* Contents

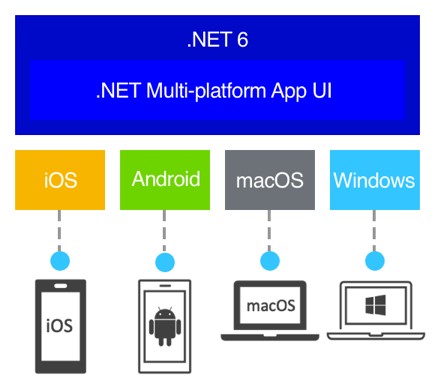
.NET MAUI

What is .NET Multi-platform App UI?

# Supported platforms Get started Installation Build your first app Migrate from Xamarin.Forms Fundamentals App startup Single project User interface Customize controls

|  |
| --- |
| What is .NET MAUI?  7/8/2021 • 5 minutes to read • Edit Online |

.NET Multi-platform App UI (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, test, and business logic across platforms.

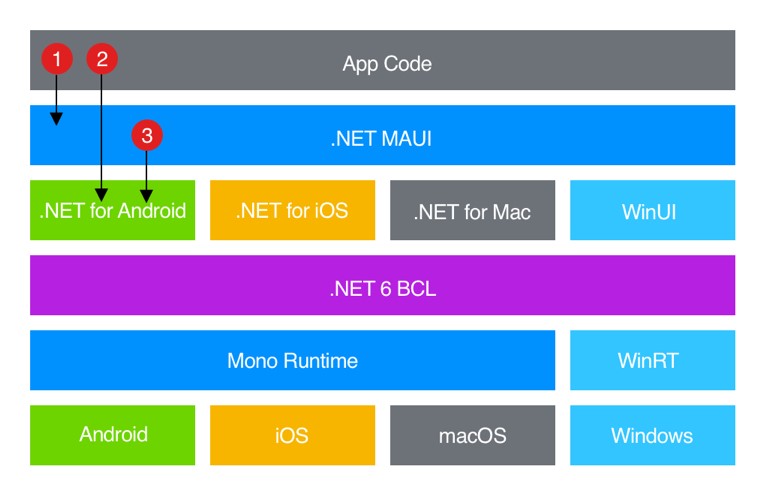
# How .NET MAUI works

.NET MAUI unifies Android, iOS, macOS, and Windows APIs into a single API that allows a write-once runanywhere 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 (WinUI) 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, WinRT performs the same role, except it's optimized for the Windows platform.

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), 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 Library (WinUI) 3 to create native apps that can target t[he Windows desktop](https://docs.microsoft.com/en-us/windows/apps/winui/) and the Universal Windows Platform (UWP). For more information about WinUI, 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.

Essential cross-platform APIs for accessing native device features. These APIs enable apps to access things like the GPS, the accelerometer, and battery and network states. For more information, see .NET MAUI essentials.

A cross-platform graphics library, that provides a common API to target multiple platforms, which enables you to share your 2D drawing code between platforms, or mix and match graphics implementations with a single app.

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.

## .NET MAUI essentials

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

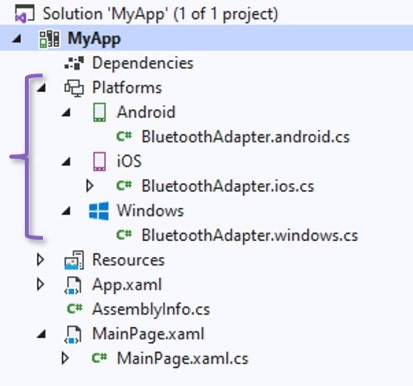
## .NET MAUI single project

.NET MAUI apps typically consist of a single project that can target Android, iOS, macOS, and Windows. This delivers the following benefits:

One project that targets multiple platforms and devices.

One location to manage resources such as fonts and images.

Multi-targeting to organize platform-specific code.



For more information about .NET MAUI single project, see .NET MAUI single project.

## .NET 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.

|  |
| --- |
| .NET MAUI supported platforms  6/16/2021 • 2 minutes to read • Edit Online |

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

Android 5.0 (API 21) or higher.

iOS 10 or higher. macOS 11 (Big Sur) or higher.

Windows desktop and the Universal Windows Platform (UWP), using Windows UI Library (WinUI) 3.

.NET MAUI apps for Android, iOS, and Windows can be built in Visual Studio. However, a networked Mac is required for iOS development using the latest version of Xcode and the minimum version of macOS required by Apple.

.NET MAUI apps for Android, iOS, and macOS can be built in Visual Studio for Mac.

# Additional platform support

.NET MAUI supports additional platforms beyond iOS, Android, and Windows:

Tizen, supported by Samsung.

Linux, supported by the community.

|  |
| --- |
| .NET MAUI installation  7/9/2021 • 3 minutes to read • Edit Online |

**IMPORTANT**

These requirements will change as new preview releases of Visual Studio and .NET MAUI are released.

To create .NET Multi-platform App UI (MAUI) apps, you currentl[y require .NET 6 Preview 5 with](https://visualstudio.microsoft.com/vs/preview/) .NET MAUI and the platform SDKs for Android, iOS, macOS, tvOS, and Mac Catalyst.

To create .NET MAUI apps in Visual Studio, you'll also need Visual Studio 16.11 Preview 2 with the following workloads installed:

Mobile development with .NET

Universal Windows Platform development

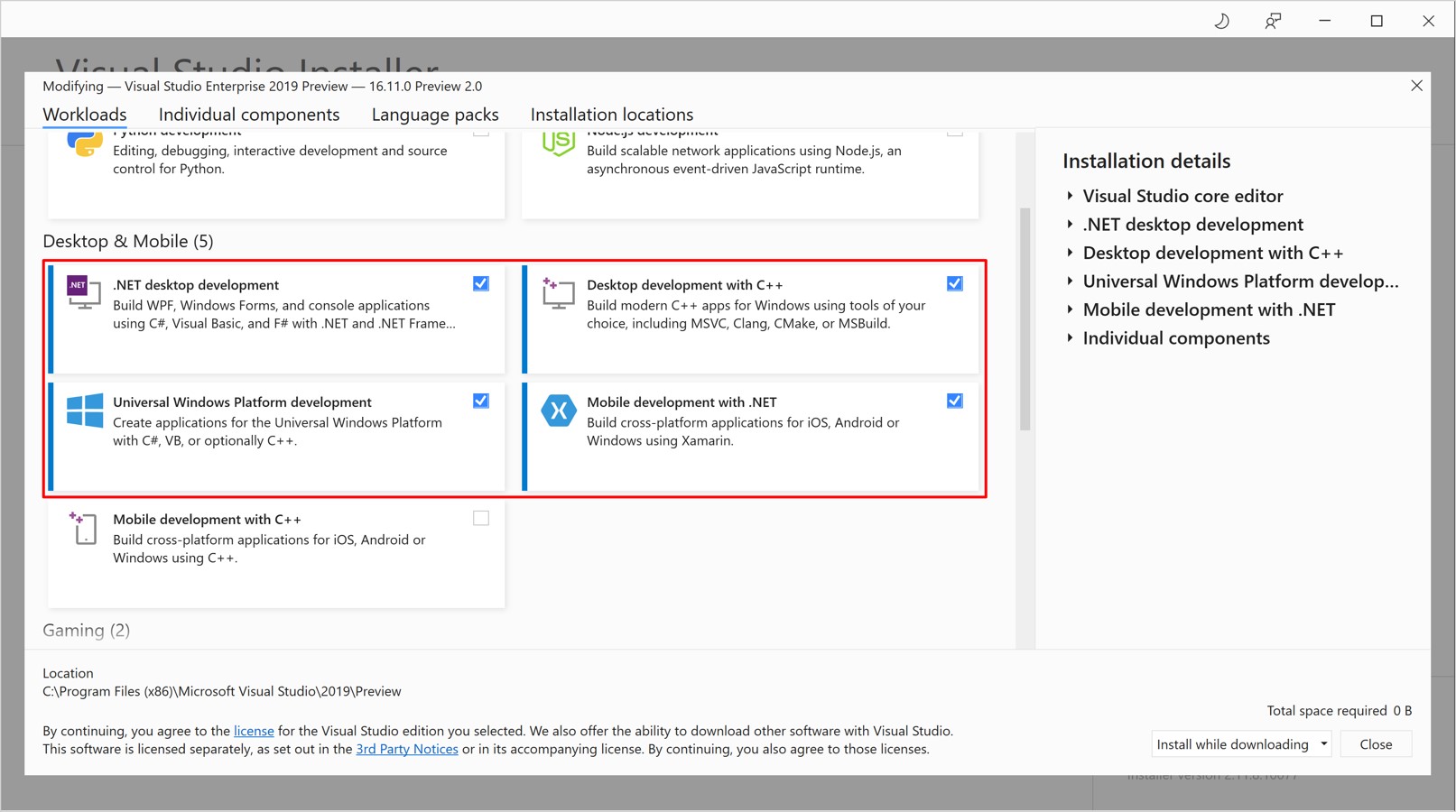
Desktop development with C++

.NET Desktop Development

ASP.NET and web development (required for Blazor Desktop and the

BlazorWebView

control)



In addition, you must currently install the following Visual Studio extensions to create apps that target Windows UI Library (WinUI) 3:

Project Reunion (Preview)

Single-project MSIX Packaging Tools

[For more information about the required workloads and components for WinUI 3 development, see Required](https://docs.microsoft.com/en-us/windows/apps/project-reunion/set-up-your-development-environment#required-workloads-and-components) workloads and components.

|  |  |  |
| --- | --- | --- |
| WebView | or | BlazorWebView |

To use the controls on Windows you need to install the WebView2 package:

[Microsoft Edge WebView2 installer](https://developer.microsoft.com/microsoft-edge/webview2/)

# Install .NET 6 Preview 5

To verify your development environment, and install any missing components, use the maui-check utility. Install this utility using the following .NET CLI command:

dotnet tool install -g redth.net.maui.check

|  |
| --- |
| maui-check |

Then, run :

maui-check

|  |
| --- |
| maui-check |

If any tools and SDKs required by .NET MAUI are missing, will install them. The example below shows the output generated if the tools and SDKs required by .NET MAUI are already installed:

|  |
| --- |
| \_ \_ \_\_\_\_\_ \_\_\_\_\_ \_\_ \_\_ \_ \_ \_ \_\_\_  | \ | | | \_\_\_\_| |\_ \_| | \/ | / \ | | | | |\_ \_|  | \| | | \_| | | | |\/| | / \_ \ | | | | | |  \_ | |\ | | |\_\_\_ | | | | | | / \_\_\_ \ | |\_| | | |  (\_) |\_| \\_| |\_\_\_\_\_| |\_| |\_| |\_| /\_/ \\_\ \\_\_\_/ |\_\_\_|  \* .NET MAUI Check v0.5.6.0 \*  ──────────────────────────────────────────────────────────────────────────────────────────────────────────── ────────────  This tool will attempt to evaluate your .NET MAUI development environment.  If problems are detected, this tool may offer the option to try and fix them for you, or suggest a way to fix them yourself.  Thanks for choosing .NET MAUI!  ────────────────────────────────────────────────────────────────────────────────────────────────────────────  ────────────  » Synchronizing configuration... ok  » Scheduling appointments... ok  > OpenJDK 11.0 Checkup...   * 11.0.10 (C:\Program Files\Microsoft\jdk-11.0.10.9-hotspot\bin\..) - 1.8.0-25 (C:\Program Files\Android\Jdk\microsoft\_dist\_openjdk\_1.8.0.25)   > Visual Studio 16.10.0 Checkup...   * 16.9.6 * 16.11.0-pre.2.0 - C:\Program Files (x86)\Microsoft Visual Studio\2019\Preview * 15.9.18   > Android SDK Checkup...   * emulator (30.6.5) * build-tools;30.0.2 (30.0.2) * platforms;android-30 (3) * system-images;android-30;google\_apis;x86 (9) * platform-tools (30.0.4)   > Android Emulator Checkup...   * Emulator: pixel\_2\_xl\_pie\_9\_0\_api\_28 found.   > .NET SDK Checkup...   * 2.1.200 - C:\Program Files\dotnet\sdk\2.1.200 * 2.1.201 - C:\Program Files\dotnet\sdk\2.1.201 * 2.1.202 - C:\Program Files\dotnet\sdk\2.1.202 * 2.1.400 - C:\Program Files\dotnet\sdk\2.1.400 * 2.1.402 - C:\Program Files\dotnet\sdk\2.1.402 * 2.1.403 - C:\Program Files\dotnet\sdk\2.1.403 * 2.1.500 - C:\Program Files\dotnet\sdk\2.1.500 * 2.1.502 - C:\Program Files\dotnet\sdk\2.1.502 * 2.1.504 - C:\Program Files\dotnet\sdk\2.1.504 |

|  |  |
| --- | --- |
| * 2.1.505 - C:\Program Files\dotnet\sdk\2.1.505 * 2.1.507 - C:\Program Files\dotnet\sdk\2.1.507 * 2.1.509 - C:\Program Files\dotnet\sdk\2.1.509 * 2.1.602 - C:\Program Files\dotnet\sdk\2.1.602 * 2.1.700 - C:\Program Files\dotnet\sdk\2.1.700 * 2.1.801 - C:\Program Files\dotnet\sdk\2.1.801 * 5.0.203 - C:\Program Files\dotnet\sdk\5.0.203 * 5.0.400-preview.21277.10 - C:\Program Files\dotnet\sdk\5.0.400-preview.21277.10 * 6.0.100-preview.5.21302.13 - C:\Program Files\dotnet\sdk\6.0.100-preview.5.21302.13   > .NET SDK - Workload Deduplication Checkup...  > .NET SDK - EnableWorkloadResolver.sentinel Checkup...   * C:\Program Files (x86)\Microsoft Visual   Studio\2019\Preview\MSBuild\Current\Bin\SdkResolvers\Microsoft.DotNet.MSBuildSdkResolver\EnableWorkloadResol ver.sentinel exists.  > .NET SDK - Workloads (6.0.100-preview.5.21302.13) Checkup...   * microsoft-android-sdk-full (Microsoft.NET.Sdk.Android.Manifest-6.0.100 : 30.0.100-preview.5.28) installed. * microsoft-ios-sdk-full (Microsoft.NET.Sdk.iOS.Manifest-6.0.100 : 14.5.100-preview.5.894) installed. * microsoft-maccatalyst-sdk-full (Microsoft.NET.Sdk.MacCatalyst.Manifest-6.0.100 : 14.5.100-preview.5.894) installed. * microsoft-tvos-sdk-full (Microsoft.NET.Sdk.tvOS.Manifest-6.0.100 : 14.5.100-preview.5.894) installed. * microsoft-macos-sdk-full (Microsoft.NET.Sdk.macOS.Manifest-6.0.100 : 11.3.100-preview.5.894) installed.   > .NET SDK - Packs (6.0.100-preview.5.21302.13) Checkup... - Microsoft.Maui.Templates (6.0.100-preview.5.794) installed. - Microsoft.Android.Sdk (30.0.100-preview.5.28) installed.   * Microsoft.Android.Sdk.BundleTool (30.0.100-preview.5.28) installed. * Microsoft.Android.Ref (30.0.100-preview.5.28) installed. - Microsoft.Android.Templates (30.0.100-preview.5.28) installed. * Microsoft.iOS.Sdk (14.5.100-preview.5.894) installed. - Microsoft.iOS.Windows.Sdk (14.5.100-preview.5.894) installed. - Microsoft.iOS.Ref (14.5.100-preview.5.894) installed. - Microsoft.iOS.Templates (14.5.100-preview.5.894) installed. - Microsoft.MacCatalyst.Sdk (14.5.100-preview.5.894) installed. * Microsoft.MacCatalyst.Ref (14.5.100-preview.5.894) installed. - Microsoft.MacCatalyst.Templates (14.5.100-preview.5.894) installed. * Microsoft.tvOS.Sdk (14.5.100-preview.5.894) installed. * Microsoft.tvOS.Ref (14.5.100-preview.5.894) installed. - Microsoft.tvOS.Templates (14.5.100-preview.5.894) installed. - Microsoft.macOS.Sdk (11.3.100-preview.5.894) installed. * Microsoft.macOS.Ref (11.3.100-preview.5.894) installed. - Microsoft.macOS.Templates (11.3.100-preview.5.894) installed.   ────────────────────────────────────────────────────────────────────────────────────────────────────────────  ────────────   * Congratulations, everything looks great! | |
| Build your first .NET MAUI app  7/8/2021 • 3 minutes to read • Edit Online |

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

**NOTE**

Visual Studio for Mac support will arrive in a future release.

# Prerequisites

An environment that has been configured for .NET MAUI development, using the maui-check tool. For more information, see Install .NET 6 Preview 5.

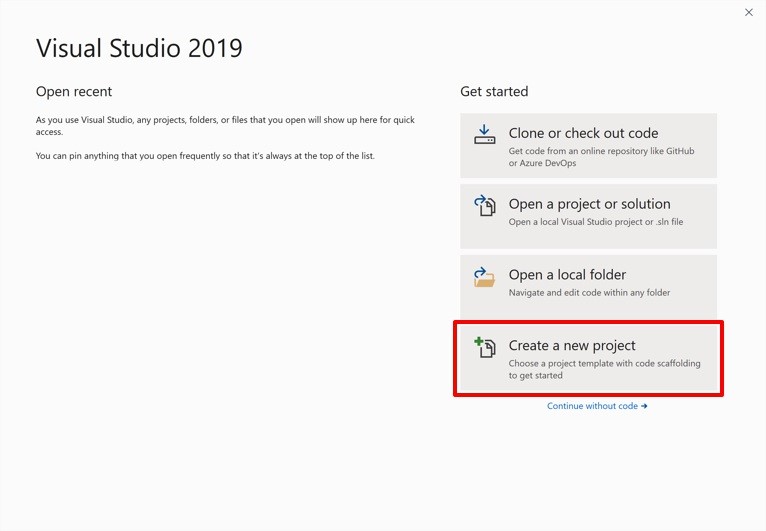
[Visual Studio 2019 (build 16.11 Preview 2 or greater) or Visual Studio 2022 (any edition), with the required workloads. For more information, see Installation.](https://docs.microsoft.com/en-us/xamarin/android/get-started/installation/android-emulator/)

A configured Android emulator. For more information about creating an Android emulator, see Android emulator setup.

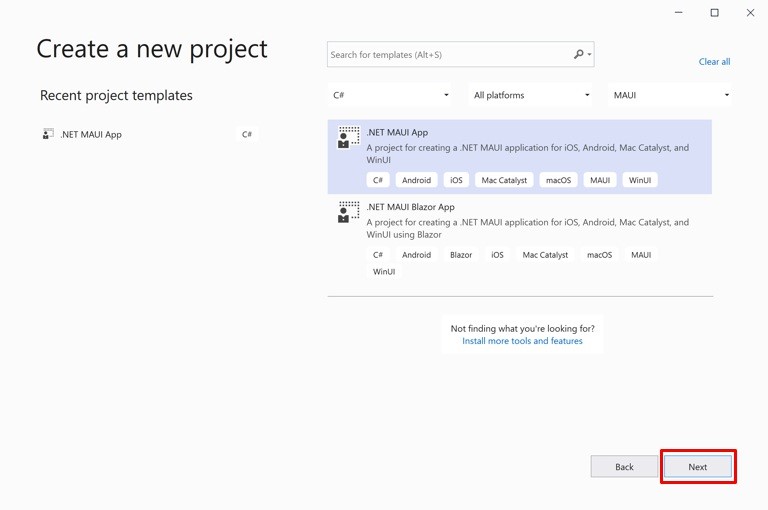
# Get started with Visual Studio 2019

In this tutorial, you'll create your first .NET MAUI app in Visual Studio 2019, and run it on an Android emulator:

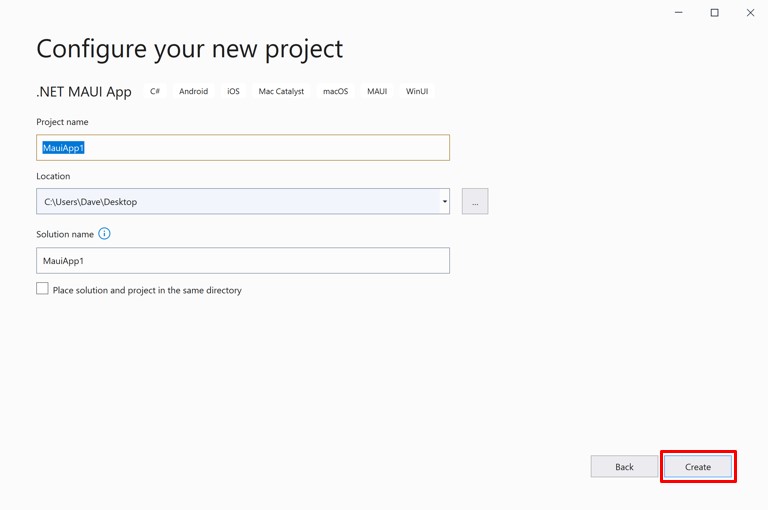
1. Launch Visual Studio 2019 build 16.11 (Preview 2 or greater), and in the start window click Create a new project to create a new project:



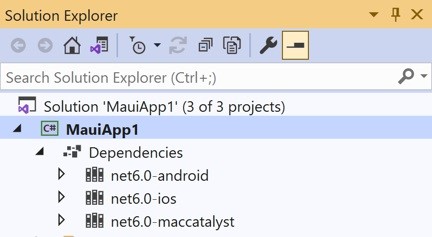
1. In the Create a new project window, select MAUI in the Project type drop-down, select the .NET MAUI App template, and click the Next button:



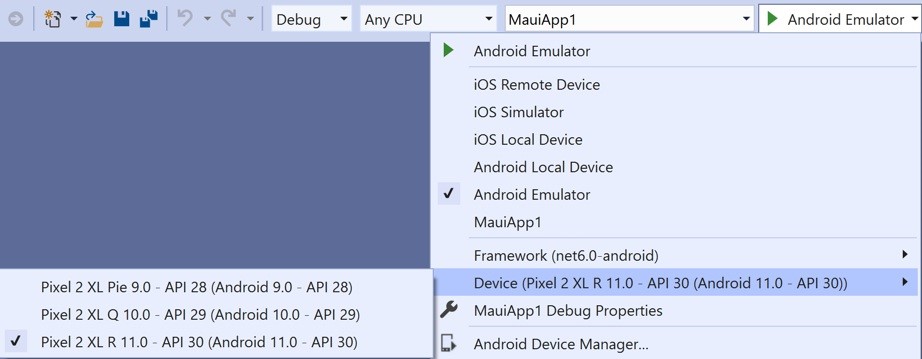
1. In the Configure your new project window, name your project, choose a suitable location for it, and click the Create button:



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



1. In the Visual Studio toolbar, select the drop-down next to the Start button (the triangular button that resembles a Play button), select Android Emulator, and then select the emulator you'd like to deploy the app to:



1. In the Visual Studio toolbar, press the Start button to launch the app in your chosen Android emulator.
2. 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.



# Build and debug iOS apps

To build and debug .NET 6 iOS apps from Visual Studio 2019 you must manually install the .NET 6 SDK and iOS workloads on both Windows and macOS (your Mac build host).

If, while connecting Visual Studio to your Mac through Xamarin Mac Agent (XMA), you are prompted to install a different version of the SDK, you can ignore the prompt since it refers to a legacy version of XMA.

**NOTE**

Visual Studio 2019 can only currently deploy .NET MAUI iOS apps to the iOS simulator, and not to physical devices.

# Prerequisites

An environment that has [been configured for .NET](https://docs.microsoft.com/en-us/xamarin/android/get-started/installation/android-emulator/) MAUI development, using the maui-check tool. For more information, see Install .NET 6 Preview 5.

A configured simulator or emulator for your chosen platform. For more information about creating an Android emulator, see Android emulator setup.

# Get started with .NET command-line interface

In this tutorial, you'll create and run your first .NET MAUI app using the .NET command-line interface (CLI):

1. In the .NET CLI, create a new .NET MAUI app:

dotnet new maui -n HelloMauiPreview

1. In the .NET CLI, change directory to the newly created project:
   1. HelloMauiPreview
2. In the .NET CLI, change directory to the single project and restore its dependencies:
   1. HelloMauiPreviewdotnet restore
3. In the .NET CLI, build and launch the app on your chosen platform:

dotnet build -t:Run -f net6.0-android dotnet build -t:Run -f net6.0-ios dotnet build -t:Run -f net6.0-maccatalyst

## NOTE

These commands will launch the app on the default platform device, if one can be found. On Android, it's recommended to start an emulator before building and launching your app.

# Select an iOS simulator

It's possible to specify which simulator is launched and used for net6.0-ios by specifying the \_DeviceName MSBuild property:

dotnet build -t:Run -f net6.0-ios -p:\_DeviceName=:v2:udid=<UDID>

|  |
| --- |
| simctl list |

You can retrieve a list of possible unique device id (UDID) values by executing the command:

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

The default iOS simulator will be launched if you don't specify a UDID.

|  |
| --- |
| Migrate your app from Xamarin.Forms to .NET  MAUI  6/18/2021 • 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 (MAUI). However, you 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.

**IMPORTANT**

Do not currently migrate your production apps to .NET MAUI.

The process for migrating a Xamarin.Forms app to .NET MAUI is expected to be:

1. Use the .NET upgrade assistant for .NET MAUI to migrate your Xamarin.Forms projects to .NET MAUI single project, and perform well-known code namespace changes.
2. Update any dependencies to .NET 6 and .NET MAUI compatible versions.
3. Register any compatibility services or renderers.
4. Build and fix any issues.
5. Run the converted app and verify that it functions correctly.

# Port an app ex[ample](https://docs.microsoft.com/en-us/samples/xamarin/xamarin-forms-samples/userinterface-buttondemos/)

This example ports the Button Demos sample. The process is as follows:

1. Create a new, blank, multi-targeted .NET MAUI project:

dotnet new maui -n ButtonDemos

1. Restore the dependencies for the newly created project:

cd ButtonDemos dotnet restore

## NOTE

If you are unable to restore project dependencies, ensure that you have the latest .NET MAUI preview installed.

1. Copy the code files (except App.xaml) into the newly created project.
2. In the newly created project, replace the following namespaces:

OLD NAMESPACE

NEW NAMESPACE

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

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

i"

|  |  |
| --- | --- |
| OLD NAMESPACE | NEW NAMESPACE |
| |  | | --- | | using Xamarin.Forms | | |  |  | | --- | --- | | using Microsoft.Maui | AND | | using Microsoft.Maui.Controls | | |
| |  | | --- | | using Xamarin.Forms.Xaml | | |  | | --- | | using Microsoft.Maui.Controls.Xaml | |

1. Run the app on your chosen platform:

dotnet build -t:Run -f net6.0-android dotnet build -t:Run -f net6.0-ios dotnet build -t:Run -f net6.0-maccatalyst

1. Examine any run-time errors, some of which will be due to incomplete handler availability.

|  |
| --- |
| .NET MAUI app startup  6/18/2021 • 2 minutes to read • Edit Online |

.NET Multi-platform App UI (MAUI) apps are bootstrapped using the .NET Generic Host. This enables apps to be initialized from a single location, and provides the ability to configure fonts, services, and third-party libraries.

|  |
| --- |
| Configure |

Each platform has an entry point that initializes the app host builder, and then invokes the method of

|  |  |  |
| --- | --- | --- |
| Startup | class in your app. The | Startup |

the class can be considered the entry point for your app, and is responsible for creating a window that defines the initial page of your app.

|  |  |  |
| --- | --- | --- |
| Startup | class, which must implement the | IStartup |

The interface, must at a minimum provide an app to run:

|  |
| --- |
| using Microsoft.Maui; using Microsoft.Maui.Hosting;  public class Startup : IStartup  {  public void Configure(IAppHostBuilder appBuilder)  {  appBuilder  .UseMauiApp<App>();  }  } |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| App | class should derive from the | Application | class, and must override the | CreateWindow |

|  |
| --- |
| Window |

The method to provide a within which your app runs, and that defines the UI for the initial page of the app:

|  |
| --- |
| using Microsoft.Maui; using Microsoft.Maui.Controls;  public partial class App : Application  {  protected override IWindow CreateWindow(IActivationState activationState)  {  return new Window(new MainPage()); }  } |

|  |  |  |
| --- | --- | --- |
| MainPage | is a | ContentPage |

In the example above, that defines the UI for the initial page of the app.

# Register fonts

Fonts can be added to your app and referenced by filename or alias. This is accomplished by invoking the

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ConfigureFonts | method on the | IAppHostBuilder | object. Then, on the | IFontCollection | object, call the | AddFont |

method to add the required font:

|  |
| --- |
| using Microsoft.Maui; using Microsoft.Maui.Hosting;  public class Startup : IStartup  {  public void Configure(IAppHostBuilder appBuilder)  {  appBuilder  .UseMauiApp<App>()  .ConfigureFonts(fonts =>  {  fonts.AddFont("Lobster-Regular.ttf", "Lobster");  });  }  } |

|  |
| --- |
| AddFont |

In the example above, the first argument to the method is the font filename, while the second argument represents an optional alias by which the font can be referenced when consuming it.

Any custom fonts consumed by an app must be included in your .csproj file. This can be accomplished by referencing their filenames, or by using a wildcard:

<ItemGroup>

<MauiFont Include="Resources\Fonts\\*" /> </ItemGroup>

## NOTE

Fonts added to the project through the Solution Explorer in Visual Studio will automatically be included in the .csproj file.

The font can then be consumed by referencing its 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" />

# Register handlers

|  |  |  |
| --- | --- | --- |
| ConfigureMauiHandlers | method on the | IAppHostBuilder |

|  |  |  |
| --- | --- | --- |
| IMauiHandlersCollection | object, call the | AddHandler |

To register your own handlers, call the object. Then, on the method to add the required handler:

|  |
| --- |
| using Microsoft.Maui; using Microsoft.Maui.Hosting;  public class Startup : IStartup  {  public void Configure(IAppHostBuilder appBuilder)  {  appBuilder  .UseMauiApp<App>()  .ConfigureMauiHandlers(handlers =>  {  handlers.AddHandler(typeof(MyEntry), typeof(MyEntryHandler));  });  }  } |

|  |  |  |
| --- | --- | --- |
| MyEntryHandler | handler is registered against the | MyEntry |

|  |  |  |
| --- | --- | --- |
| MyEntry | control will be handled by the | MyEntryHandler |

In this example, the control. Therefore, any instances of the .

# Register renderers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ConfigureMauiHandlers | method on the | IAppHostBuilder | object. Then, on the | IMauiHandlersCollection |

|  |
| --- |
| AddCompatibilityRenderer |

To use controls backed by .NET MAUI handlers, with specific controls backed by Xamarin.Forms renderers, call the object, call the method to add the required renderer:

|  |
| --- |
| using Microsoft.Maui; using Microsoft.Maui.Hosting;  using Microsoft.Maui.Controls.Compatibility;  public class Startup : IStartup  {  public void Configure(IAppHostBuilder appBuilder)  {  appBuilder  .UseMauiApp<App>()  #if \_\_ANDROID\_\_  .ConfigureMauiHandlers(handlers =>  {  handlers.AddCompatibilityRenderer(typeof(Microsoft.Maui.Controls.BoxView), typeof(Microsoft.Maui.Controls.Compatibility.Platform.Android.BoxRenderer)); handlers.AddCompatibilityRenderer(typeof(Microsoft.Maui.Controls.Frame),    typeof(Microsoft.Maui.Controls.Compatibility.Platform.Android.FastRenderers.FrameRenderer));  });  #elif \_\_IOS\_\_  .ConfigureMauiHandlers(handlers =>  {  handlers.AddCompatibilityRenderer(typeof(Microsoft.Maui.Controls.BoxView), typeof(Microsoft.Maui.Controls.Compatibility.Platform.iOS.BoxRenderer)); handlers.AddCompatibilityRenderer(typeof(Microsoft.Maui.Controls.Frame), typeof(Microsoft.Maui.Controls.Compatibility.Platform.iOS.FrameRenderer));  });  #endif  }  } |

|  |  |  |
| --- | --- | --- |
| BoxView | and | Frame |

In this example, all controls in the app will be backed by handlers, aside from the controls that will be backed by Xamarin.Forms renderers.

|  |
| --- |
| .NET MAUI single project  6/16/2021 • 2 minutes to read • Edit Online |

.NET Multi-platform App UI (MAUI) single project is a collection of features that brings all the platform-specific experiences you encounter while developing apps into one shared head project that can target Android, iOS, macOS, and Windows.

Single project is enabled using multi-targeting and the use of SDK-style projects in .NET 6. You can still expect access to all the platform-specific experiences and tools when you need them, while enjoying a simplified, shared development experience across all the platforms you're targeting.

# Simplify development

Single project is built on top of a collection of experiences that are being simplified in .NET 6. The following list shows the experiences that will be shared in .NET MAUI single project:

Resources

Images

Fonts

App icons

Splash screens

Raw Assets

App manifest

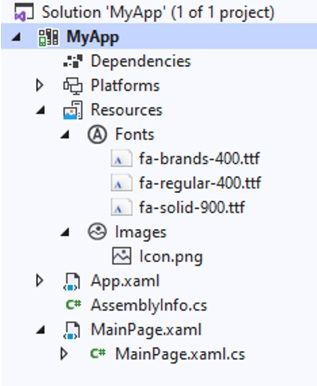
NuGet

Platform-specific code

All other features are being moved from their own platform-projects into platform folders in the single project.

# Visual Studio changes

In addition to the simplified, shared experiences in single project, there are changes being made to Visual Studio to support single project. These changes will enable the use of a shared resource file within the single project, platform files for platform-specific development, and a simplified debug target selection for running your .NET MAUI apps:



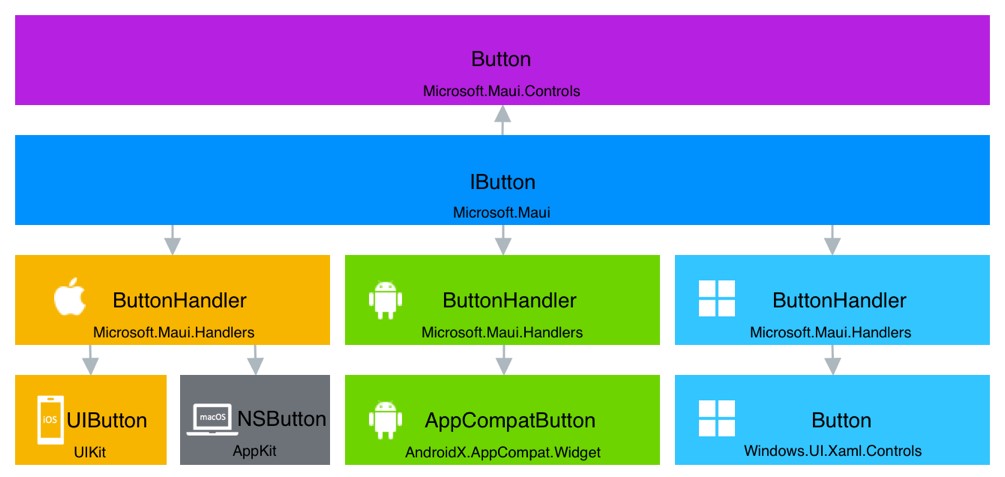
|  |
| --- |
| Customize .NET MAUI controls with handlers  7/8/2021 • 2 minutes to read • Edit Online |

.NET Multi-platform App UI (MAUI) provides a collection of controls that can be used to display data, initiate actions, indicate activity, display collections, pick data, and more. By default, handlers map these cross-platform

controls to native controls on each platform. For example, on iOS a .NET MAUI handler will map a .NET MAUI

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Button | to an iOS | UIButton | . On Android, the | Button | will be mapped to a | AppCompatButton |

:



|  |
| --- |
| IButton |

Handlers can be accessed through a control-specific interface provided by .NET MAUI, such as for a

|  |
| --- |
| Button |

. This avoids the cross-platform control having to reference its handler, and the handler having to reference the cross-platform control. The mapping of the cross-platform control API to the platform API is provided by a mapper.

Handlers expose the native control that implements the .NET MAUI cross-platform control via the NativeView property. This property can be accessed to set platform properties, invoke platform methods, and subscribe to platform events.

|  |
| --- |
| App |

Handlers are global and therefore customization can occur anywhere in your .NET MAUI app. For example, customizing a handler for a control in your class will result in all controls of that type being customized in the app. Similarly, customizing a handler for a control in the first page of your app will also result in all controls of that type in the app being customized.

# Examples

|  |
| --- |
| #ifdef |

Handlers are typically customized to augment the appearance and behavior of a native platform control. Handlers can be customized per platform by using the compiler directive, to multi-target code based on the platform. However, you can just as easily use platform-specific folders or files to organize such code.

**NOTE**

Handler customization can range from the simple to the complex. More complex examples will appear in time.

## Customize the background color of all controls

The following example customizes the background color of every control in the app, on Android, to cyan:

|  |
| --- |
| using Microsoft.Maui; using Microsoft.Maui.Controls;  public partial class App : Application  {  public App()  {  InitializeComponent();  #if \_\_ANDROID\_\_  Microsoft.Maui.Handlers.ViewHandler.ViewMapper[nameof(IView.BackgroundColor)] = (h, v) =>  {  (h.NativeView as  Android.Views.View).SetBackgroundColor(Microsoft.Maui.Graphics.Colors.Cyan.ToNative());  };  #endif  }  } |

## Remove Android Entry underline

|  |
| --- |
| Entry |

The following example removes the underline from all controls in the app, on Android:

|  |
| --- |
| using Microsoft.Maui; using Microsoft.Maui.Controls;  public partial class MainPage : ContentPage, IPage  {  public MainPage()  {  InitializeComponent();  #if \_\_ANDROID\_\_  Handlers.EntryHandler.EntryMapper[nameof(IEntry.BackgroundColor)] = (h, v) => {  (h.NativeView as global::Android.Views.Entry).UnderlineVisible = false;  };  #endif  }  } |

## Customize specific control instances

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entry | controls on a page that contains multiple | Entry | controls, you should first subclass the | Entry |

Handlers for specific control instances can be customized by subclassing the control, and then by modifying the handler for the parent control only when the control is of the subclassed type. For example, to customize specific control:

using Microsoft.Maui.Controls;

namespace MyMauiApp

{

public class MyEntry : Entry

{

}

}

|  |  |  |
| --- | --- | --- |
| EntryHandler | to perform the desired modification to | MyEntry |

You can then customize the instances:

|  |
| --- |
| using Microsoft.Maui; using Microsoft.Maui.Controls; using Microsoft.Maui.Graphics;  namespace MauiApp1  {  public partial class App : Application  {  public App()  {  InitializeComponent();  Microsoft.Maui.Handlers.EntryHandler.EntryMapper[nameof(IView.BackgroundColor)] = (handler, view)  =>  {  if (view is MyEntry)  { #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  }  };  }  }  } |

|  |
| --- |
| MyEntry |

Any instances will then be customized as per the handler modification.