/values/String.xml`.

- **Application package name**

A string that's used to uniquely identify the application. Typically, the package name is based on a reversed internet domain name convention, such as `com.company.appname`.

- **Application icon**

Specifies the application icon resource that will be displayed for the app. The setting `@drawable/icon` refers to the image file `icon.png` located in the `Resources/mipmap` folder.

- **Application theme**

Sets the UI style that's applied to the entire app. Every view in the app applies to the style attributes that are defined in the selected theme.

- **Application version number**

An integer value greater than zero that defines the version number of the app. Higher numbers indicate more recent versions. This value is evaluated programmatically by Android and by other apps, it isn't shown to users.

- **Application version name**

A string that specifies the version of the app to users. The version name can be a raw string or a reference

to a string resource.

- **Install location**

Indicates a preference as to where the app should be stored, whether in internal or external storage.

OPTION	DESCRIPTION
<code>Internal-only</code>	(Default) Specifies that the app can't be installed or moved to external storage.
<code>Prefer external</code>	Specifies that the app should be installed in external storage, if possible.
<code>Prefer internal</code>	Specifies that the app should be installed in internal storage, if possible.

- **Minimum Android version**

The oldest API level of an Android device that can install and run the app. Also referred to as

`minSdkVersion`.

- **Target Android version**

The target API level of the Android device where the app expects to run. This API level is used at run-time, unlike Target Framework, which is used at build time. Android uses this version as a way to provide forward compatibility. Also referred to as `targetSdkVersion`, this should match Target Framework `compileSdkVersion`.

Options

Miscellaneous options for building an Android app.

- **Android package format**

Either `apk` or `bundle`, which packages the Android application as an APK file or Android App Bundle, respectively. This can be set individually for both Debug and Release modes.

App Bundles are the latest format for Android release builds that are intended for submission on Google Play.

The default value is `apk`.

When `bundle` is selected, other MSBuild properties are set:

- `AndroidUseAapt2` is set to `True`.
- `AndroidUseApkSigner` is set to `False`.
- `AndroidCreatePackagePerAbi` is set to `False`.

- **Fast deployment (debug mode only)**

When enabled, deploys the app faster than normal to the target device. This process speeds up the build/deploy/debug cycle because the package isn't reinstalled when only assemblies are changed. Only the updated assemblies are resynchronized to the target device.

This is enabled by default.

- **Generate per ABI**

When enabled, generates one Android package (apk) per selected Application Binary Interface (ABI). This is disabled by default.

- **Use incremental packaging**

When enabled, uses the incremental Android packaging system (aapt2). This is enabled by default.

- **Multi-dex**

When enabled, allows the Android build system to use multidex. The default is disabled.

- **Code shrinker**

Selects the code shrinker to use.

- `ProGuard` (default) is considered the legacy code shrinker.
- `r8` is the next-generation tool which converts Java byte code to optimized dex code.

- **Uncompressed resources**

Leaves the specified resource extensions uncompressed. Separate extensions with a semicolon `;`. For example: `.mp3;.dll;.png`.

- **Developer instrumentation**

When enabled, developer instrumentation is provided for debugging and profiling. This can be set for individually for both Debug and Release modes.

The default is enabled for Debug builds.

- **Debugger**

Selects which debugger to use. The default is `.NET (Xamarin)`, which is used for managed code. The C++ debugger can be selected to debug native libraries used by the app.

- **AOT**

Enables Ahead-of-Time (AOT) compilation. This can be set for individually for both Debug and Release modes.

The default is enabled for Release builds.

- **LLVM**

Enables the LLVM optimizing compiler. The default is disabled.

- **Startup Tracing**

Enables startup tracing. This can be set for individually for both Debug and Release modes.

The default is enabled for Release builds.

- **Garbage Collection**

When enabled, uses the concurrent garbage collector. Defaults to enabled.

- **Enable trimming**

When enabled, trims the application during publishing. This can be set for individually for both Debug and Release modes. For more information, see [Trim self-contained deployments and executables](#) and [Trim options](#).

The default is enabled for Release builds.

- **Trimming granularity**

Controls how aggressively IL is discarded. There are two modes to select from:

- `Link` enables member-level trimming, which removes unused members from types.
- `CopyUsed` (default) enables assembly-level trimming, which keeps an entire assembly if any part of it is used.

- **Java max heap size**

Set this value to increase the size of memory that an app can use. For example, a value of `2G` increases the heap size to 2 gigabytes. Note that there isn't a guarantee of how large the heap will be, and requesting too much heap memory may force other apps to terminate prematurely.

The default is `1G`.

- **Additional Java options**

Specifies additional command-line options to pass to the Java compiler when building a `.dex` file. From the command line, you can type `java -help` to see the available options.

Package Signing

When enabled, signs the `.APK` file using the keystore details. This is disabled by default.

iOS

These are iOS-specific .NET MAUI settings.

Build

Settings related to building the iOS app.

- **Linker behavior**

The linker can strip out unused methods, properties, fields, events, structs, and even classes in order to reduce the overall size of the application. You can add a `Preserve` attribute to any of these in order to prevent the linker from stripping it out if it's needed for serialization or reflection.

WARNING

Enabling this feature may hinder debugging, as it may strip out property accessors that would allow you to inspect the state of your objects.

Options are:

- `Don't link`
- `Link Framework SDKs only` (default)
- `Link All`

- **LLVM**

When enabled, uses the LLVM optimized compiler. This can be set for individually for both Debug and Release modes.

The default is enabled for Release builds.

- **Float operations**

Performs all 32-bit float operations as 64-bit float operations.

- **Symbols**

When enabled, strips native debugging symbols from the output. This is enabled by default.

- **Garbage collector**

When enabled, uses the concurrent garbage collector. This is disabled by default.

- **Additional arguments**

Additional command line arguments to be passed to the application bundling code.

- **Optimization**

When enabled, optimizes *.PNG* images. This is enabled by default.

Bundle Signing

These settings are related to generating and signing the app bundle.

- **Scheme**

Configures the signing scheme for the bundle. It can be set to one of the following values:

- `Manual provisioning` : With this value, you'll be responsible for setting provisioning profiles and signing certificates yourself.
- `Automatic provisioning` : (default) With this value, Visual Studio will set provisioning profiles and signing certificates for you, which simplifies app deployment when testing on a device.

- **Signing identity**

A signing identity is the certificate and private key pair that's used for code-signing app bundle using Apple's codesign utility.

- `Developer (automatic)` (default)
- `Distribution (automatic)`

- **Provisioning profile**

Provisioning profiles are a way of tying together a team of developers with an App ID and, potentially, a list of test devices. The provisioning profiles list is filtered to only show provisioning profiles that match both the chosen identity and the App ID (aka bundle identifier) set in the *Info.plist*. If the provisioning profile that you're looking for isn't in the list, make sure that you've chosen a compatible identity and double-check that the bundle identifier set in your *Info.plist* is correct.

- **Custom Entitlements**

The plist file to use for entitlements. For more information, see [Entitlements and capabilities](#).

- **Custom Resource Rules**

The plist file containing custom rules used by Apple's codesign utility.

NOTE

As of Mac OSX 10.10, Apple has deprecated the use of custom resource rules. So, this setting should be avoided unless absolutely necessary.

- **Additional arguments**

Additional command line arguments to be passed to Apple's codesign utility during the code-signing phase of the build.

Debug

These are settings related to debugging.

- **Debugging**

When enabled, turns on debugging. The default is based on the current profile. Debug profiles enable debugging, while Release profiles disable debugging.

- **Profiling**

When enabled, turns on profiling.

IPA Options

When enabled, builds an iTunes Package Archive (IPA).

On Demand Resources

Settings related to on-demand resources. For more information, see [Apple Developer Documentation - On-Demand Resources Essentials](#).

- **Initial Tags**

The tags of the on-demand resources that are downloaded at the same time the app is downloaded from the app store. Separate tags with a semicolon ; .

- **Pre-fetch Order**

The tags of the on-demand resources that are downloaded after the app is installed. Separate tags with a semicolon ; .

- **Embed**

When enabled, embeds on-demand resources in the app bundle. This is enabled by default. Disable this setting to use the **Web server**.

- **Web server**

The URI of a web server that hosts on-demand resources.

Run Options

Options related to running the app on an iOS or macOS device.

- **Execution mode**

This setting determines how the app is run on the target device.

- **Start arguments**

Additional command line arguments to be passed to the app when it's started on the device.

- **Extra mlaunch arguments**

Additional command line arguments to be passed to **mlaunch**.

- **Environment variables**

Name-value pairs of environment variables to set when the app is run on the device.

Publish a .NET MAUI app for Android

9/20/2022 • 4 minutes to read • [Edit Online](#)

When distributing your .NET Multi-platform App UI (.NET MAUI) app for Android, you generate an *apk* (Android Package) or an *aab* (Android App Bundle) file. The *apk* is used for installing your app to an Android device, and the *aab* is used to publish your app to an Android store.

With just a few configuration changes to your project, your app can be packaged for distribution.

Validate package settings

Every Android app specifies a unique package identifier and a version. These identifiers are generally set in the Android app manifest file, which is located in your project folder at `.\Platforms\Android\AndroidManifest.xml`. However, these specific settings are provided by the project file itself. When a .NET MAUI app is built, the final `AndroidManifest.xml` file is automatically generated using the project file and the original `AndroidManifest.xml` file.

Your project file must declare `<ApplicationId>` and `<ApplicationVersion>` within a `<PropertyGroup>` node. These items should have been generated for you when the project was created. Just validate that they exist and are set to valid values:

```
<Project Sdk="Microsoft.NET.Sdk">

    <!-- other settings -->

    <PropertyGroup>
        <ApplicationId>com.companyname.myproject</ApplicationId>
        <ApplicationVersion>1</ApplicationVersion>
    </PropertyGroup>

</Project>
```

TIP

Some settings are available in the **Project Properties** editor in Visual Studio to change values. Right-click on the project in the **Solution Explorer** pane and choose **Properties**. For more information, see [Project configuration in .NET MAUI](#).

The following table describes how each project setting maps to the manifest file:

PROJECT SETTING	MANIFEST SETTING
<code>ApplicationId</code>	The <code>package</code> attribute of the <code><manifest></code> node: <code><manifest ... package="com.companyname.myproject"></code> .
<code>ApplicationVersion</code>	The <code>android:versionCode</code> attribute of the <code><manifest></code> node: <code><manifest ... android:versionCode="1"></code> .

Here's an example of an automatically generated manifest file with the package and version information specified:

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    package="com.hi.companyname.myproject"
    android:versionName="1.0.0">
    <!-- other settings -->
</manifest>
```

For more information about the manifest, see [Google Android App Manifest Overview](#).

Create a keystore file

Your app package should be signed. You use a keystore file to sign your package. The Java/Android SDKs includes the tools you need to generate a keystore. After generating a keystore file, you'll add it to your project and configure your project file to reference it. The Java SDK should be in your system path so that you can run the *keytool* tool.

Perform the following steps to create a keystore file:

1. Open a terminal and navigate to the folder of your project.

TIP

If Visual Studio is open, use the **View > Terminal** menu to open a terminal at the location of the solution or project. Navigate to the project folder.

2. Run the *keytool* tool with the following parameters:

```
keytool -genkey -v -keystore myapp.keystore -alias key -keyalg RSA -keysize 2048 -validity 10000
```

You'll be prompted to provide and confirm a password, followed by other settings.

The tool generates a *myapp.keystore* file, which should be located in the same folder as your project.

Add a reference to the keystore file

There are project-level settings you must set to sign your Android app with the keystore file. These settings are configured in a `<PropertyGroup>` node:

- `<AndroidKeyStore>` – Set to `True` to sign the app.
- `<AndroidSigningKeyStore>` – The keystore file created in the previous section: **myapp.keystore**.
- `<AndroidSigningKeyAlias>` – The `-alias` parameter value passed to the *keytool* tool: **key**.
- `<AndroidSigningKeyPass>` – The password you provided when creating the keystore file.
- `<AndroidSigningStorePass>` – The password you provided when creating the keystore file.

For security reasons, you don't want to supply a value for `<AndroidSigningKeyPass>` and `<AndroidSigningStorePass>` in the project file. You can provide these values on the command line when you publish the app. An example of providing the password is in the [Publish section](#).

```

<PropertyGroup Condition="$(TargetFramework.Contains('-android')) and '$(Configuration)' == 'Release'>
  <AndroidKeyStore>True</AndroidKeyStore>
  <AndroidSigningKeyStore>myapp.keystore</AndroidSigningKeyStore>
  <AndroidSigningKeyAlias>key</AndroidSigningKeyAlias>
  <AndroidSigningKeyPass></AndroidSigningKeyPass>
  <AndroidSigningStorePass></AndroidSigningStorePass>
</PropertyGroup>

```

The example `<PropertyGroup>` above adds a condition check, preventing those settings from being processed unless the condition check passes. The condition check looks for two things:

1. The target framework is set to something containing the text `-android`.
2. The build configuration is set to `Release`.

If either of those conditions fail, the settings aren't processed. More importantly, the `<AndroidKeyStore>` setting isn't set, preventing the app from being signed.

Publish

At this time, publishing is only supported through the .NET command line interface.

To publish your app, open a terminal and navigate to the folder for your .NET MAUI app project. Run the `dotnet publish` command, providing the following parameters:

PARAMETER	VALUE
<code>-f</code> or <code>--framework</code>	The target framework, which is <code>net6.0-android</code> .
<code>-c</code> or <code>--configuration</code>	The build configuration, which is <code>Release</code> .
<code>/p:AndroidSigningKeyPass</code>	This is the value used for the <code><AndroidSigningKeyPass></code> project setting, the password you provided when you created the keystore file.
<code>/p:AndroidSigningStorePass</code>	This is the value used for the <code><AndroidSigningStorePass></code> project setting, the password you provided when you created the keystore file.

WARNING

Attempting to publish a .NET MAUI solution will result in the `dotnet publish` command attempting to publish each project in the solution individually, which can cause issues when you've added other project types to your solution. Therefore, the `dotnet publish` command should be scoped to your .NET MAUI app project.

For example:

```

dotnet publish -f:net6.0-android -c:Release /p:AndroidSigningKeyPass=mypassword
/p:AndroidSigningStorePass=mypassword

```

Publishing builds the app, and then copies the `aab` and `apk` files to the `bin\Release\net6.0-android\publish` folder. There are two `aab` files, one unsigned and another signed. The signed variant has `-signed` in the file name.

For more information about the `dotnet publish` command, see [dotnet publish](#).

To learn how to upload a signed Android App Bundle to the Google Play Store, see [Upload your app to the Play Console](#).

See also

- [GitHub discussion and feedback: .NET MAUI Android target publishing/archiving](#)
- [Android Developers: About Android App Bundles](#)
- [Android Developers: Meet Google Play's target API level requirement](#)

Publish a .NET MAUI app for iOS

9/20/2022 • 8 minutes to read • [Edit Online](#)

When distributing your .NET Multi-platform App UI (.NET MAUI) app for iOS, you generate an *.ipa* file. An *.ipa* file is an iOS app archive file that stores an iOS app. With just a few configuration changes to your project, your app can be packaged for distribution.

Publishing a .NET MAUI app for iOS builds on top of Apple's provisioning process, which requires you to have:

- Created an Apple ID. For more information, see [Create Your Apple ID](#).
- Enrolled your Apple ID in the Apple Developer Program, which you have to pay to join. Enrolling in the Apple Developer Program enables you to create a *provisioning profile*, which contains code signing information. For more information, see [Apple Developer Program](#).
- A Mac on which you can build your app.

The process for publishing a .NET MAUI iOS app is as follows:

1. Create a certificate signing request on your Mac.
2. Create a distribution certificate in your Apple Developer Account.
3. Create a distribution profile in your Apple Developer Account.
4. Download provisioning profiles on your Mac.
5. Add any required entitlements to your app.
6. Add the code signing data to your app project.
7. Validate package settings.
8. Connect Visual Studio 2022 to a Mac build host.
9. Publish your app using .NET CLI.

Create a certificate signing request

To sign a .NET MAUI iOS app you must first create a certificate signing request (CSR) in Keychain Access on a Mac. For more information, see [Create a certificate signing request](#).

Create a distribution certificate

The CSR allows you to generate a distribution certificate, which will be used to confirm your identity. The distribution certificate must be created using the Apple ID for your Apple Developer Account. For more information, see [Create a distribution certificate](#).

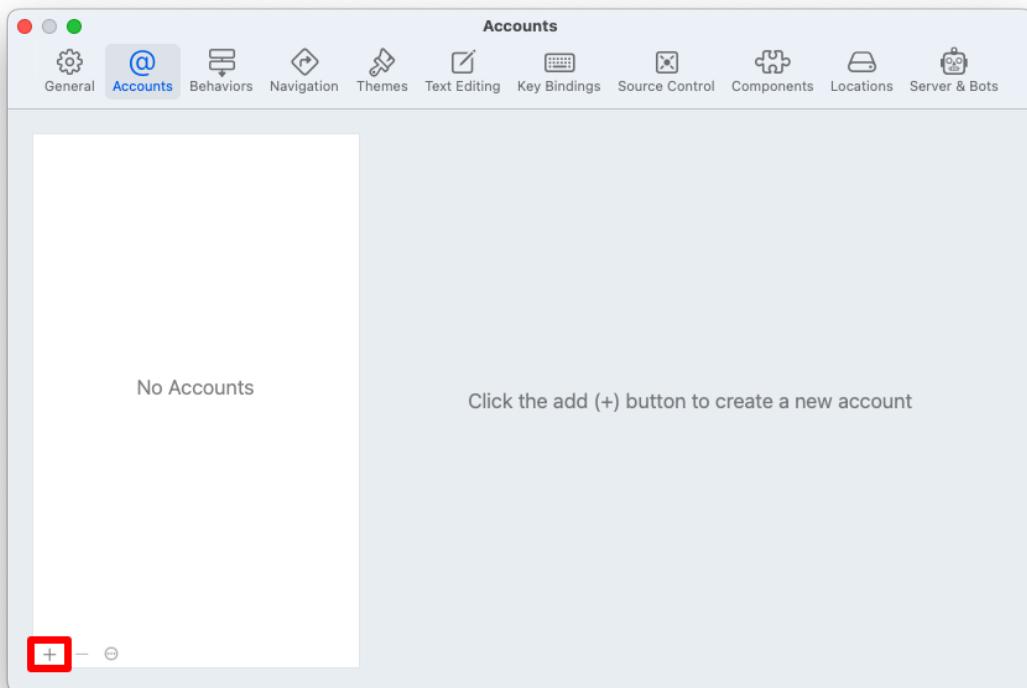
Create a distribution profile

To publish a .NET MAUI iOS app, you'll need to build a *Distribution Provisioning Profile* specific to it. This profile enables the app to be digitally signed for release so that it can be installed on an iOS device. A distribution provisioning profile contains an app ID and a distribution certificate. For more information, see [Create a distribution profile](#).

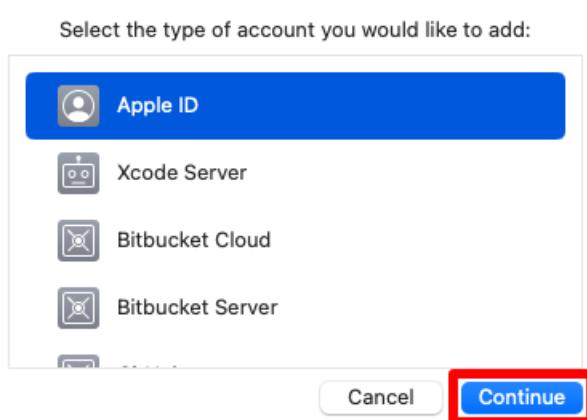
Download the provisioning profile on your Mac build host

After creating the provisioning profile, it must be added to your Mac build host. The profile can be downloaded to your Mac build host with the following steps:

1. On your Mac, launch Xcode.
2. In Xcode, select the **Xcode > Preferences...** menu item.
3. In the **Preferences** dialog, select the **Accounts** tab.
4. In the **Accounts** tab, click the **+** button to add your Apple Developer Account to Xcode:



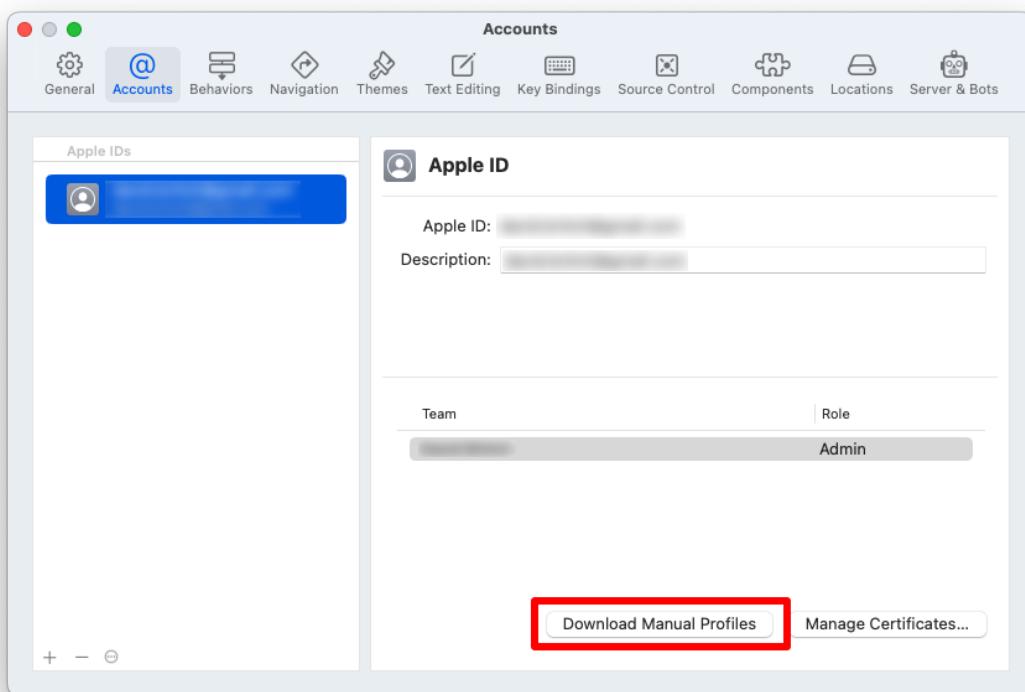
5. In the account type popup, select **Apple ID** and then click the **Continue** button:



6. In the sign in popup, enter your Apple ID and click the **Next** button.
7. In the sign in popup, enter your Apple ID password and click the **Next** button:



8. In the **Accounts** tab, click the **Manage Certificates...** button to ensure that your distribution certificate has been downloaded.
9. In the **Accounts** tab, click the **Download Manual Profiles** button to download your provisioning profiles:



10. Wait for the download to complete and then close Xcode.

Add entitlements to your app

In iOS, apps run in a sandbox that provides a set of rules that limit access between the app and system resources or user data. *Entitlements* are used to request the expansion of the sandbox to give your app additional capabilities. Any entitlements used by your app must be specified in an entitlements file. For more information about entitlements, see [Entitlements](#).

Add code signing data to your app project

There are project-level settings you must set to sign your iOS app with its provisioning profile. These settings are configured in a `<PropertyGroup>` node:

- `<RuntimeIdentifier>` – the runtime identifier (RID) for the project. Set to `ios-arm64`.
- `<CodesignKey>` – the name of the distribution certificate you installed into Keychain Access on your Mac build host.

- `<CodesignProvision>` – the provisioning profile name. This is the name you entered in the Apple Developer portal when creating your provisioning profile.
- `<CodesignEntitlement>` – the name of the entitlements file. Set to `Entitlements.plist`. This setting need only be specified if you're using entitlements.
- `<ArchiveOnBuild>` – a boolean value that indicates whether to build the app package. Set to `true`.
- `<TcpPort>` – the TCP port on which to communicate with your Mac build host. Set to `58181`.
- `<ServerAddress>` – the IP address of your Mac build host.
- `<ServerUser>` – the username to use when logging into your Mac build host. Use your system username rather than your full name.
- `<ServerPassword>` – the password for the username used to log into the Mac build host.
- `<_DotNetRootRemoteDirectory>` – the folder on the Mac build host that contains the .NET SDK. Set to `/Users/{macOS username}/Library/Caches/Xamarin/XMA/SDKs/dotnet/`.

IMPORTANT

Values for these settings don't have to be provided in the project file. They can also be provided on the command line when you publish the app. This enables you to omit specific values from your project file. For example, values for `<ServerAddress>`, `<ServerUser>`, `<ServerPassword>`, and `<_DotNetRootRemoteDirectory>` will typically be provided on the command line for security reasons. An example of this can be found in the [Publish](#) section.

The following example shows a typical property group for building and signing your iOS app with its provisioning profile:

```
<PropertyGroup Condition="$(TargetFramework.Contains('-ios')) and '$(Configuration)' == 'Release'>
  <RuntimeIdentifier>ios-arm64</RuntimeIdentifier>
  <CodesignKey>iPhone Distribution: John Smith (AY2GDE9QM7)</CodesignKey>
  <CodesignProvision>MyMauiApp</CodesignProvision>
  <ArchiveOnBuild>true</ArchiveOnBuild>
  <TcpPort>58181</TcpPort>
</PropertyGroup>
```

This example `<PropertyGroup>` adds a condition check, preventing the settings from being processed unless the condition check passes. The condition check looks for two items:

1. The target framework is set to something containing the text `-ios`.
2. The build configuration is set to `Release`.

If either of these conditions fail, the settings aren't processed. More importantly, the `<CodesignKey>` and `<CodesignProvision>` settings aren't set, preventing the app from being signed.

Validate package settings

Every iOS app specifies a unique package identifier and a version. These identifiers are generally set in your project file, from where they are copied to your `Info.plist` file that's located in your project folder at `Platforms\iOS\Info.plist`.

Your project file must declare `<ApplicationId>` and `ApplicationVersion` within a `<PropertyGroup>` node. These items should have been generated for you when the project was created. Just validate that they exist and are set to valid values:

```

<Project Sdk="Microsoft.NET.Sdk">

    <!-- other settings -->

    <PropertyGroup>
        <ApplicationId>com.companyname.mymauiapp</ApplicationId>
        <ApplicationVersion>1</ApplicationVersion>
    </PropertyGroup>

</Project>

```

TIP

Some settings are available in the **Project Properties** editor in Visual Studio to change values. Right-click on the project in the **Solution Explorer** pane and choose **Properties**. For more information, see [Project configuration in .NET MAUI](#).

The following table describes how each project setting maps to the *Info.plist* file:

PROJECT SETTING	INFO.PLIST MANIFEST FIELD
<code>ApplicationId</code>	Bundle Identifier
<code>ApplicationVersion</code>	Build

Connect Visual Studio 2022 to your Mac build host

Building native iOS apps using .NET MAUI requires access to Apple's build tools, which only run on a Mac. Because of this, Visual Studio 2022 must connect to a network-accessible Mac to build .NET MAUI iOS apps. For more information, see [Pair to Mac for iOS development](#).

NOTE

The first time Pair to Mac logs into a Mac build host from Visual Studio 2022, it sets up SSH keys. With these keys, future logins will not require a username or password.

Publish

At this time, publishing is only supported through the .NET command line interface.

To publish your app, open a terminal and navigate to the folder for your .NET MAUI app project. Run the `dotnet publish` command, providing the following parameters:

PARAMETER	VALUE
<code>-f</code> or <code>--framework</code>	The target framework, which is <code>net6.0-ios</code> .
<code>-c</code> or <code>--configuration</code>	The build configuration, which is <code>Release</code> .

WARNING

Attempting to publish a .NET MAUI solution will result in the `dotnet publish` command attempting to publish each project in the solution individually, which can cause issues when you've added other project types to your solution. Therefore, the `dotnet publish` command should be scoped to your .NET MAUI app project.

In addition, the following common parameters can be specified on the command line if they aren't provided in a `<PropertyGroup>` in your project file:

PARAMETER	VALUE
<code>/p:RuntimeIdentifier</code>	The runtime identifier (RID) for the project. Use <code>ios-arm64</code> .
<code>/p:CodesignKey</code>	The name of the distribution certificate you installed into Keychain Access on your Mac build host.
<code>/p:CodesignProvision</code>	The provisioning profile name.
<code>/p:CodesignEntitlement</code>	The name of the entitlements file. Use <code>Entitlements.plist</code> .
<code>/p:ArchiveOnBuild</code>	A boolean value that indicates whether to produce the archive. Use <code>true</code> to produce the <code>.ipa</code> .
<code>/p:TcpPort</code>	The TCP port to use to communicate with the Mac build host, which is 58181.
<code>/p:ServerAddress</code>	The IP address of the Mac build host.
<code>/p:ServerUser</code>	The username to use when logging into the Mac build host. Use your system username rather than your full name.
<code>/p:ServerPassword</code>	The password for the username used to log into the Mac build host.
<code>/p:_DotNetRootRemoteDirectory</code>	The folder on the Mac build host that contains the .NET SDK. Use <code>/Users/{macOS username}/Library/Caches/Xamarin/XMA/SDKs/dotnet/</code>

IMPORTANT

Values for these parameters don't have to be provided on the command line. They can also be provided in the project file. For more information, see [Add code signing data to your app project](#).

For example, use the following command to create an `.ipa`:

```
dotnet publish -f:net6.0-ios -c:Release /p:ServerAddress={macOS build host IP address} /p:ServerUser={macOS username} /p:ServerPassword={macOS password} /p:TcpPort=58181 /p:ArchiveOnBuild=true /p:_DotNetRootRemoteDirectory=/Users/{macOS username}/Library/Caches/Xamarin/XMA/SDKs/dotnet/
```

NOTE

If the `ServerPassword` parameter is omitted from a command line build invocation, Pair to Mac attempts to log in to the Mac build host using the saved SSH keys.

Publishing builds the app, and then copies the `.ipa` to the `bin\Release\net6.0-ios\ios-arm64\publish` folder. During the publishing process it may be necessary to allow `codesign` to run on your paired Mac:



The `.ipa` file can then be uploaded to the App Store using App Store Connect. To learn how to use App Store Connect, see [App Store Connect workflow](#).

For more information about the `dotnet publish` command, see [dotnet publish](#).

See also

- [GitHub discussion and feedback: .NET MAUI iOS target publishing/archiving](#)

Provision an iOS app for app store distribution

9/20/2022 • 4 minutes to read • [Edit Online](#)

Distributing a .NET Multi-platform App UI (.NET MAUI) app on iOS requires that the app is provisioned using a *provisioning profile*. Provisioning profiles are files that contain code signing information, as well as the identity of the app and its intended distribution mechanism.

To publish a .NET MAUI iOS app, you'll need to build a *Distribution Provisioning Profile* specific to it. This profile enables the app to be digitally signed for release so that it can be installed on an iOS device. A distribution provisioning profile contains an app ID and a distribution certificate.

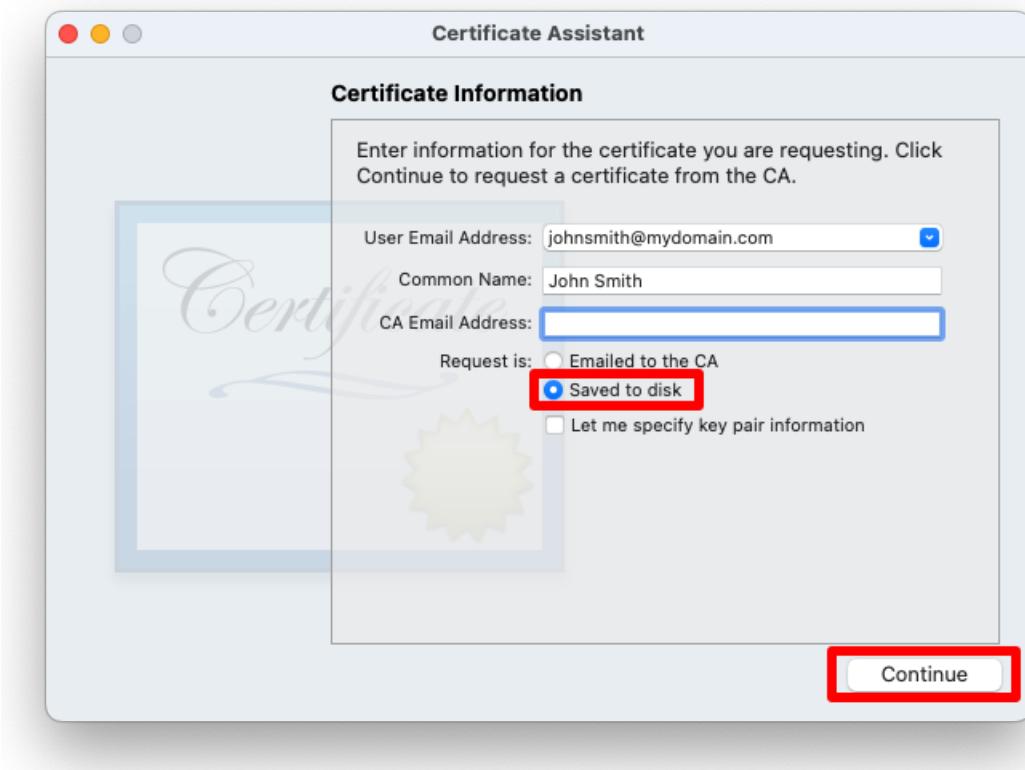
The process for creating a distribution provisioning profile is as follows:

1. Create a certificate signing request.
2. Create a distribution certificate.
3. Create an App ID.
4. Create a provisioning profile.

Create a certificate signing request

To create a distribution certificate, you'll first need to create a certificate signing request (CSR) in Keychain Access on a Mac:

1. On your Mac, launch Keychain Access.
2. In Keychain Access, select the **Keychain Access > Certificate Assistant > Request a Certificate from a Certificate Authority...** menu item.
3. In the **Certificate Assistant** dialog, enter an email address in the **User Email Address** field.
4. In the **Certificate Assistant** dialog, enter a name for the key in the **Common Name** field.
5. In the **Certificate Assistant** dialog, leave the **CA Email Address** field empty.
6. In the **Certificate Assistant** dialog, choose the **Saved to disk** radio button and click **Continue**:



7. Save the certificate signing request to a known location.
8. Close Keychain Access.

Create a distribution certificate

The CSR allows you to generate a distribution certificate, which confirms your identity. The distribution certificate must be created using the Apple ID for your Apple Developer Account:

1. In a web browser, login to your [Apple Developer Account](#).
2. In your Apple Developer Account, select the **Certificates, IDs & Profiles** tab.
3. On the **Certificates, Identifiers & Profiles** page, click the + button to create a new certificate.
4. On the **Create a New Certificate** page, select the **iOS Distribution (App Store and Ad Hoc)** radio button before clicking the **Continue** button:

Certificates, Identifiers & Profiles

[All Certificates](#)

Create a New Certificate

Software

Apple Development
Sign development versions of your iOS, macOS, tvOS, and watchOS apps. For use in Xcode 11 or later.

Apple Distribution
Sign your apps for submission to the App Store or for Ad Hoc distribution. For use with Xcode 11 or later.

iOS App Development
Sign development versions of your iOS app.

iOS Distribution (App Store and Ad Hoc)
Sign your iOS app for submission to the App Store or for Ad Hoc distribution.

Continue

5. On the **Create a New Certificate** page, click **Choose File**:

Certificates, Identifiers & Profiles

[« All Certificates](#)

Create a New Certificate

[Back](#)

[Continue](#)

Upload a Certificate Signing Request
To manually generate a Certificate, you need a Certificate Signing Request (CSR) file from your Mac. [Learn more >](#)

[Choose File](#)

6. In the Choose Files to Upload dialog, select the certificate request file (a file with a `.certSigningRequest` file extension) and then click **Upload**.

7. On the Create a New Certificate page, click the **Continue** button:

Certificates, Identifiers & Profiles

[« All Certificates](#)

Create a New Certificate

[Back](#)

[Continue](#)

Upload a Certificate Signing Request
To manually generate a Certificate, you need a Certificate Signing Request (CSR) file from your Mac. [Learn more >](#)

[Choose File](#)

CertificateSigningRequest.certSigningRequest

8. On the Download Your Certificate page, click the **Download** button:

Certificates, Identifiers & Profiles

[« All Certificates](#)

Download Your Certificate

[Download](#)

Certificate Details

Certificate Name

Certificate Type

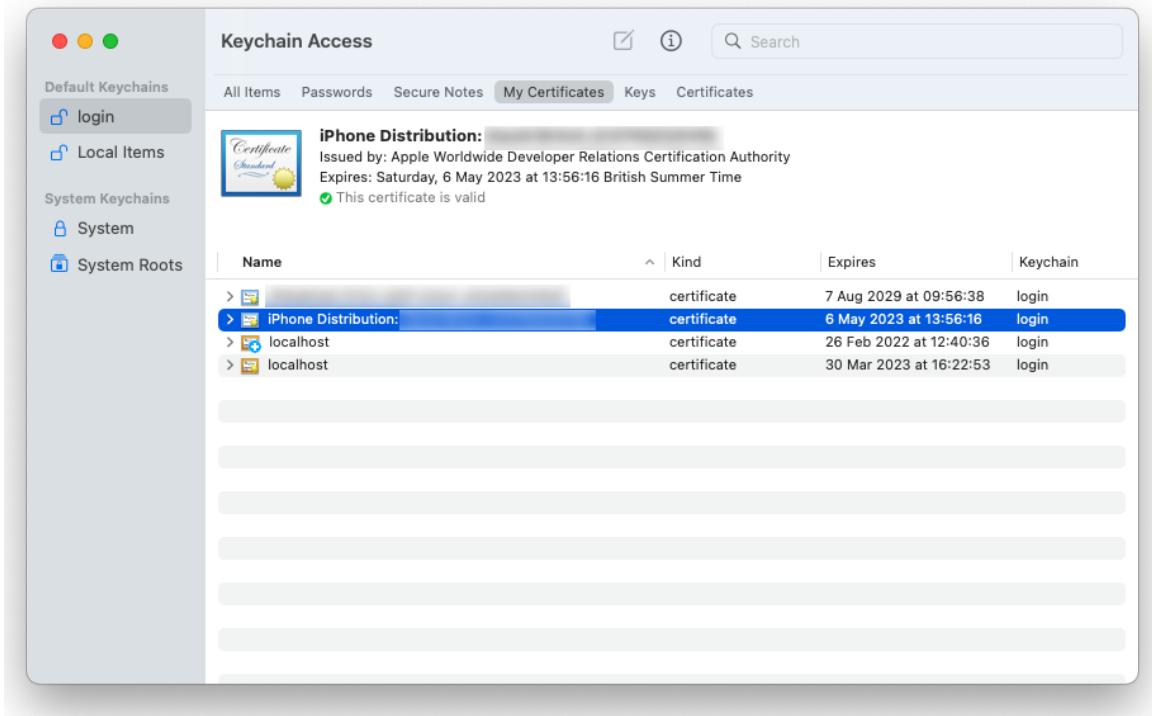
Download your certificate to your Mac, then double click the `.cer` file to install in Keychain Access. Make sure to save a backup copy of your private and public keys somewhere secure.

Expiration Date
2023/05/06

Created By

The certificate file (a file with a `.cer` extension) will be downloaded to your chosen location.

9. On your Mac, double-click the downloaded certificate file to install the certificate to your keychain. The certificate appears in the **My Certificates** category in **Keychain Access**, and begins with **iPhone Distribution**:



NOTE

Make a note of the full distribution certificate name in Keychain Access. It will be required when signing your app.

Create a distribution profile

A distribution provisioning profile enables your .NET MAUI iOS app to be digitally signed for release, so that it can be installed on an iOS device. A distribution provisioning profile contains an app ID and a distribution certificate.

Create an App ID

An App ID is required to identify the app that you are distributing:

1. In your Apple Developer Account, select the **Certificates, IDs & Profiles** tab.
2. On the **Certificates, Identifiers & Profiles** page, select the **Identifiers** tab.
3. On the **Identifiers** page, click the **+** button to create a new App ID.
4. On the **Register a new identifier** page, select the **App IDs** radio button before clicking the **Continue** button:

Certificates, Identifiers & Profiles

< All Identifiers

Register a new identifier

App IDs
Register an App ID to enable your app, app extensions, or App Clip 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.

Continue

5. On the **Register a new identifier** page, select **App** before clicking the **Continue** button:

Certificates, Identifiers & Profiles

< All Identifiers

Register a new identifier

Back

Continue

Select a type



App



App Clip

6. On the **Register an App ID** page, enter a description, and select either the **Explicit** or **Wildcard** Bundle ID radio button. Then, enter the Bundle ID for your app in reverse DS format:

Certificates, Identifiers & Profiles

< All Identifiers

Register an App ID

Back

Continue

Platform
iOS, macOS, tvOS, watchOS

App ID Prefix
C475GZ2XV9 (Team ID)

Description

MyMauiApp

You cannot use special characters such as @, &, *, ', ", -, .

Bundle ID Explicit Wildcard

com.companyname.mymauiapp

We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

IMPORTANT

The Bundle ID you enter must correspond to the Bundle ID defined in the **Info.plist** file in your app project.

7. On the **Register an App ID** page, select any capabilities that the app uses. Any capabilities must be configured both on this page and in the **Entitlements.plist** file in your app project.
8. On the **Register an App ID** page, click the **Continue** button.
9. On the **Confirm your App ID** page, click the **Register** button.

Create a provisioning profile

Once you've created a distribution certificate and an App ID you'll be able to create a provisioning profile:

1. In your Apple Developer Account, select the **Profiles** tab.
2. In the **Profiles** tab, click the + button to create a new profile.
3. In the **Register a New Provisioning Profile** page, select the **App Store** radio button before clicking the **Continue** button:

Certificates, Identifiers & Profiles

< All Profiles

Register a New Provisioning Profile

Continue

Development

- iOS App Development**
Create a provisioning profile to install development apps on test devices.
- tvOS App Development**
Create a provisioning profile to install development apps on tvOS test devices.
- macOS App Development**
Create a provisioning profile to install development apps on test devices.

Distribution

- Ad Hoc**
Create a distribution provisioning profile to install your app on a limited number of registered devices.
- tvOS Ad Hoc**
Create a distribution provisioning profile to install your app on a limited number of registered tvOS devices.
- App Store**
Create a distribution provisioning profile to submit your app to the App Store.

4. In the **Generate a Provisioning Profile** page, in the App ID drop down, select the App ID that you previously created before clicking the **Continue** button:

Certificates, Identifiers & Profiles

< All Profiles

Generate a Provisioning Profile

Back

Continue

Select Type > Configure > Generate > Download

Select an App ID

If you plan to use services such as Game Center, In-App Purchase, and Push Notifications, or want a Bundle ID unique to a single app, use an explicit App ID. Uploading apps to the App Store requires an explicit App ID. In the future, wildcard app IDs will no longer appear when creating an App Store provisioning profile.

App ID: 5 App IDs

5. In the **Generate a Provisioning Profile** page, select the radio button that corresponds to your distribution certificate before clicking the **Continue** button:

Certificates, Identifiers & Profiles

< All Profiles

Generate a Provisioning Profile

Back

Continue

Select Type > Configure > Generate > Download

Select Certificates

Select the certificates you wish to include in this provisioning profile. To use this profile to install an app, the certificate the app was signed with must be included.

- (iOS Distribution) May 06, 2023

6. In the **Generate a Provisioning Profile** page, enter a name for the provisioning profile before clicking the **Generate** button:

Certificates, Identifiers & Profiles

< All Profiles

Generate a Provisioning Profile

Back

Generate

Select Type > Configure > Generate > Download

Review, Name and Generate.

The name you provide will be used to identify the profile in the portal.

Provisioning Profile Name

MyMauiApp

Type
App Store

App ID
MyMauiApp(com.companyname.mymauiapp)

Certificates
1 Selected

NOTE

Make a note of the provisioning profile name, as it will be required when signing your app.

7. In the **Generate a Provisioning Profile** page, optionally click the **Download** button to download your provisioning profile.

NOTE

It's not necessary to download your provisioning profile now. Instead, you will do this later from Xcode.

Entitlements and capabilities

9/20/2022 • 2 minutes to read • [Edit Online](#)

In iOS, apps run in a sandbox that provides a set of rules that limit access between the app and system resources or user data. *Entitlements* are used to request the expansion of the sandbox to give your app additional capabilities. Any entitlements used by your app must be specified in the app's entitlements file. For more information about entitlements, see [Entitlements](#).

Apple provides *capabilities*, also known as *app services*, as a means of extending functionality and widening the scope of what iOS apps can do. Capabilities allow you to add a deeper integration with platform features to your app, such as integration with Siri. For more information about capabilities, see [Capabilities](#).

To extend the capabilities of your app, an entitlement must be provided in your app's *Entitlements.plist* file. Entitlements are a key/value pair, and generally only one entitlement is required per capability. In addition to specifying entitlements, the *Entitlements.plist* file is used to sign the app.

IMPORTANT

An *Entitlements.plist* file isn't linked to an Apple Developer Account. Therefore, any entitlements used by an app must also be specified when creating a provisioning profile for an app.

Add an Entitlements.plist file

To add a new entitlements file to your .NET Multi-platform App UI (.NET MAUI) app project, add a new XML file named *Entitlements.plist* to the *Platforms\iOS* folder of your app project. Then add the following XML to the file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
</dict>
</plist>
```

Set entitlements

Entitlements can be configured in Visual Studio by double-clicking the *Entitlements.plist* file to open it in the iOS Entitlements editor:

Entitlements:	Description:
Access WiFi Information Apple Pay AutoFill Credential Provider ClassKit Fonts Hotspot Configuration Wireless Accessory Configuration Multipath Near Field Communication Tag Reading	Near Field Communication Tag Reading enables your application to detect NFC tags. <input checked="" type="checkbox"/> Enable Near Field Communication Tag Reading

When a .NET MAUI iOS app uses entitlements, the *Entitlements.plist* file must be referenced from the `<CodesignEntitlement>` node within a `<PropertyGroup>`. For more information, see [Add code signing data to your app project](#).

Publish a .NET MAUI app for macOS

9/20/2022 • 2 minutes to read • [Edit Online](#)

When distributing your .NET Multi-platform App UI (.NET MAUI) app for macOS, you generate an `.app` or a `.pkg` file. An `.app` file is a self-contained app that can be run without installation, whereas a `.pkg` is an app packaged in an installer.

Publish an unsigned app

At this time, publishing is only supported through the .NET command line interface.

To publish your app, open a terminal and navigate to the folder for your .NET MAUI app project. Run the `dotnet build` command, providing the following parameters:

PARAMETER	VALUE
<code>-f</code> or <code>--framework</code>	The target framework, which is <code>net6.0-maccatalyst</code> .
<code>-c</code> or <code>--configuration</code>	The build configuration, which is <code>Release</code> .
<code>/p:CreatePackage</code>	An optional parameter that controls whether to create an <code>.app</code> or a <code>.pkg</code> . Use <code>true</code> for a <code>.pkg</code> .

WARNING

Attempting to publish a .NET MAUI solution will result in the `dotnet publish` command attempting to publish each project in the solution individually, which can cause issues when you've added other project types to your solution. Therefore, the `dotnet publish` command should be scoped to your .NET MAUI app project.

For example, use the following command to create an `.app`:

```
dotnet build -f:net6.0-maccatalyst -c:Release
```

Use the following command to create a `.pkg`:

```
dotnet build -f:net6.0-maccatalyst -c:Release /p:CreatePackage=true
```

Publishing builds the app, and then copies the `.app` or `.pkg` to the `bin/Release/net6.0-maccatalyst/maccatalyst-x64` folder.

For more information about the `dotnet publish` command, see [dotnet publish](#).

Run the unsigned app

By default, `.app` and `.pkg` files that are downloaded from the internet can't be run by double-clicking on them. For more information, see [Open a Mac app from an unidentified developer](#).

To ensure that a `.pkg` installs the app to your `Applications` folder, copy the `.pkg` to outside of your build artifacts folder and delete the `bin` and `obj` folders before double-clicking on the `.pkg`.

See also

- [GitHub discussion and feedback: .NET MAUI macOS target publishing/archiving](#)
- [Open apps safely on your Mac](#)

Publish a .NET MAUI app for Windows

9/20/2022 • 5 minutes to read • [Edit Online](#)

When distributing your .NET Multi-platform App UI (.NET MAUI) app for Windows, you can publish the app and its dependencies to a folder for deployment to another system. You can also package the app into an MSIX package, which has numerous benefits for the users installing your app. For more information about the benefits of MSIX, see [What is MSIX?](#)

.NET MAUI currently only allows publishing an MSIX package. You can't yet publish a Windows executable file for distribution.

Create a signing certificate

You must use a signing certificate for use in publishing your app. This certificate is used to sign the MSIX package. The following steps demonstrate how to create and install a self-signed certificate with PowerShell:

NOTE

When you create and use a self-signed certificate only users who install and trust your certificate can run your app. This is easy to implement for testing but it may prevent additional users from installing your app. When you are ready to publish your app we recommend that you use a certificate issued by a trusted source. This system of centralized trust helps to ensure that the app ecosystem has levels of verification to protect users from malicious actors.

1. Open a PowerShell terminal and navigate to the directory with your project.
2. Use the `New-SelfSignedCertificate` command to generate a self-signed certificate.

The `<PublisherName>` value is displayed to the user when they install your app, supply your own value and omit the `< >` characters. You can set the `FriendlyName` parameter to any string of text you want.

```
New-SelfSignedCertificate -Type Custom `  
    -Subject "CN=<PublisherName>" `  
    -KeyUsage DigitalSignature `  
    -FriendlyName "My temp dev cert" `  
    -CertStoreLocation "Cert:\CurrentUser\My" `  
    -TextExtension @("2.5.29.37={text}1.3.6.1.5.5.7.3.3", "2.5.29.19={text}")
```

3. Use the following PowerShell command to query the certificate store for the certificate that was created:

```
Get-ChildItem "Cert:\CurrentUser\My" | Format-Table Subject, FriendlyName, Thumbprint
```

You should see results similar to the following output:

Thumbprint	Subject	FriendlyName
DE8B962E7BF797CB48CCF66C8BCACE65C6585E2F	CN=1f23fa36-2a2f-475e-a69e-3a14fe56ed4	
A6CA34FD0BA6B439787391F51C87B1AD0C9E7FAE	CN=someone@microsoft.com	
94D93DBC97D4F7E4364A215F15C6ACFEFC71E569	CN=localhost	ASP.NET Core HTTPS
development certificate		
F14211566DACE867DA0BF9C2F9C47C01E3CF1D9B	CN=john	
568027317BE8EE5E6AACDE5079D2D76EC46EB88	CN=e1f823e2-4674-03d2-aaad-21ab23ad84ae	
DC602EE83C95FEDF280835980E22306067EFCA96	CN=John Smith, OU=MSE, OU=Users, DC=com	
07AD38F3B646F5AAC16F2F2570CAE40F4842BBE0	CN=Contoso	My temp dev cert

4. The **Thumbprint** of your certificate will be used later, copy it to your clipboard. It's the **Thumbprint** value whose entry matches the **Subject** and **FriendlyName** of your certificate.

For more information, see [Create a certificate for package signing](#).

Configure the project build settings

The project file is a good place to put Windows-specific build settings. You may not want to put some settings into the project file, such as passwords. The settings described in this section can be passed on the command line with the `/p:name=value` format. If the setting is already defined in the project file, a setting passed on the command line will override the project setting.

Add the following `<PropertyGroup>` node to your project file. This property group is only processed when the target framework is Windows and the configuration is set to `Release`. This config section runs whenever a build or publish in `Release` mode.

```
<PropertyGroup Condition="&lt;MSBuild&gt;::GetTargetPlatformIdentifier('$(TargetFramework)') == 'windows' and '$(Configuration)' == 'Release'">
  <AppxPackageSigningEnabled>true</AppxPackageSigningEnabled>
  <PackageCertificateThumbprint>A10612AF095FD8F8255F4C6691D88F79EF2B135E</PackageCertificateThumbprint>
</PropertyGroup>
<PropertyGroup Condition="&lt;MSBuild&gt;::GetTargetPlatformIdentifier('$(TargetFramework)') == 'windows' and '$(RuntimeIdentifierOverride)' != ''">
  <RuntimeIdentifier>$(RuntimeIdentifierOverride)</RuntimeIdentifier>
</PropertyGroup>
```

Replace the `<PackageCertificateThumbprint>` property value with the certificate thumbprint you previously generated. Alternatively, you can remove this setting from the project file and provide it on the command line. For example: `/p:PackageCertificateThumbprint=A10612AF095FD8F8255F4C6691D88F79EF2B135E`.

The second `<PropertyGroup>` in the example is required to work around a bug in the Windows SDK. For more information about the bug, see [WindowsAppSDK Issue #2940](#).

Publish

To publish your app, open the **Developer Command Prompt for VS 2022** terminal and navigate to the folder for your .NET MAUI app project. Run the `dotnet publish` command, providing the following parameters:

PARAMETER	VALUE
<code>-f net6.0-windows{version}</code>	The target framework, which is a Windows TFM, such as <code>net6.0-windows10.0.19041.0</code> . Ensure that this value is identical to the value in the <code><TargetFrameworks></code> node in your <code>.csproj</code> file.

PARAMETER	VALUE
<code>-c Release</code>	Sets the build configuration, which is <code>Release</code> .
<code>/p:RuntimeIdentifierOverride=win10-x64</code> - or - <code>/p:RuntimeIdentifierOverride=win10-x86</code>	Avoids the bug detailed in WindowsAppSDK Issue #2940 . Choose the <code>-x64</code> or <code>-x86</code> version of the parameter based on your target platform.

WARNING

Attempting to publish a .NET MAUI solution will result in the `dotnet publish` command attempting to publish each project in the solution individually, which can cause issues when you've added other project types to your solution. Therefore, the `dotnet publish` command should be scoped to your .NET MAUI app project.

For example:

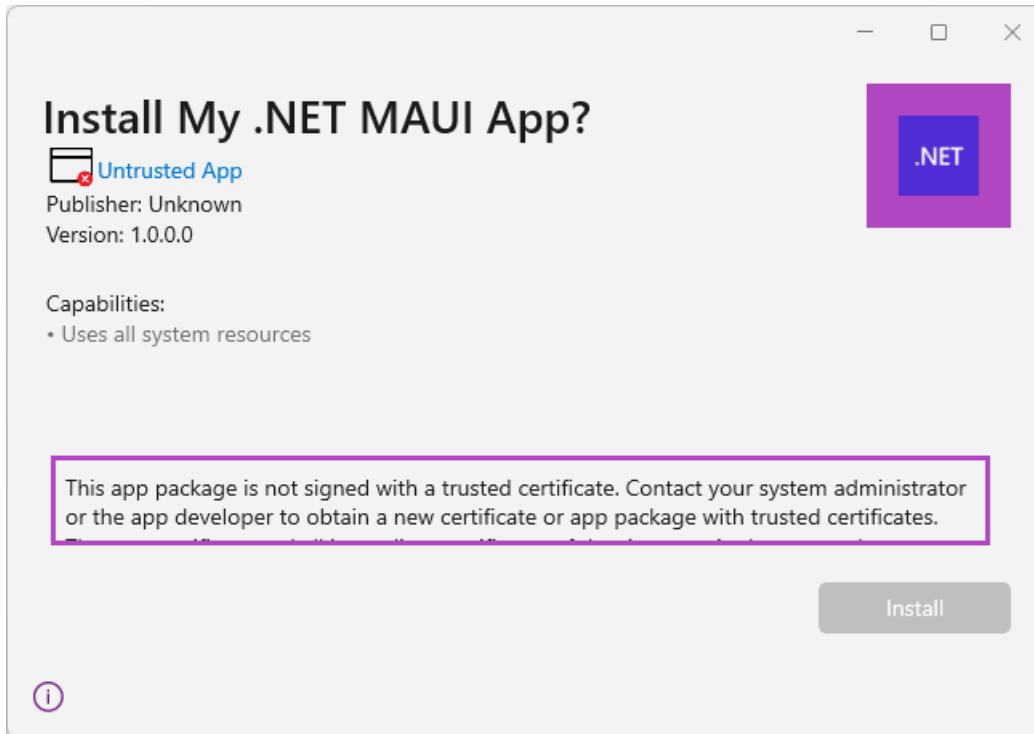
```
dotnet publish -f net6.0-windows10.0.19041.0 -c Release /p:RuntimeIdentifierOverride=win10-x64
```

Publishing builds and packages the app, copying the signed package to the `bin\Release\net6.0-windows10.0.19041.0\win10-x64\appPackages\<appname>\` folder. `<appname>` is a folder named after both your project and version. In this folder, there's an `msix` file, and that's the app package.

For more information about the `dotnet publish` command, see [dotnet publish](#).

Installing the app

To install the app, it must be signed with a certificate that you already trust. If it isn't, Windows won't let you install the app. You'll be presented with a dialog similar to the following, with the Install button disabled:

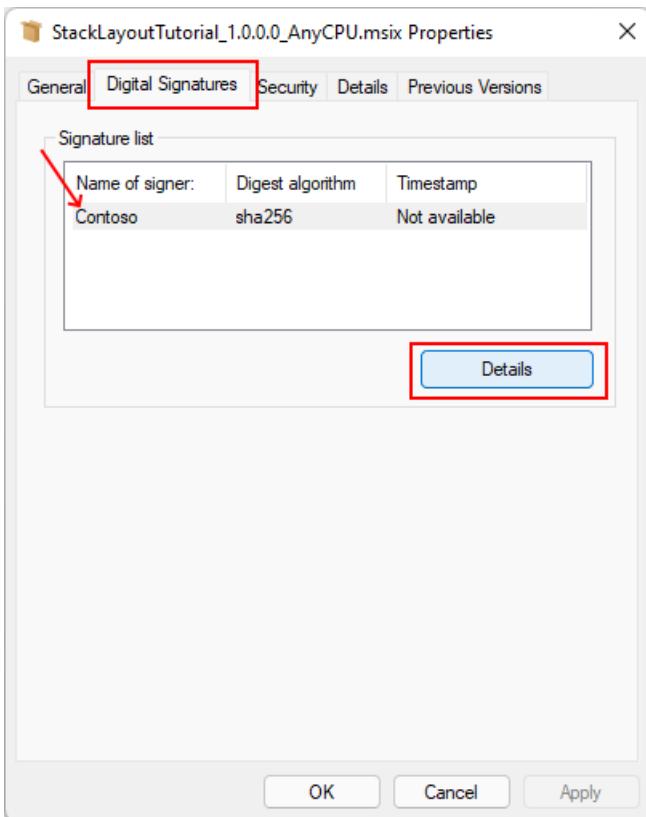


Notice that in the previous image, the Publisher was "unknown."

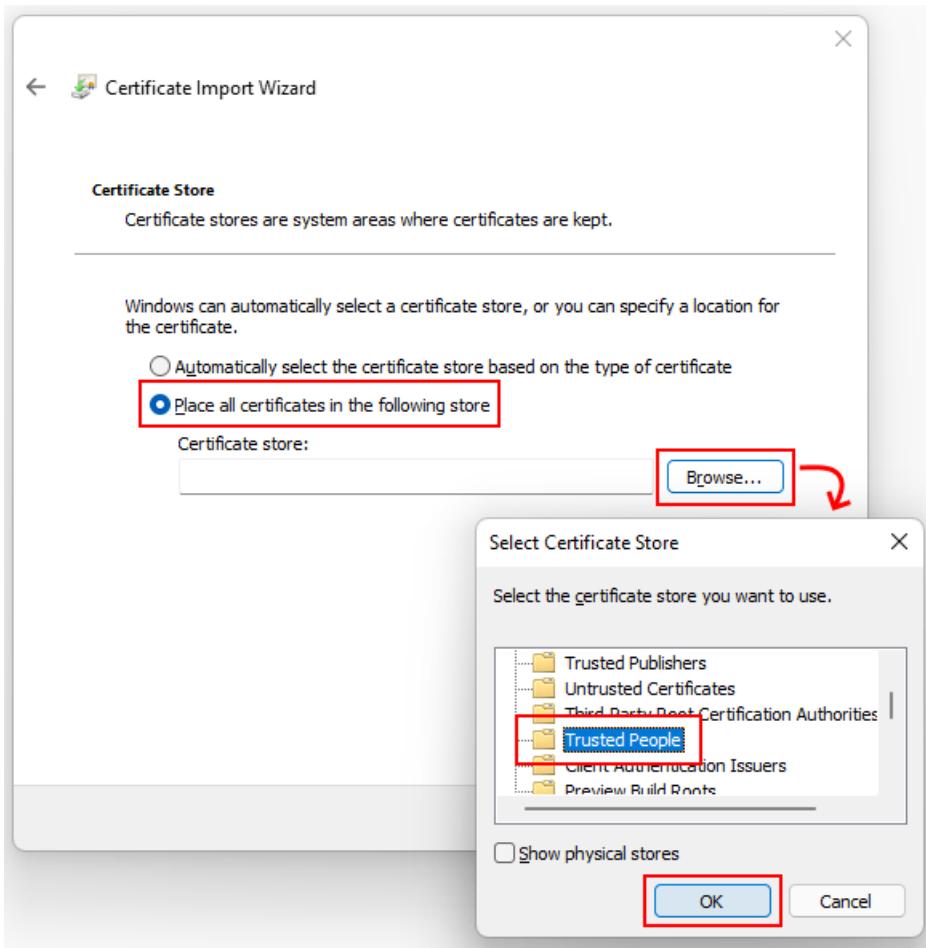
To trust the certificate of app package, perform the following steps:

1. Right-click on the `.msix` file and choose **Properties**.

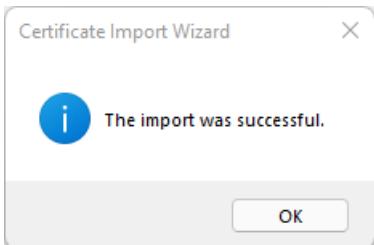
2. Select the **Digital Signatures** tab.
3. Choose the certificate then press **Details**.



4. Select **View Certificate**.
5. Select **Install Certificate...**
6. Choose **Local Machine** then select **Next**.
If you're prompted by User Account Control to **Do you want to allow this app to make changes to your device?**, select **Yes**.
7. In the **Certificate Import Wizard** window, select **Place all certificates in the following store**.
8. Select **Browse...** and then choose the **Trusted People** store. Select **OK** to close the dialog.

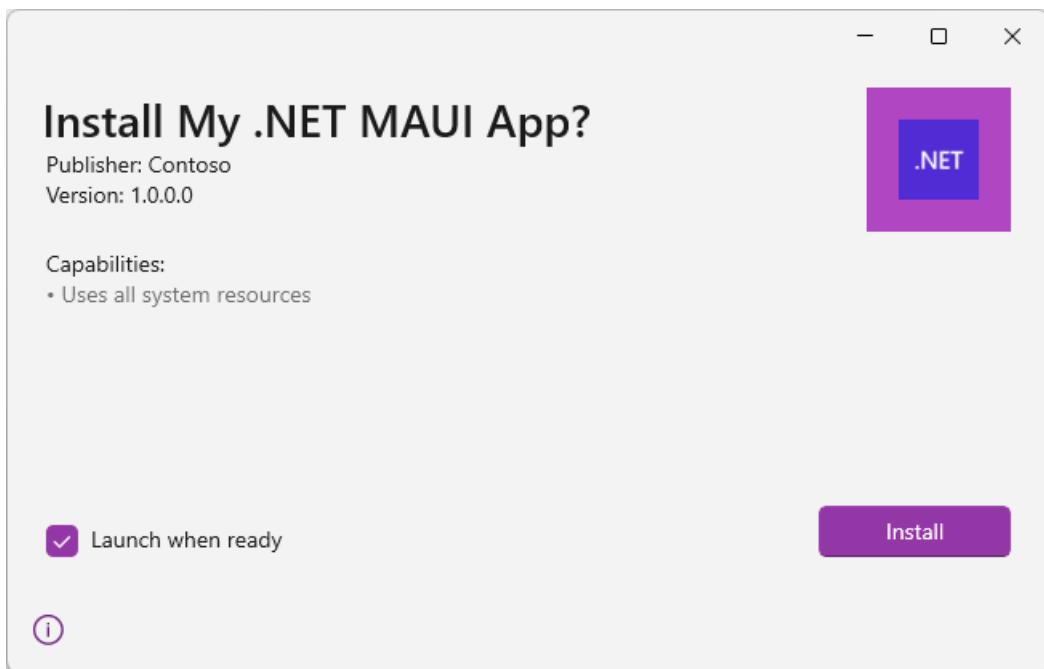


9. Select **Next** and then **Finish**. You should see a dialog that says: **The import was successful**.



10. Select **OK** on any window opened as part of this process, to close them all.

Now, try opening the package file again to install the app. You should see a dialog similar to the following, with the Publisher correctly displayed:



Select the **Install** button if you would like to install the app.

Current limitations

The following list describes the current limitations with publishing and packaging:

1. The published app doesn't work if you try to run it directly with the executable file out of the publish folder.
2. The way to run the app is to first install it through the packaged *MSIX* file.

See also

- [GitHub discussion and feedback: .NET MAUI Windows target publishing/archiving](#)

Troubleshooting known issues

9/20/2022 • 3 minutes to read • [Edit Online](#)

This article describes some of the known issues with .NET MAUI, and how you can solve or work around them. The [.NET MAUI code repository](#) also details some known issues.

Templates are missing

If you've installed Visual Studio 2022 and the [.NET Multi-platform App UI development](#) workload, but the .NET MAUI templates are missing, you most likely have a conflict with a .NET 7 preview version. To see if .NET 7 is being resolved as your current version of .NET, perform the following steps:

1. Open a terminal.
2. Run the `dotnet --version` command.

If the result starts with `7.0`, you're running in the context of .NET 7. Use one of the fixes below.

TIP

You can see all versions of .NET installed with the `dotnet --info` command.

Fix (1)

Uninstall the .NET 7 preview.

Fix (2)

Use a `global.json` config file in the folder where you'll create the project. This config file can force the context of .NET 6, allowing you to create a new project:

1. Open a terminal and navigate to a folder where you want to create a project.
2. Run the following command: `dotnet new globaljson --sdk-version 6.0.300`
3. Run `dotnet new maui --list` to show a list of projects you can create. For example, you may see the following output:

```
dotnet new maui --list

These templates matched your input: 'maui'

Template Name           Short Name     Language  Tags
-----
----- .NET MAUI App          maui          [C#]      MAUI/Android/iOS/macOS/Mac
Catalyst/Windows/Tizen
----- .NET MAUI Blazor App   maui-blazor   [C#]      MAUI/Android/iOS/macOS/Mac
Catalyst/Windows/Tizen/Blazor
----- .NET MAUI Class Library mauilib       [C#]      MAUI/Android/iOS/macOS/Mac
Catalyst/Windows/Tizen
----- .NET MAUI ContentPage (C#) maui-page-csharp [C#]      MAUI/Android/iOS/macOS/Mac
Catalyst/WinUI/Tizen/Xaml/Code
----- .NET MAUI ContentPage (XAML) maui-page-xaml   [C#]      MAUI/Android/iOS/macOS/Mac
Catalyst/WinUI/Tizen/Xaml/Code
----- .NET MAUI ContentView (C#)  maui-view-csharp  [C#]      MAUI/Android/iOS/macOS/Mac
Catalyst/WinUI/Tizen/Xaml/Code
----- .NET MAUI ContentView (XAML) maui-view-xaml   [C#]      MAUI/Android/iOS/macOS/Mac
Catalyst/WinUI/Tizen/Xaml/Code
----- .NET MAUI ResourceDictionary (XAML) maui-dict-xaml  [C#]      MAUI/Android/iOS/macOS/Mac
Catalyst/WinUI/Xaml/Code
```

4. Next, create a new .NET MAUI project with the `dotnet new` command, using either `maui` or `maui-blazor` as the project type:

```
dotnet new maui
```

5. Open the project in Visual Studio.

Until this issue is resolved by Microsoft, you'll need to do this for every project.

ERROR Platform version is not present.

Visual Studio may not be resolving the required workloads if you try to compile a project and receive an error similar to the following:

```
Platform version is not present for one or more target frameworks, even though they have specified a
platform: net6.0-android, net6.0-ios, net6.0-maccatalyst
```

This may be caused by having the .NET 7 preview installed. To see if .NET 7 is being resolved as your current version of .NET, perform the following steps:

1. Open a terminal.
2. Run the `dotnet --version` command.

If the result shows a `7.0.xxx` value, you're running in the context of .NET 7. Use one of the fixes below.

Fix (1)

Uninstall the .NET 7 preview.

Fix (2)

Use a `global.json` config file in the same folder as your current project. This config file can force the context of .NET 6, allowing you to keep a .NET 7 preview version installed:

1. Close Visual Studio.

2. Open a terminal and navigate to the folder where your project is located.
3. Run the following command: `dotnet new globaljson --sdk-version 6.0.300`
4. Reopen the project in Visual Studio.

Until this issue is resolved by Microsoft, you'll need to do this for every project.

The `WINDOWS #if` directive is broken

The `WINDOWS` definition doesn't resolve correctly in the latest release of .NET MAUI. To work around this issue, add the following entry to the `<PropertyGroup>` element of your project file.

```
<DefineConstants Condition="$(MSBuild)::GetTargetPlatformIdentifier('$(TargetFramework')') == 'windows'">$(_DefineConstants);WINDOWS</DefineConstants>
```

The definitions that identify a specific version of Windows will still be missing.

ERROR Type or namespace 'Default' does not exist

When using the [Contacts API](#), you may see the following error related to iOS and macOS:

```
The type or namespace name 'Default' does not exist in the namespace 'Contacts' (are you missing an assembly reference?)
```

The iOS and macOS platforms contain a root namespace named `Contacts`. This conflict causes a conflict for those platforms with the `Microsoft.Maui.ApplicationModel.Communication` namespace, which contains a `Contacts` type. The `Microsoft.Maui.ApplicationModel.Communication` namespace is automatically imported by the `<ImplicitUsings>` setting in the project file.

To write code that also compiles for iOS and macOS, fully qualify the `Contacts` type. Alternatively, provide a `using` directive at the top of the code file to map the `Communication` namespace:

```
using Communication = Microsoft.Maui.ApplicationModel.Communication;

// Code that uses the namespace:
var contact = await Communication.Contacts.Default.PickContactAsync();
```