Get started with Xamarin

Xamarin.Forms is a cross-platform UI toolkit that allows developers to efficiently create native user interface layouts that can be shared across iOS, Android, and Universal Windows Platform apps.

Get started

OVERVIEW

What is Xamarin?

Supported platforms

HOW-TO GUIDE

Installation

Build your first app

GET STARTED

Xamarin.Forms on Q&A

Quickstarts

HOW-TO GUIDE

Create a Xamarin.Forms application

Perform navigation in a Xamarin.Forms application

Store data in a local SQLite.NET database

Style a cross-platform Xamarin.Forms application

CONCEPT

Quickstart deep dive

User interface tutorials

HOW-TO GUIDE

StackLayout

Label

Button

Entry

Editor

Image

Grid

CollectionView

Pop-ups

App fundamentals tutorials

HOW-TO GUIDE

App lifecycle

Local database

Web services

Learn about Xamarin

CONCEPT

.NET developers

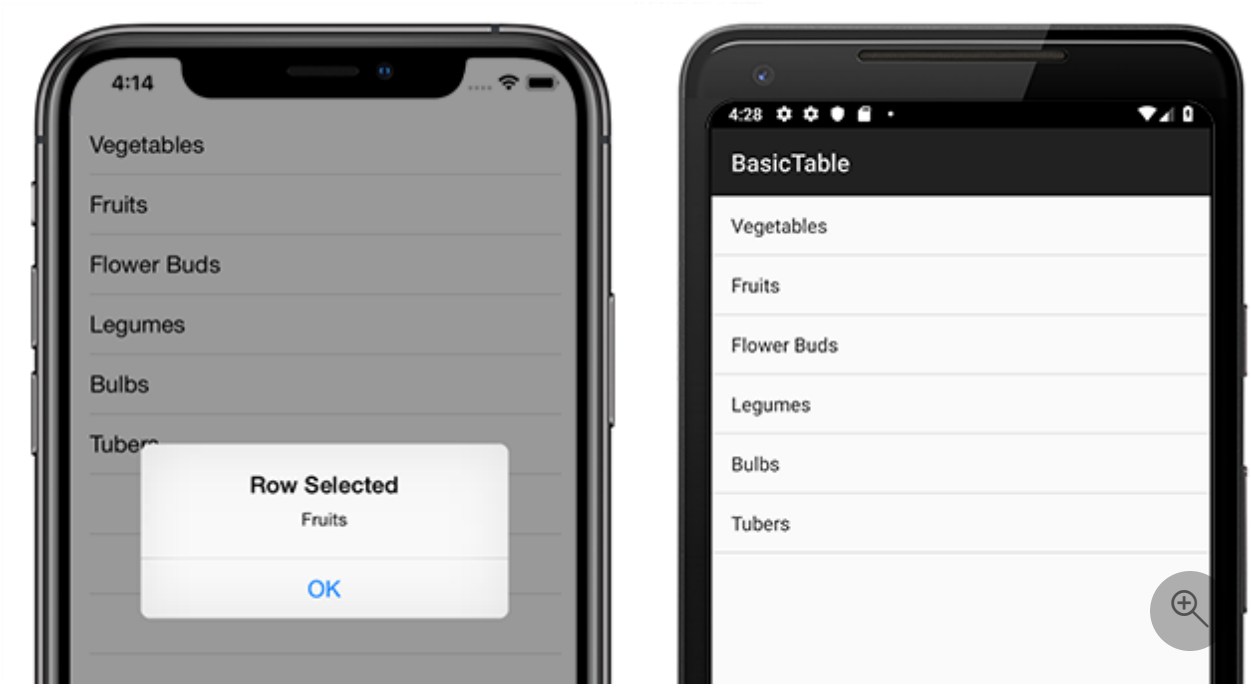
Java developers

Objective-C developers

Azure

What is Xamarin?

Article • 09/20/2022 • 4 minutes to read



Xamarin is an open-source platform for building modern and performant applications for iOS, Android, and Windows with .NET. Xamarin is an abstraction layer that manages communication of shared code with underlying platform code. Xamarin runs in a managed environment that provides conveniences such as memory allocation and garbage collection.

Xamarin enables developers to share an average of 90% of their application across platforms. This pattern allows developers to write all of their business logic in a single language (or reuse existing application code) but achieve native performance, look, and feel on each platform.

Xamarin applications can be written on PC or Mac and compile into native application packages, such as an **.apk** file on Android, or an **.ipa** file on iOS.

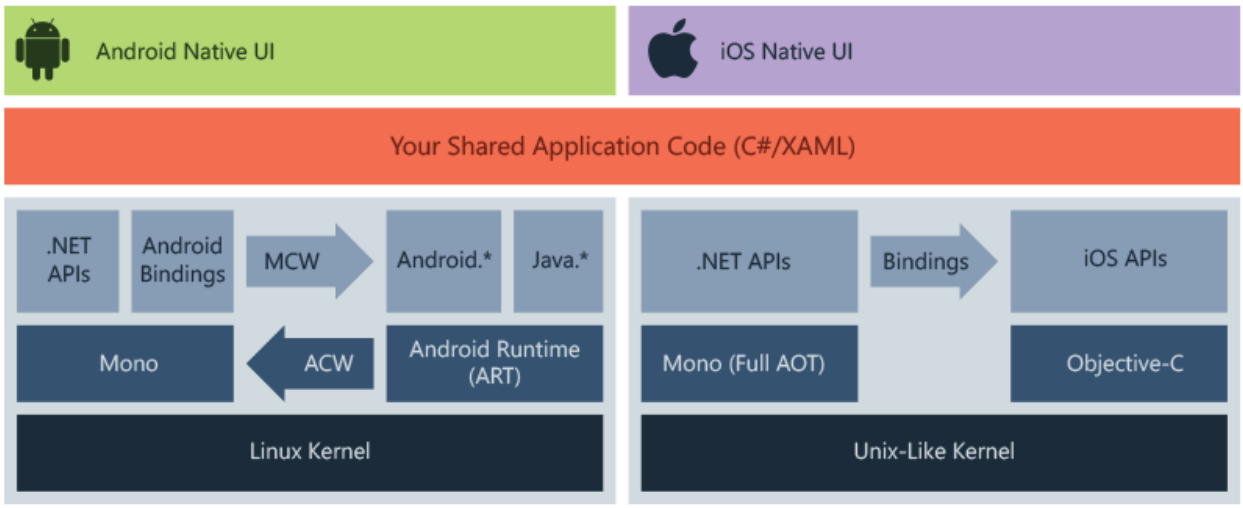
> ⓘ **Note**
>
> Compiling and deploying applications for iOS currently requires a MacOS machine. For more information about development requirements, see **system requirements**.

Who Xamarin is for

Xamarin is for developers with the following goals:

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

How Xamarin works



The diagram shows the overall architecture of a cross-platform Xamarin application. Xamarin allows you to create native UI on each platform and write business logic in C# that is shared across platforms. In most cases, 80% of application code is sharable using Xamarin.

Xamarin is built on top of .NET, which automatically handles tasks such as memory allocation, garbage collection and interoperability with underlying platforms.

For more information about platform-specific architecture, see Xamarin.Android and Xamarin.iOS.

Added features

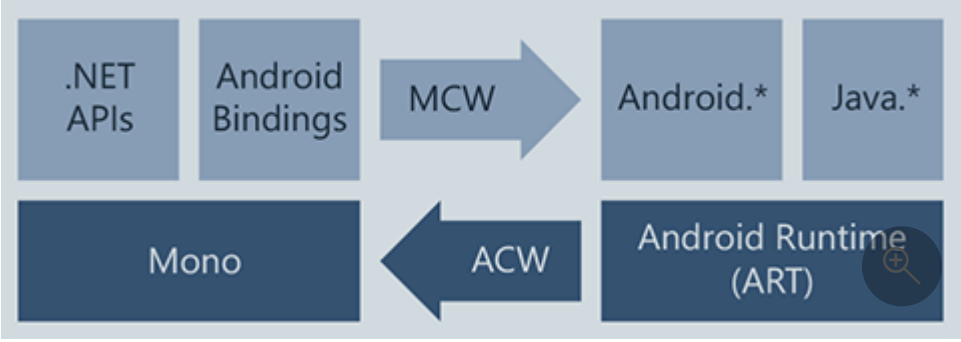
Xamarin combines the abilities of native platforms, while adding features that include:

1. **Complete binding for the underlying SDKs** – Xamarin contains bindings for nearly the entire underlying platform SDKs in both iOS and Android. Additionally, these bindings are strongly-typed, which means that they're easy to navigate and use, and provide robust compile-time type checking and during development. Strongly-typed bindings lead to fewer runtime errors and higher-quality applications.
2. **Objective-C, Java, C, and C++ Interop** – Xamarin provides facilities for directly invoking Objective-C, Java, C, and C++ libraries, giving you the power to use a wide array of third party code. This functionality lets you use existing iOS and Android libraries written in Objective-C, Java, or C/C++. Additionally, Xamarin

offers binding projects that allow you to bind native Objective-C and Java libraries using a declarative syntax.

3. **Modern language constructs** – Xamarin applications are written in C#, a modern language that includes significant improvements over Objective-C and Java such as dynamic language features, functional constructs such as lambdas, LINQ, parallel programming, generics, and more.
4. **Robust Base Class Library (BCL)** – Xamarin applications use the .NET BCL, a large collection of classes that have comprehensive and streamlined features such as powerful XML, Database, Serialization, IO, String, and Networking support, and more. Existing C# code can be compiled for use in an app, which provides access to thousands of libraries that add functionality beyond the BCL.
5. **Modern Integrated Development Environment (IDE)** – Xamarin uses Visual Studio, a modern IDE that includes features such as code auto completion, a sophisticated project and solution management system, a comprehensive project template library, integrated source control, and more.
6. **Mobile cross-platform support** – Xamarin offers sophisticated cross-platform support for the three major platforms of iOS, Android, and Windows. Applications can be written to share up to 90% of their code, and Xamarin.Essentials offers a unified API to access common resources across all three platforms. Shared code can significantly reduce both development costs and time to market for mobile developers.

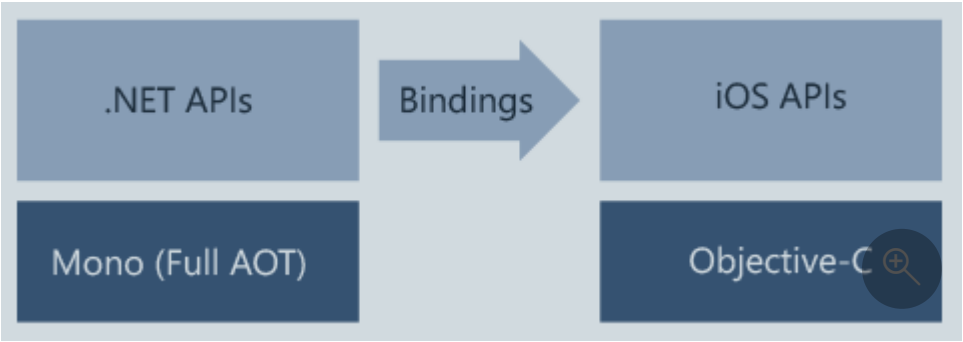
Xamarin.Android



Xamarin.Android applications compile from C# into **Intermediate Language (IL)** which is then **Just-in-Time (JIT)** compiled to a native assembly when the application launches. Xamarin.Android applications run within the Mono execution environment, side by side with the Android Runtime (ART) virtual machine. Xamarin provides .NET bindings to the Android.* and Java.* namespaces. The Mono execution environment calls into these namespaces via **Managed Callable Wrappers (MCW)** and provides **Android Callable Wrappers (ACW)** to the ART, allowing both environments to invoke code in each other.

For more information, see Xamarin.Android architecture.

Xamarin.iOS



Xamarin.iOS applications are fully **Ahead-of-Time (AOT)** compiled from C# into native ARM assembly code. Xamarin uses **Selectors** to expose Objective-C to managed C# and Registrars to expose managed C# code to Objective-C. Selectors and Registrars collectively are called "bindings" and allow Objective-C and C# to communicate.

For more information, see Xamarin.iOS architecture.

Xamarin.Essentials

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

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

For more information, see Xamarin.Essentials.

Xamarin.Forms

Xamarin.Forms is an open-source UI framework. Xamarin.Forms allows developers to build Xamarin.iOS, Xamarin.Android, and Windows applications from a single shared codebase. Xamarin.Forms allows developers to create user interfaces in XAML with code-behind in C#. These user interfaces are rendered as performant native controls on each platform. Some examples of features provided by Xamarin.Forms include:

- XAML user-interface language

- Databinding
- Gestures
- Effects
- Styling

For more information, see Xamarin.Forms.

Get started

The following guides will help you build your first app using Xamarin:

- Get started with Xamarin.Forms
- Get started with Xamarin.Android
- Get started with Xamarin.iOS
- Get started with Xamarin.Mac

Related video

https://learn.microsoft.com/shows/Xamarin-101/What-is-Xamarin-1-of-11/player

Find more Xamarin videos on Channel 9 and YouTube.

Installing Xamarin

Article • 09/22/2022 • 2 minutes to read

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

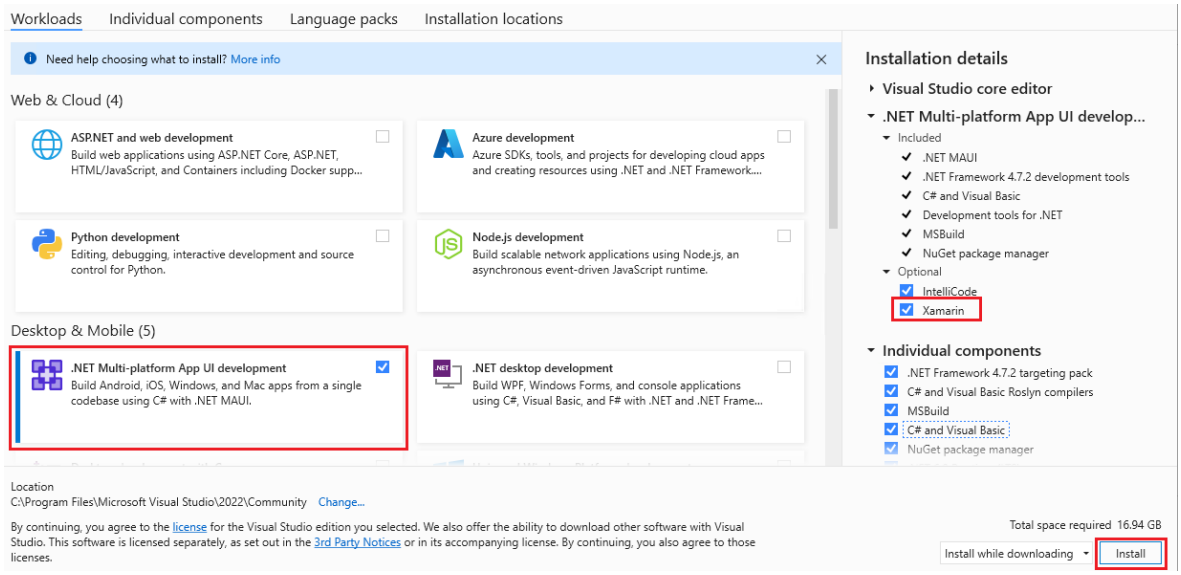
Installing Xamarin on Windows

Xamarin can be installed as part of a *new* Visual Studio 2022 installation, with the following steps:

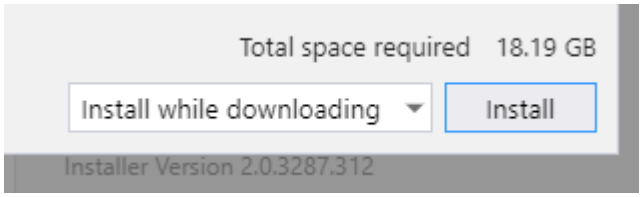
1. Download Visual Studio 2022 Community, Visual Studio Professional, or Visual Studio Enterprise from the Visual Studio page.

2. Double-click the downloaded package to start installation.

3. Select the **.NET Multi-platform App UI development** workload from the installation screen, and under **Optional** check **Xamarin**:



4. When you are ready to begin Visual Studio 2022 installation, click the **Install** button in the lower right-hand corner:



5. When Visual Studio 2022 installation has completed, click the **Launch** button to start Visual Studio.

Adding Xamarin to Visual Studio 2022

If Visual Studio 2022 is already installed, add Xamarin by re-running the Visual Studio 2022 installer to modify workloads (see Modify Visual Studio for details). Next, follow the steps listed above to install .NET Multi-platform App UI development and the optional Xamarin install..

For more information about downloading and installing Visual Studio 202022, see Install Visual Studio 2022.

Related Links

- Uninstalling Xamarin
- Xamarin Firewall Configuration Instructions

Installing Xamarin in Visual Studio 2019

Article • 07/08/2021 • 2 minutes to read

Check the system requirements before you begin.

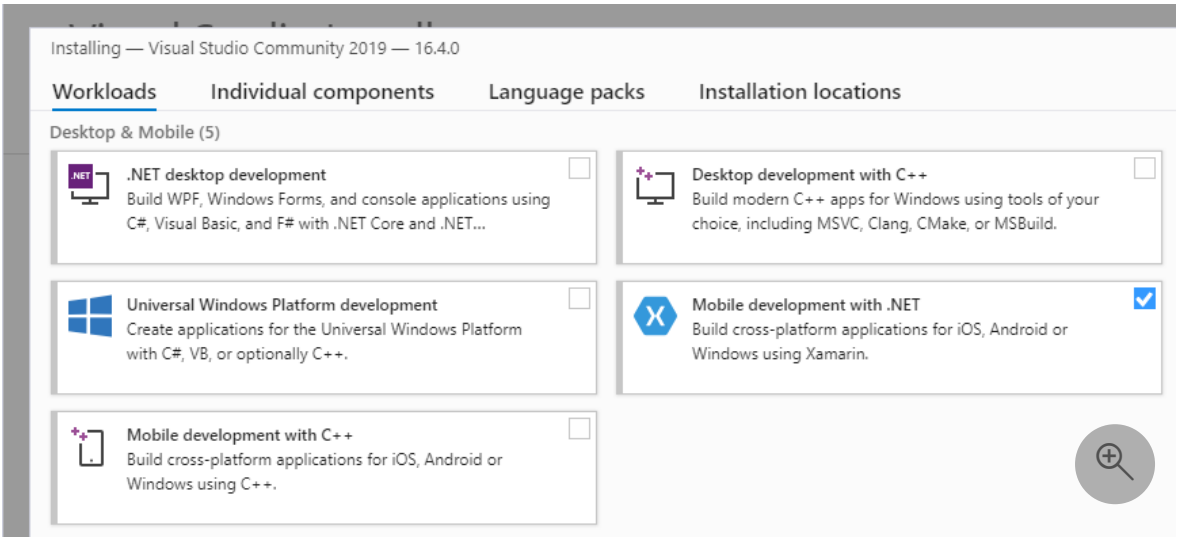
Installation

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

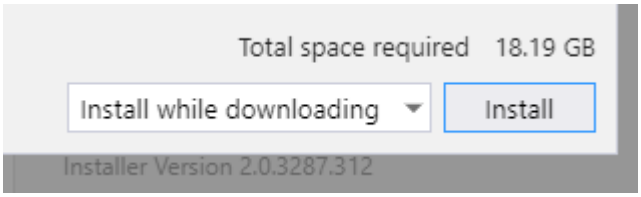
1. Download Visual Studio 2019 Community, Visual Studio Professional, or Visual Studio Enterprise from the Visual Studio page.

2. Double-click the downloaded package to start installation.

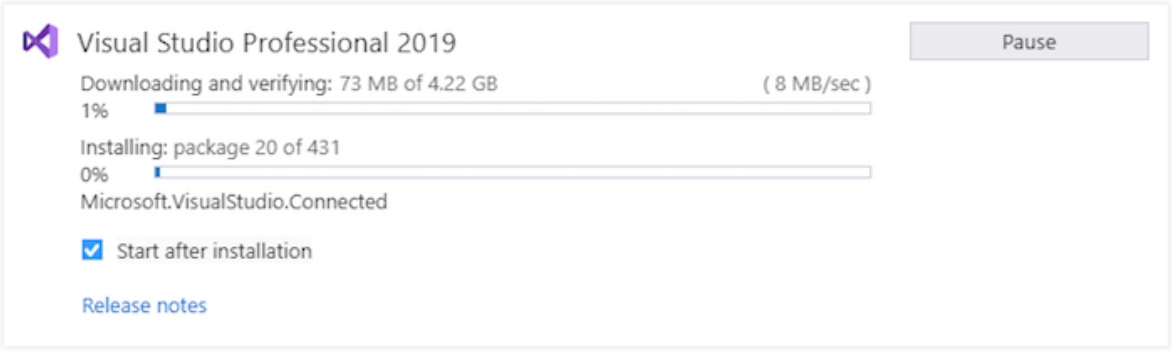
3. Select the **Mobile development with .NET** workload from the installation screen:



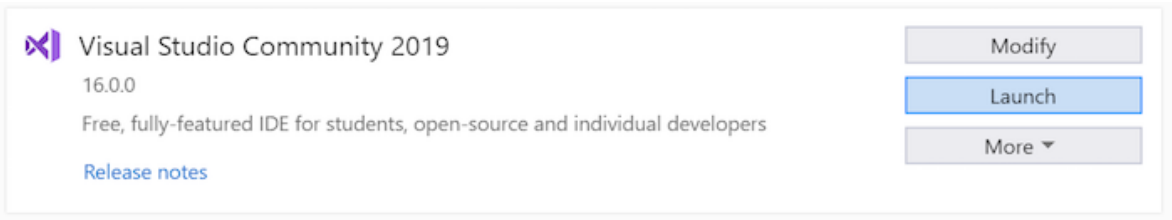
4. When you are ready to begin Visual Studio 2019 installation, click the **Install** button in the lower right-hand corner:



   Use the progress bars to monitor the installation:



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

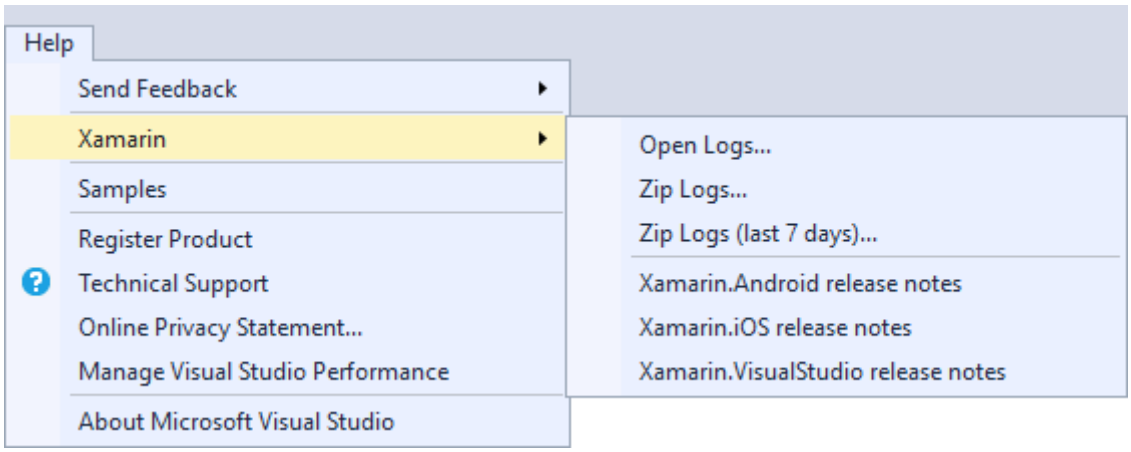


Adding Xamarin to Visual Studio 2019

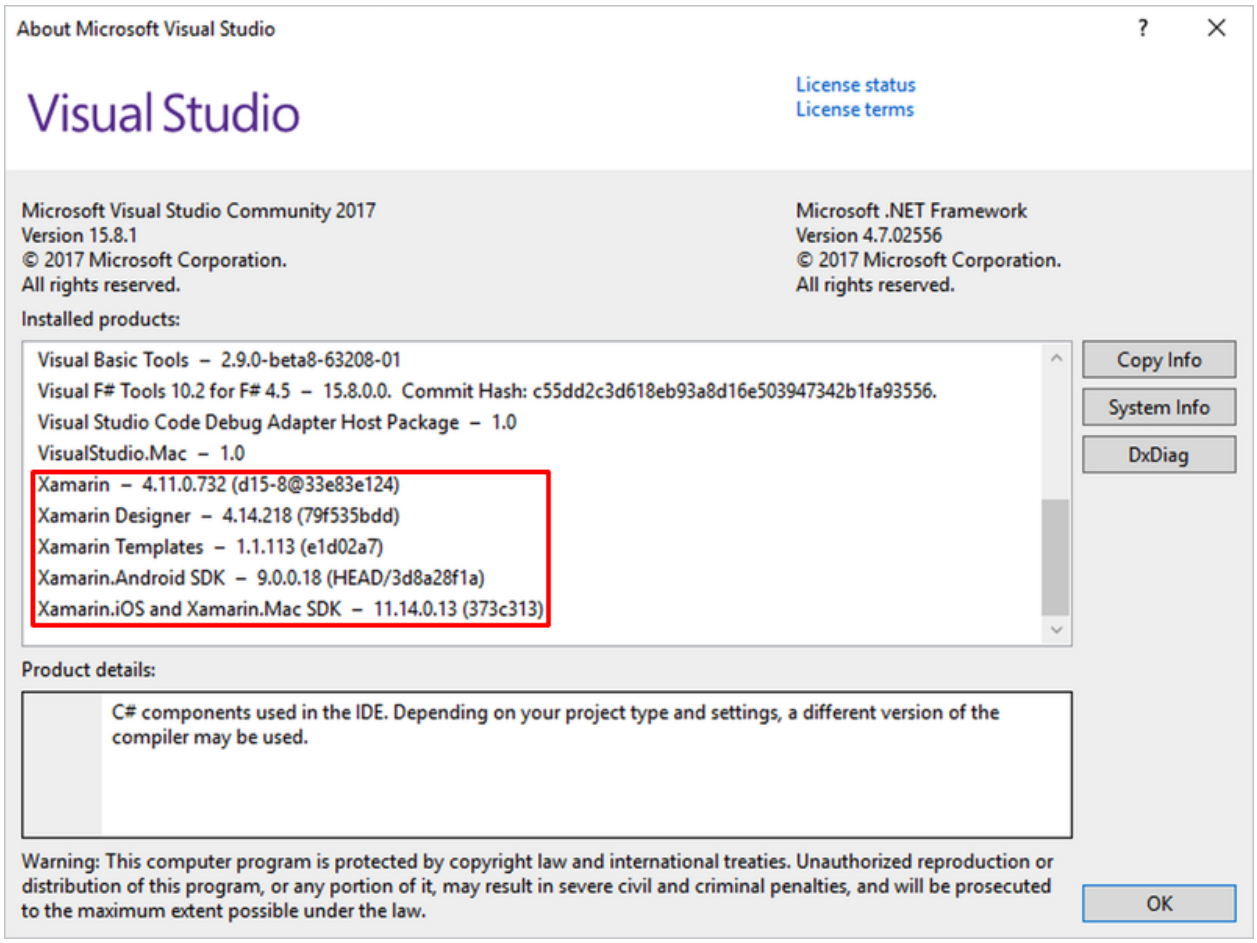
If Visual Studio 2019 is already installed, add Xamarin by re-running the Visual Studio 2019 installer to modify workloads (see Modify Visual Studio for details). Next, follow the steps listed above to install Xamarin.

For more information about downloading and installing Visual Studio 2019, see Install Visual Studio 2019.

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



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



For more information about locating version information, see Where can I find my version information and logs?

Next steps

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

iOS

For more detailed information, see the Installing Xamarin.iOS on Windows guide.

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

Android

For more detailed information, see the Installing Xamarin.Android on Windows guide.

1. Xamarin.Android Configuration
2. Using the Xamarin Android SDK Manager
3. Android SDK Emulator
4. Set Up Device for Development

Installing Xamarin Preview on Windows

Article • 07/08/2021 • 2 minutes to read

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

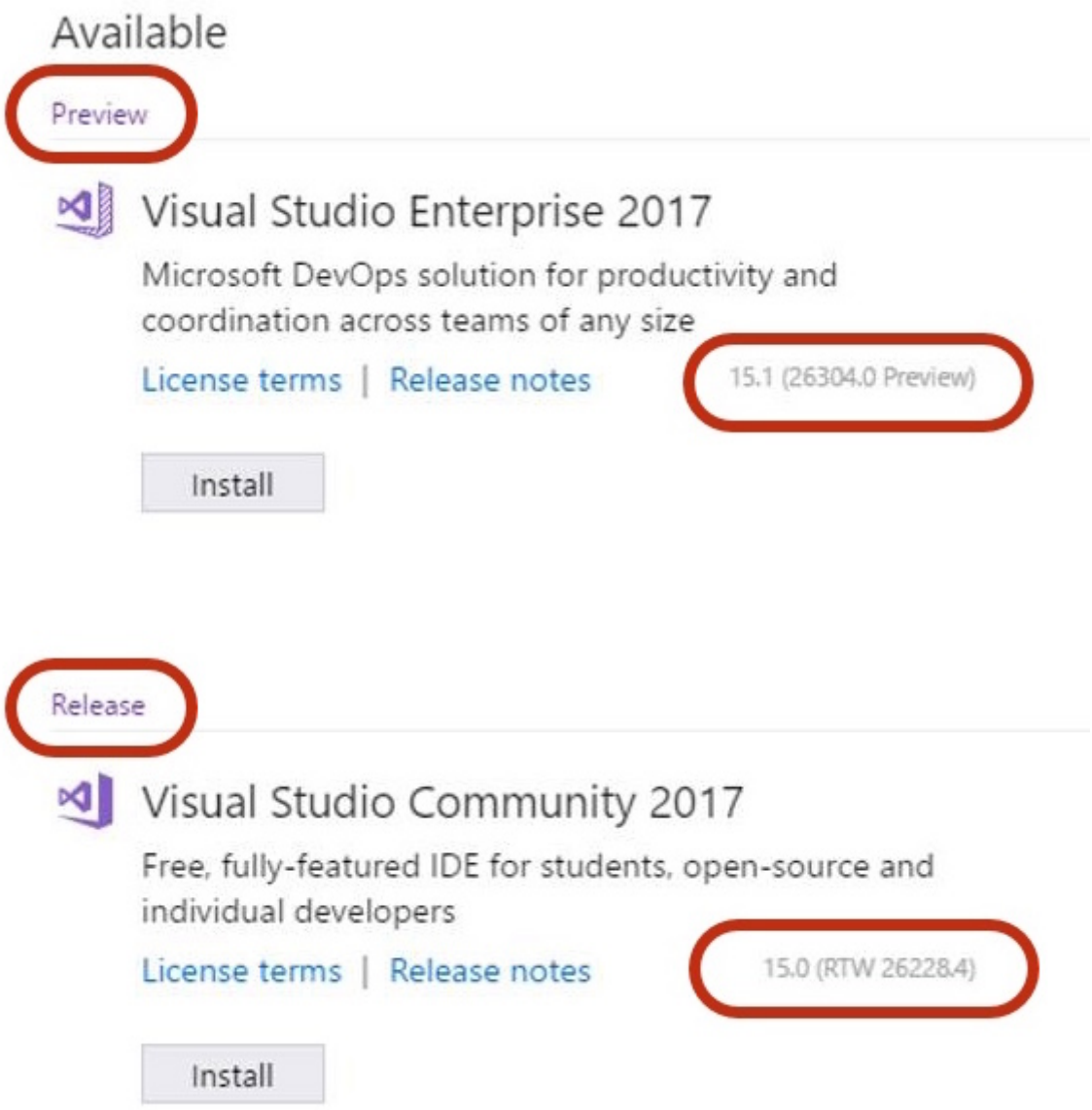
- **Release** – equivalent to the *Stable* channel in Visual Studio for Mac
- **Preview** – equivalent to the *Alpha* and *Beta* channels in Visual Studio for Mac

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

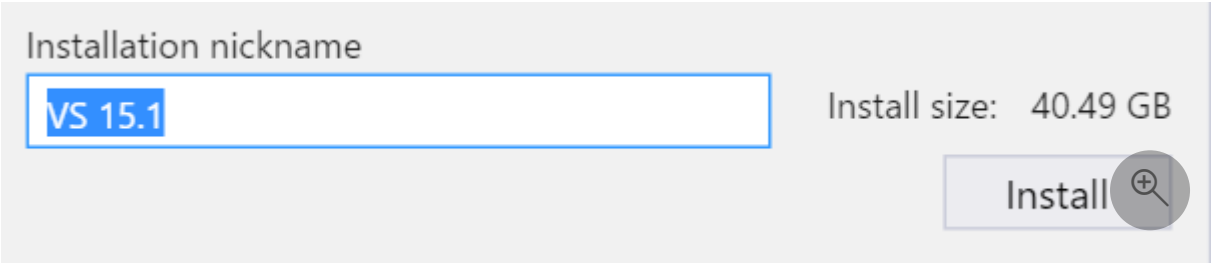
The Preview version of Visual Studio may include corresponding Preview versions of Xamarin functionality, including:

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

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



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



Uninstalling Visual Studio 2019 Preview

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

Uninstall Xamarin from Visual Studio

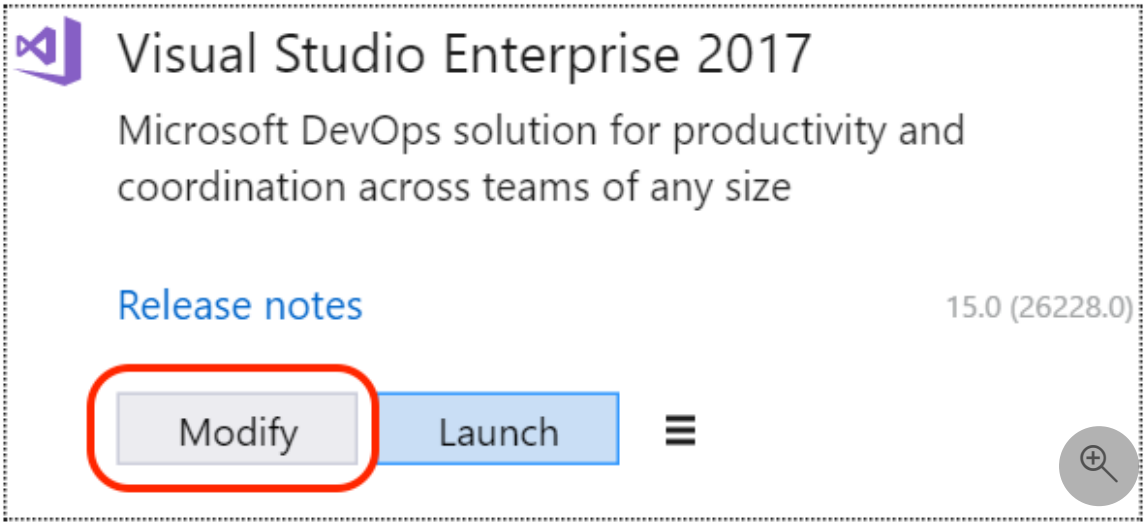
Article • 07/08/2021 • 2 minutes to read

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

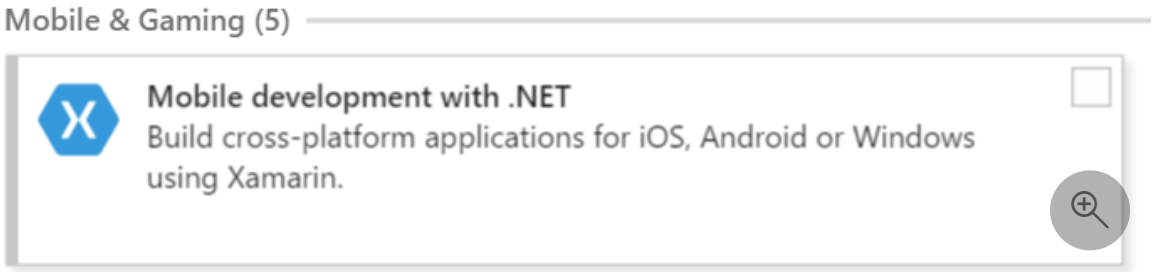
Visual Studio 2019 and Visual Studio 2017

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

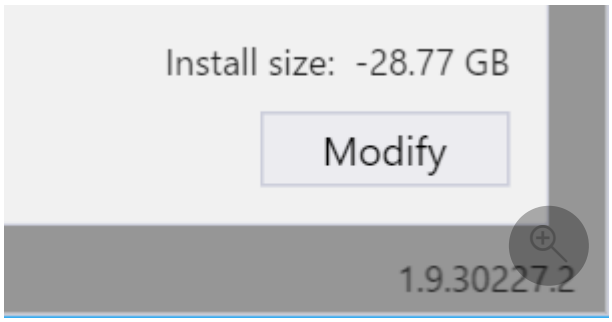
1. Use the **Start menu** to open the **Visual Studio Installer**.
2. Press the **Modify** button for the instance you wish to change.



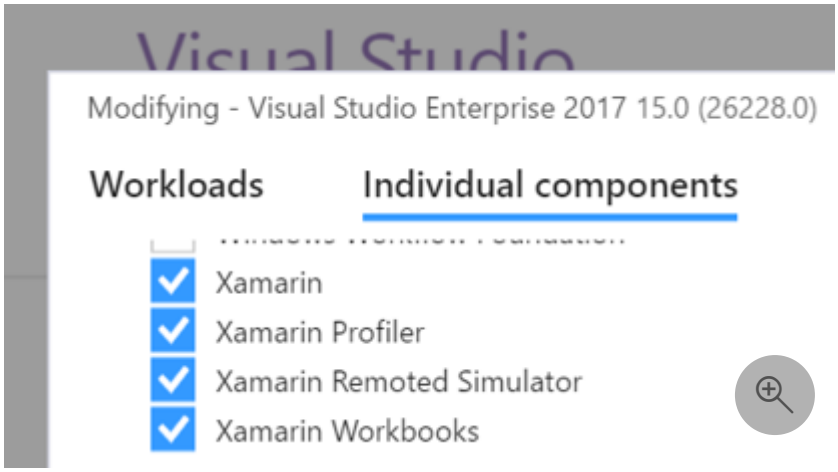
3. In the **Workloads** tab, de-select the **Mobile Development with .NET** option (in the **Mobile & Gaming** section).



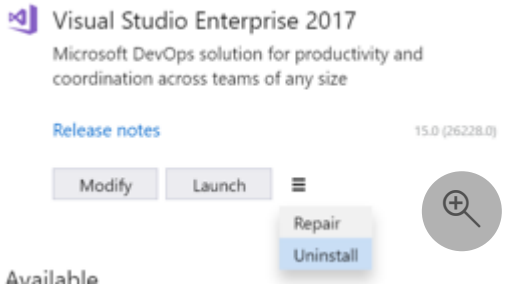
4. Click the **Modify** button in the bottom right of the window.
5. The installer will remove the de-selected components (Visual Studio 2017 must be closed before the installer can make any changes).



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



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



> **ⓘ Important**
>
> If you have two (or more) instances of Visual Studio installed side-by-side (SxS) – such as a Release and a Preview version – uninstalling one instance might remove some Xamarin functionality from the other Visual Studio instance(s), including:
>
> - Xamarin Profiler
> - Xamarin Workbooks/Inspector
> - Xamarin Remote iOS Simulator
> - Apple Bonjour SDK

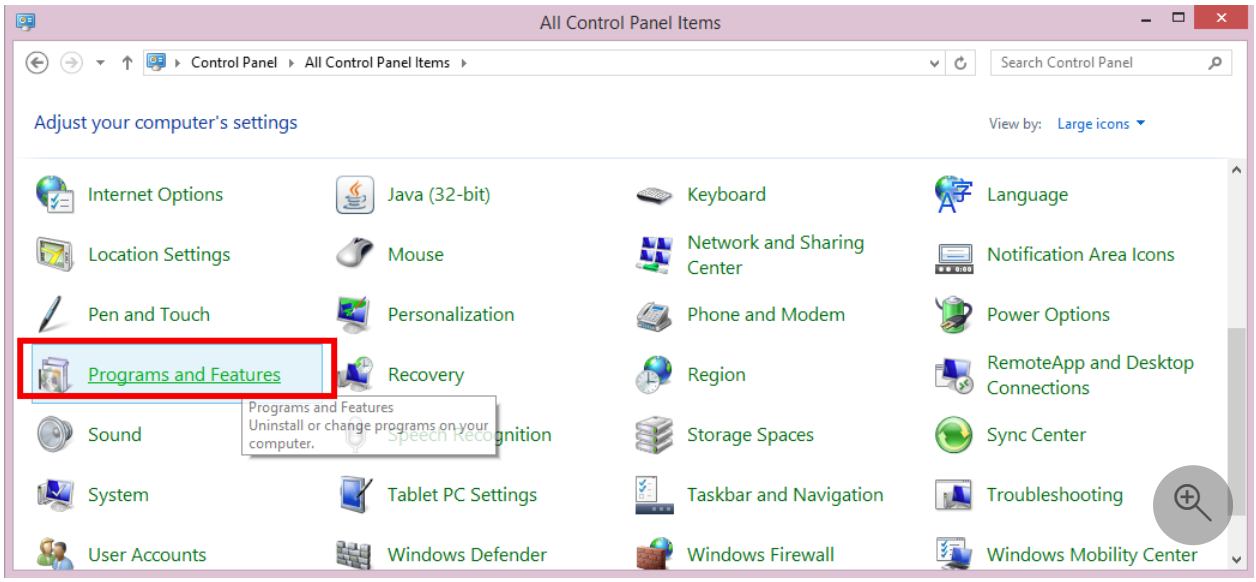
Under certain conditions, uninstalling one of the SxS instances can result in the incorrect removal of these features. This may degrade the performance of the Xamarin Platform on the Visual Studio instance(s) that remain on the system after the uninstallation of the SxS instance.

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

Visual Studio 2015 and earlier

To uninstall Visual Studio 2015 completely, use the support answer on visualstudio.com.

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



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

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

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

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

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

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

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

```
%LOCALAPPDATA%\VirtualStore
```

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

```
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Sha
redDlls
```

Find and delete any entries that match this pattern:

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

Look for this key:

```
HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\1*.0\ExtensionManager\Pend
ingDeletions
```

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

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

example, for Visual Studio 2015:

cmd

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

Install Visual Studio for Mac

Article • 12/17/2022 • 3 minutes to read

Applies to: Visual Studio for Mac Visual Studio

To start developing native, cross-platform .NET apps on macOS, install Visual Studio for Mac following the steps below.

Download Visual Studio for Mac

Learn more about the changes in the release notes.

Prerequisites

- See Visual Studio 2022 for Mac System Requirements for supported operating systems, hardware, supported languages, and additional requirements and guidance.

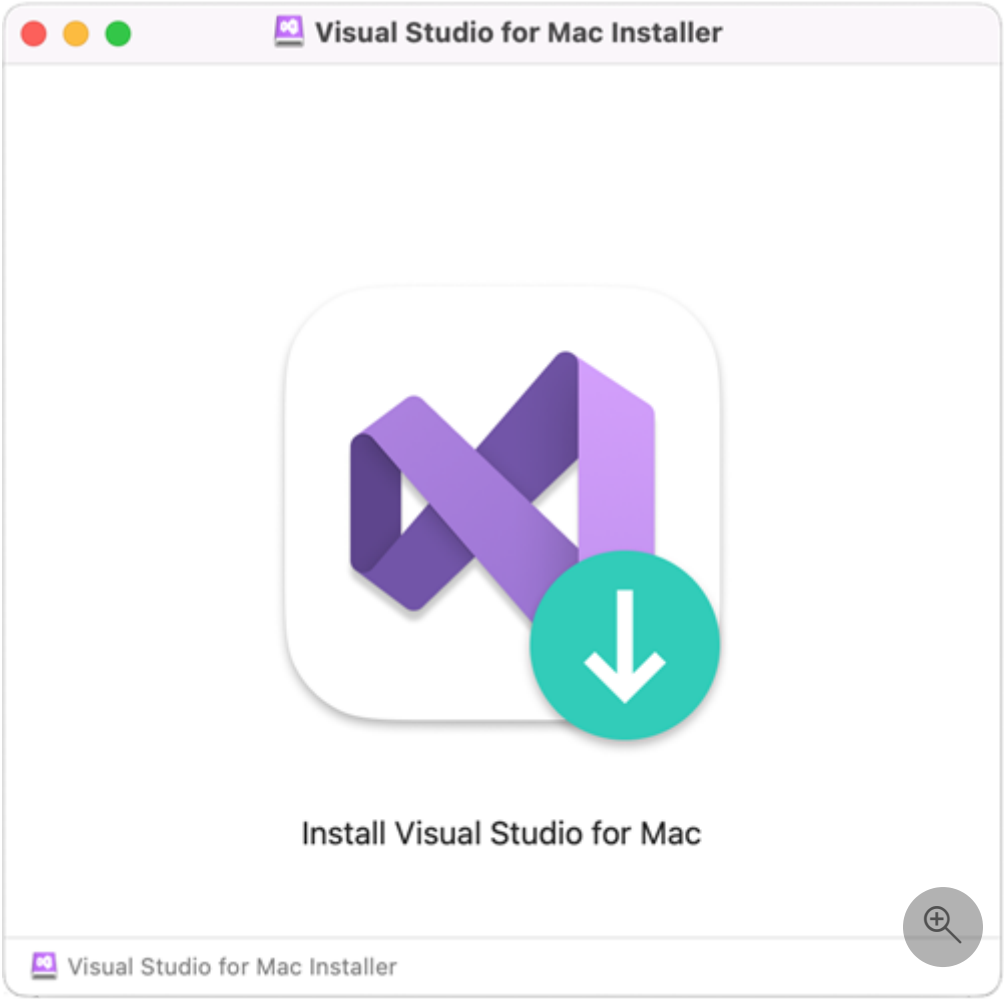
To build Xamarin apps for iOS or macOS, you'll also need:

- A Mac that is compatible with the latest version of Xcode. See Apple's minimum requirements documentation
- The latest version of Xcode. It may be possible to use an older version of Xcode if your Mac isn't compatible with the latest version.
- An Apple ID. If you don't have an Apple ID already, you can create a new one at https://appleid.apple.com. It's necessary to have an Apple ID for installing and signing into Xcode.

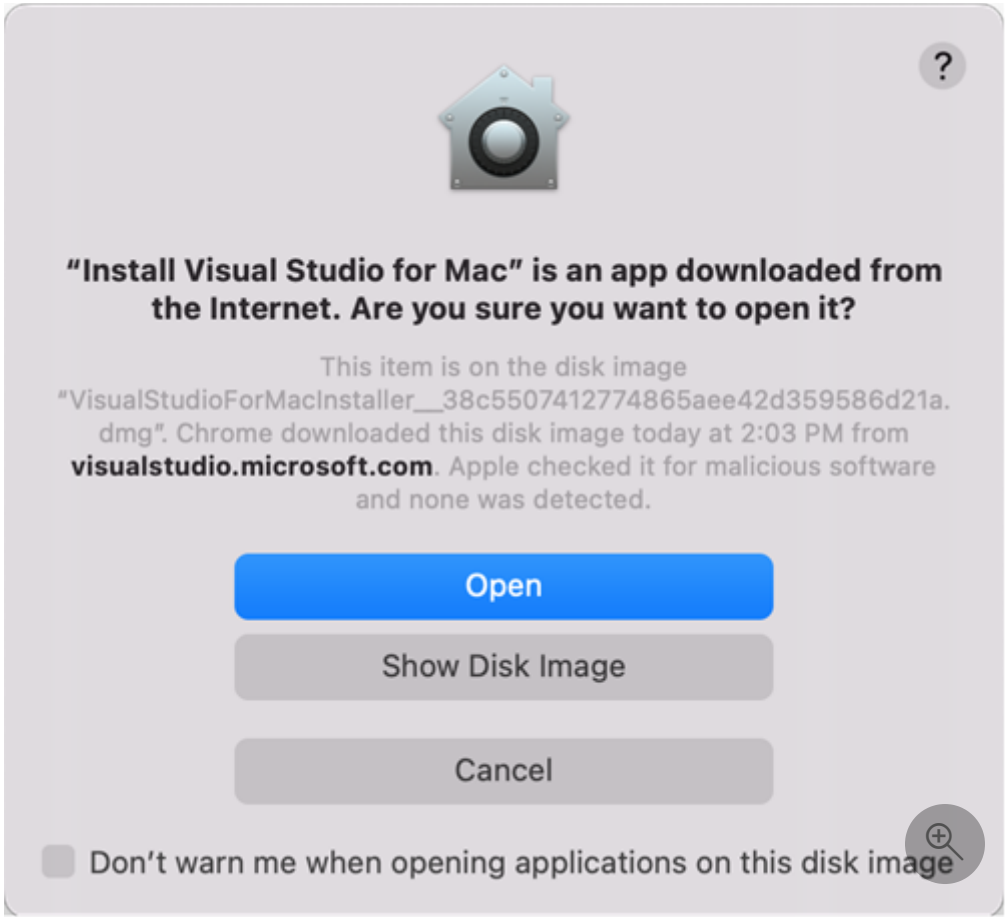
Installation instructions

1. Download the installer from the Visual Studio for Mac download page.

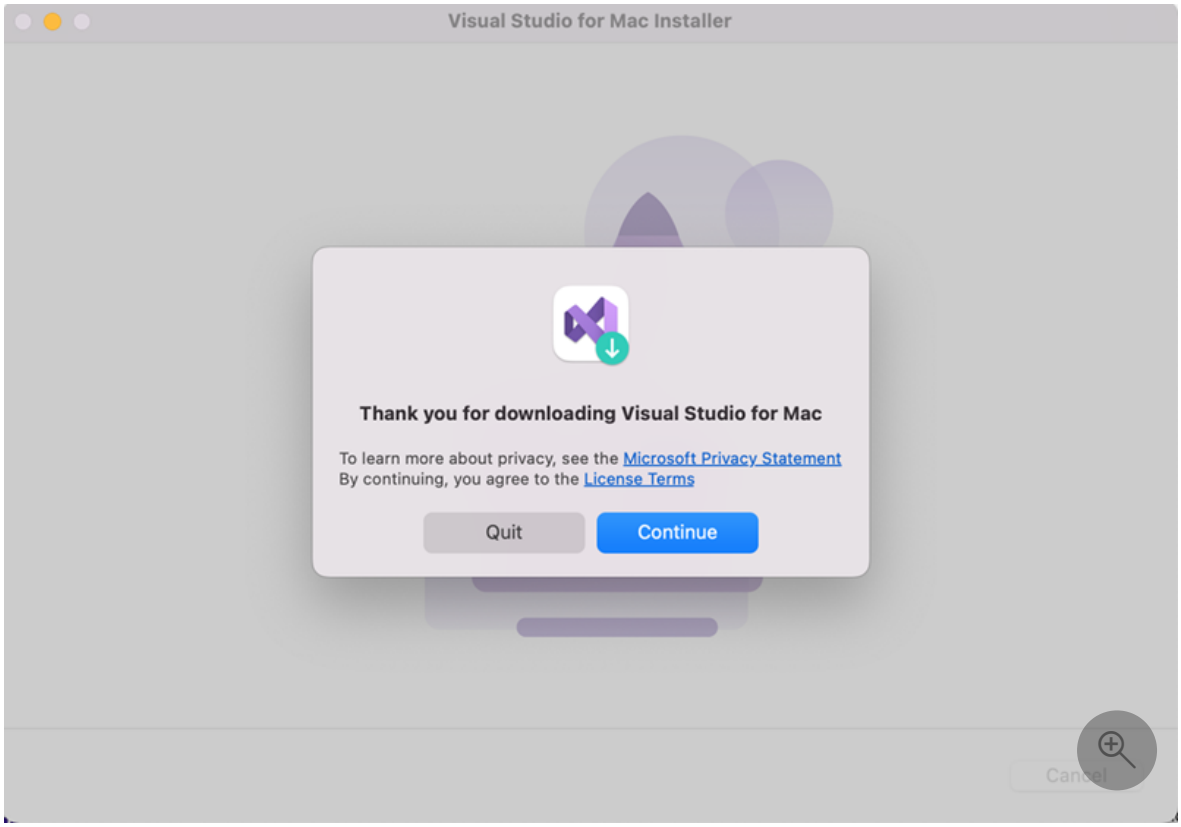
2. Once the download is complete, click the **VisualStudioForMacInstaller_<build_number>.dmg** to mount the installer, then run it by double-clicking the arrow logo:



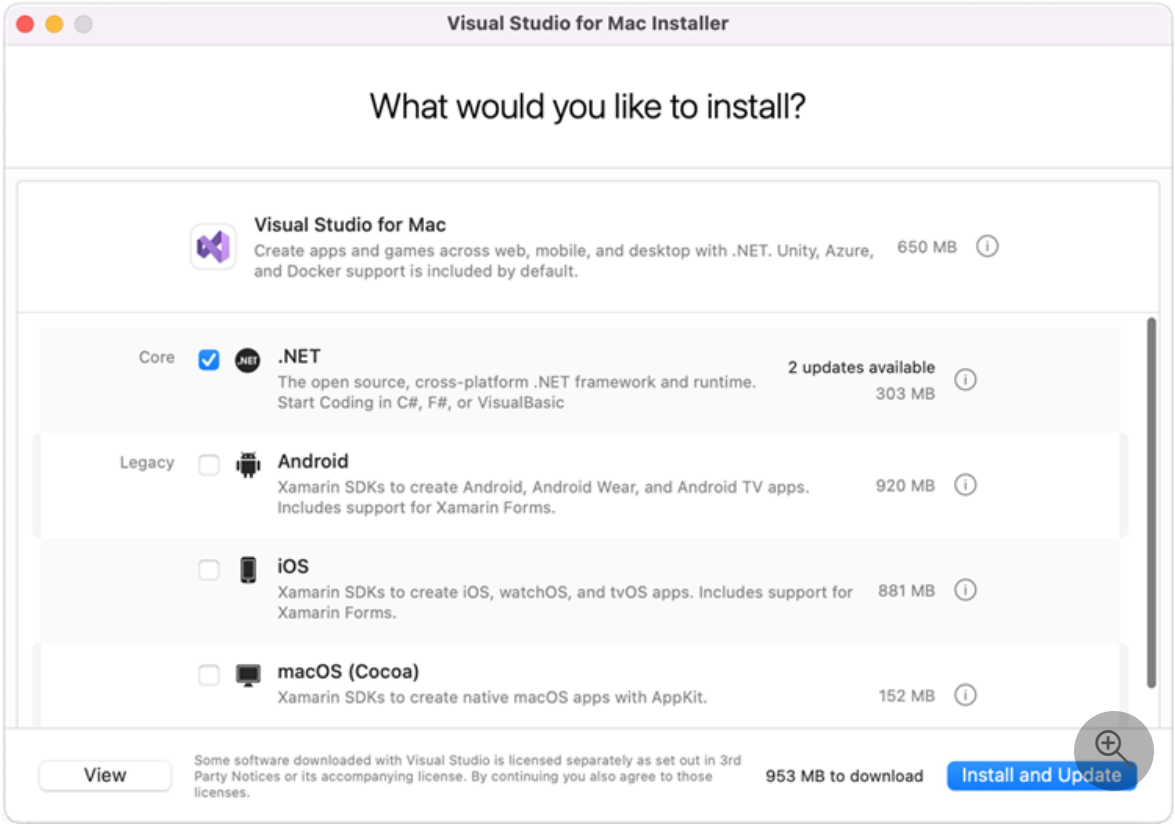
3. You may be presented with a warning about the application being downloaded from the Internet. Select **Open**.



4. An alert will appear asking you to acknowledge the privacy and license terms. Follow the links to read them, then select **Continue** if you agree:



5. The list of available workloads is displayed. Select the components you wish to use:



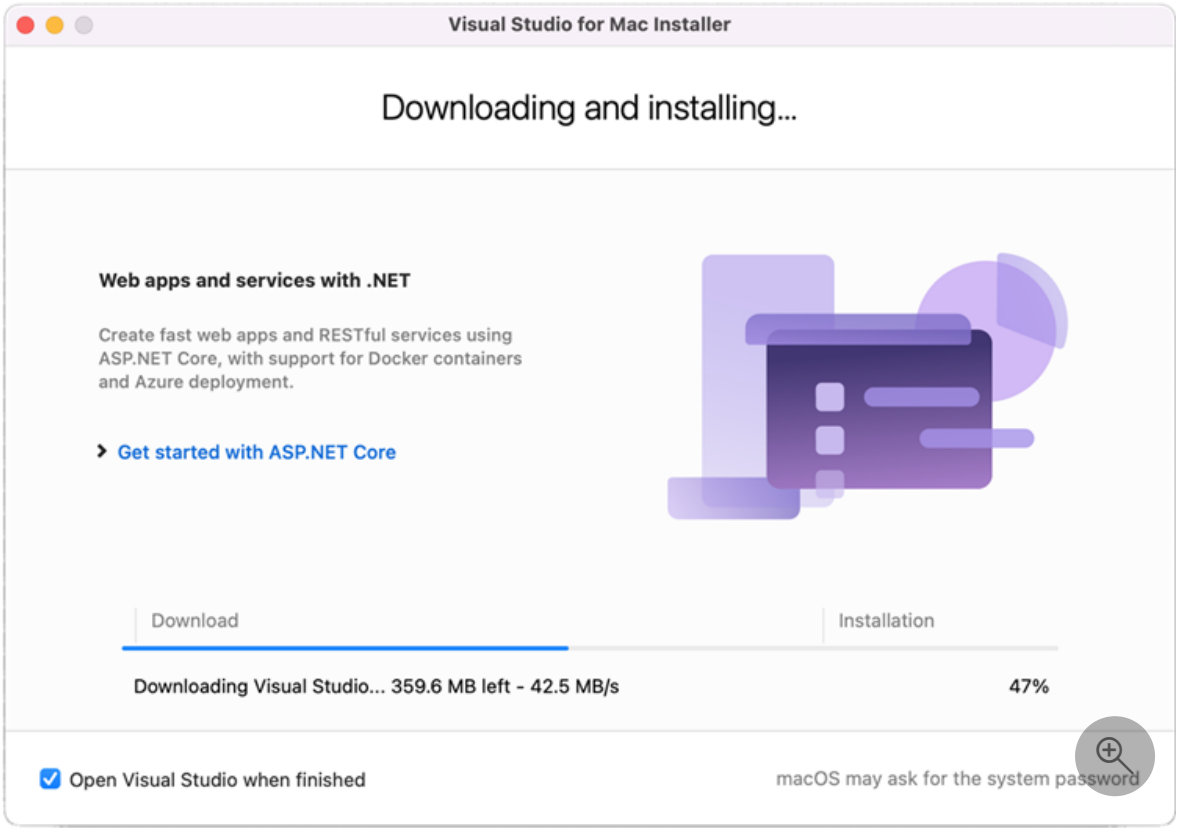
If you do not wish to install all platforms, use the guide below to help you decide which platforms to install:

Type of App	Target	Selection	Notes
Apps Using Xamarin	Xamarin.Forms	Select **Android** and **iOS** platforms	You will need to install Xcode
	iOS only	Select **iOS** platform	You will need to install Xcode
	Android only	Select **Android** platform	Note that you should also select the relevant dependencies
	Mac only	Select **macOS (Cocoa)** platform	You will need to install Xcode
.NET Core applications		Select **.NET Core** platform.	
ASP.NET Core Web Applications		Select **.NET Core** platform.	
Azure Functions		Select **.NET Core** platform.	

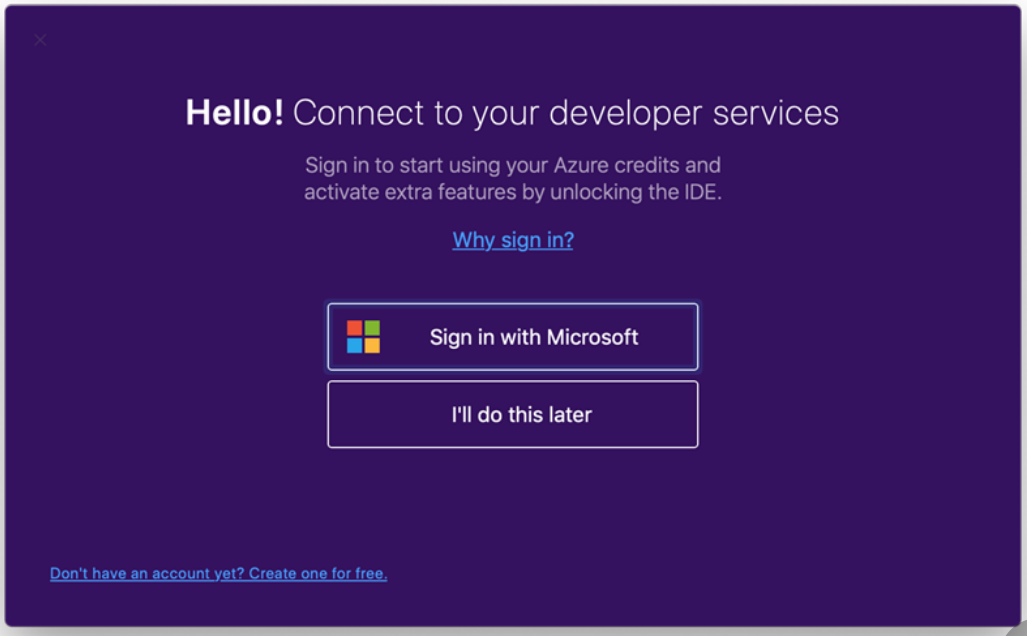
Type of App	Target	Selection	Notes
Cross-platform Unity Game Development		No additional platforms need to be installed beyond Visual Studio for Mac.	Refer to the Unity setup guide for more information on installing the Unity extension.

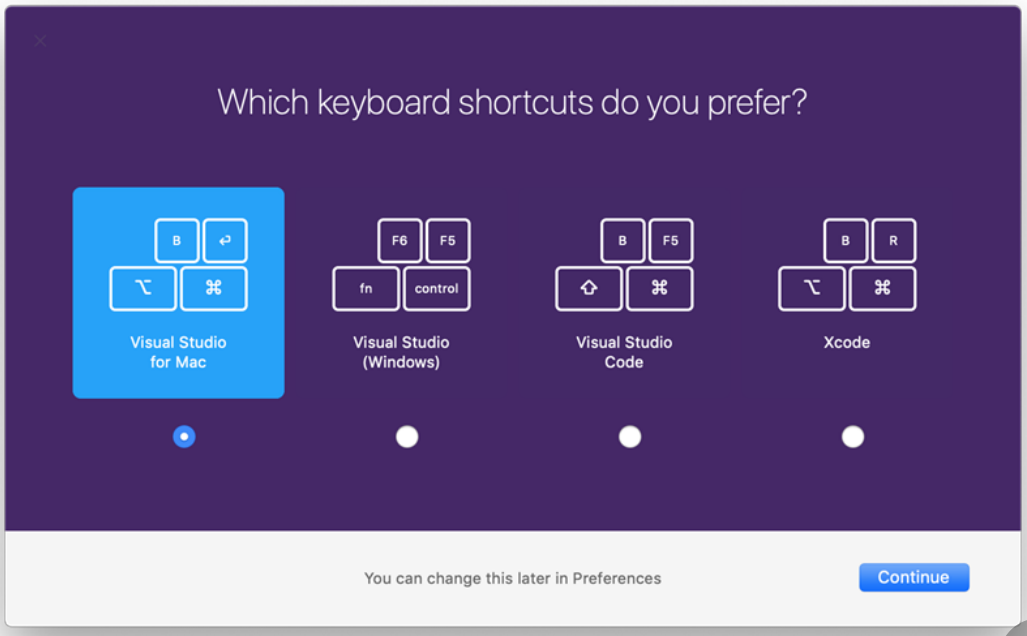
6. After you've made your selections, select the **Install** button.

7. The installer will display progress as it downloads and installs Visual Studio for Mac and the selected workloads. You'll be prompted to enter your password to grant the privileges necessary for installation.

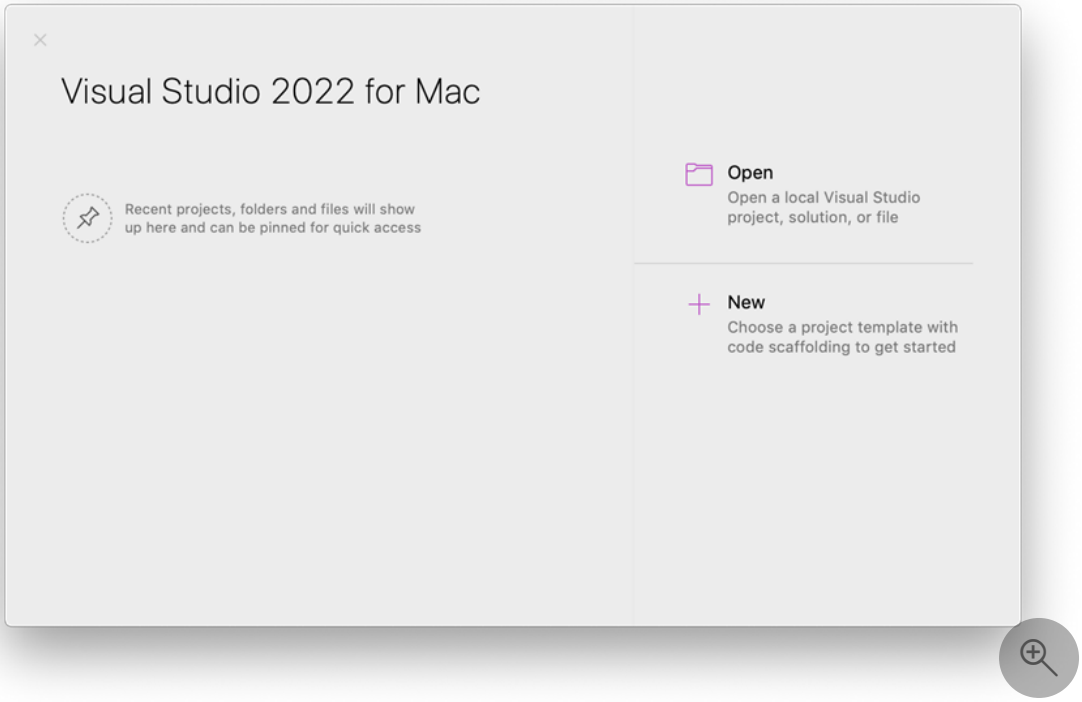


8. Once installed, Visual Studio for Mac will prompt you to personalize your installation by signing in and selecting the key bindings that you'd like to use:





9. Visual Studio for Mac will launch, and you can open a project or create a new one.



If you have network trouble while installing in a corporate environment, review the installing behind a firewall or proxy instructions.

> **Note**
>
> If you chose not to install a platform or tool during the original installation (by unselecting it in step #6), you must run the installer again if you wish to add the components later.

Install Visual Studio for Mac behind a firewall or proxy server

To install Visual Studio for Mac behind a firewall, certain endpoints must be made accessible in order to allow downloads of the required tools and updates for your software.

Configure your network to allow access to the following locations:

- Visual Studio endpoints

Next steps

Installing Visual Studio for Mac allows you to start writing code for your apps. The following guides are provided to guide you through the next steps of writing and deploying your projects.

iOS

- Hello, iOS
- Device Provisioning (To run your application on device)

Android

- Hello, Android
- Using the Xamarin Android SDK Manager
- Android SDK Emulator
- Set Up Device for Development

Xamarin.Forms

Build native cross-platform applications with Xamarin.Forms:

- Xamarin.Forms Quickstarts

.NET Core apps, ASP.NET Core web apps, Unity game development

For other Workloads, refer to the Workloads page.

Related Video

https://learn.microsoft.com/shows/Visual-Studio-Toolbox/Visual-Studio-for-Mac-Acquisition/player

Update Visual Studio for Mac

Article • 12/17/2022 • 2 minutes to read

Applies to: Visual Studio for Mac Visual Studio

Visual Studio for Mac distributes updates for the IDE and supported frameworks regularly. These updates can be in the form of new features, improvements, and bug fixes.

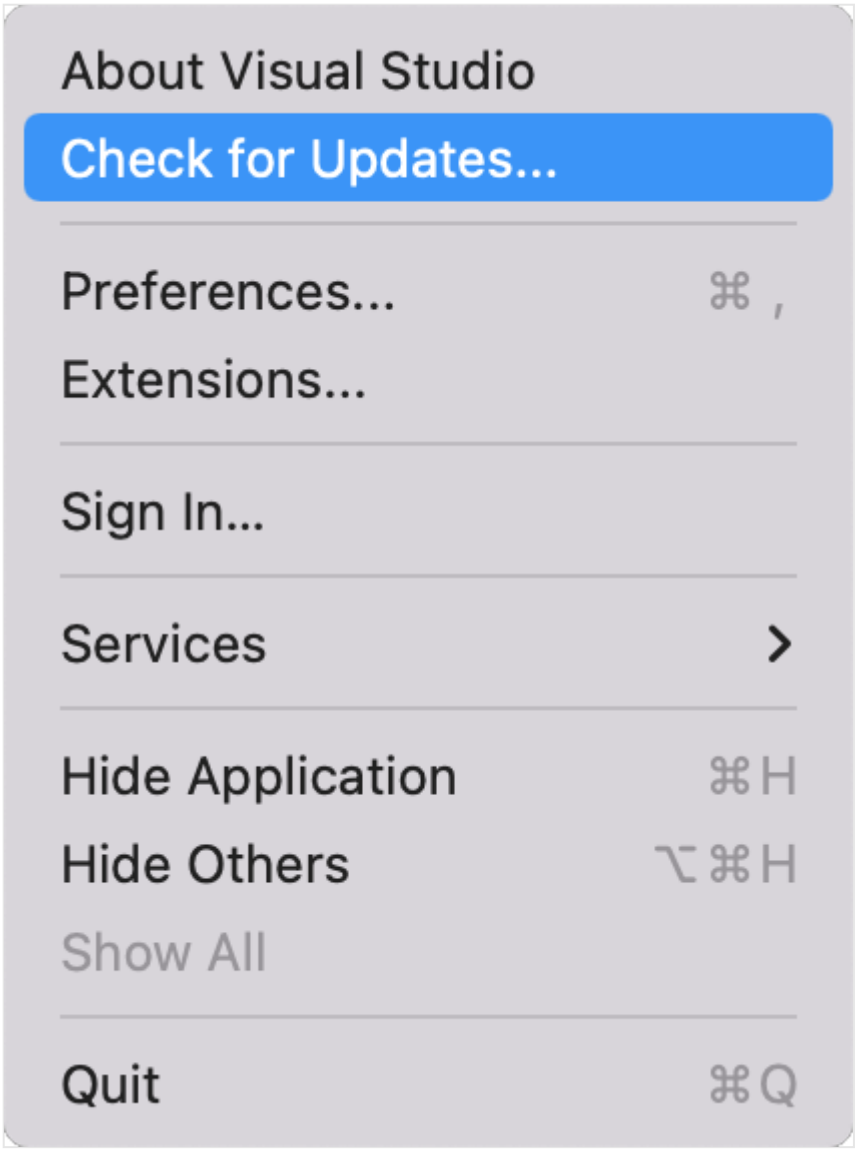
Visual Studio for Mac provides two channels to get these latest versions:

- **Stable** - Provides thoroughly tested updates. This channel is recommended for the best development experience.
- **Preview** - Provides early access to updates that are candidates for release in the Stable Channel. These releases may not be reliable for everyday use.

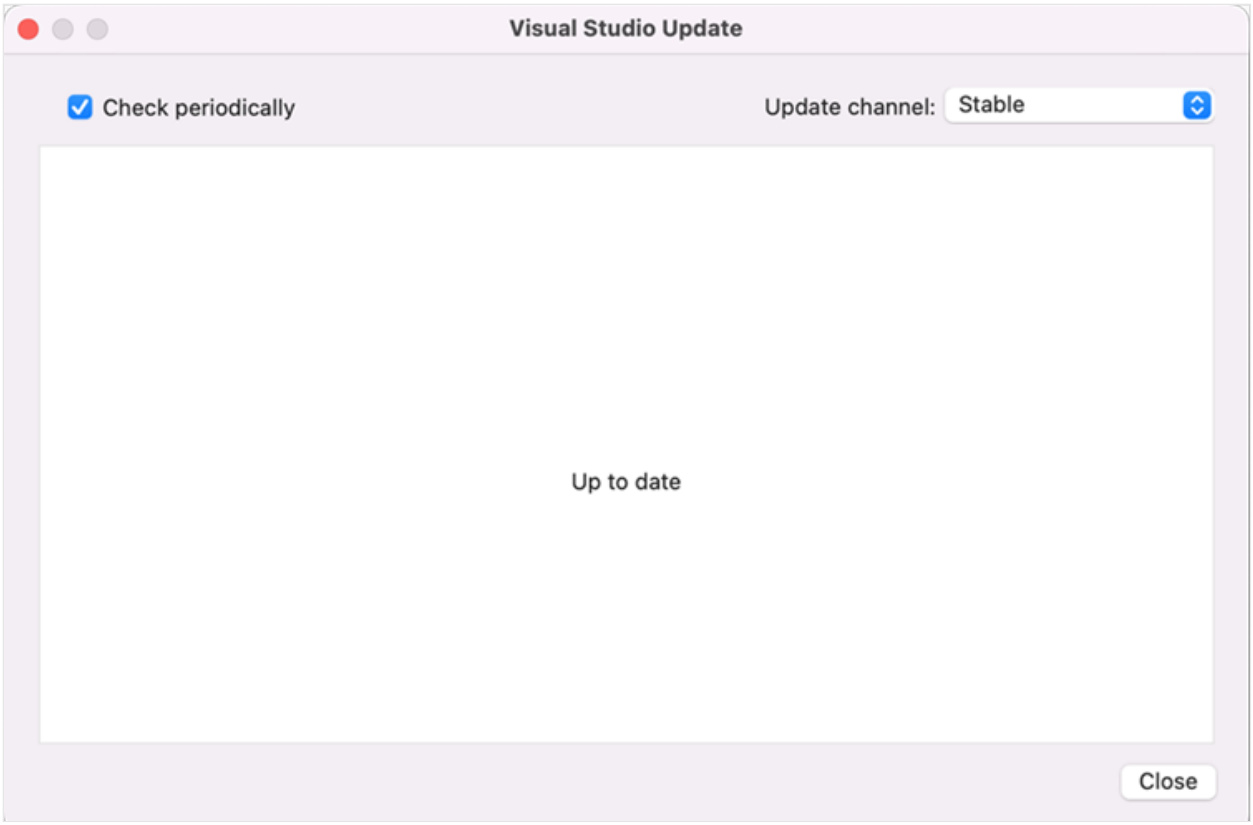
Checking for updates

You can use the **Visual Studio Update** box to check for new updates, change channels, and download and install updates.

To open the **Visual Studio Update**, select **Visual Studio** > **Check for Updates...**:

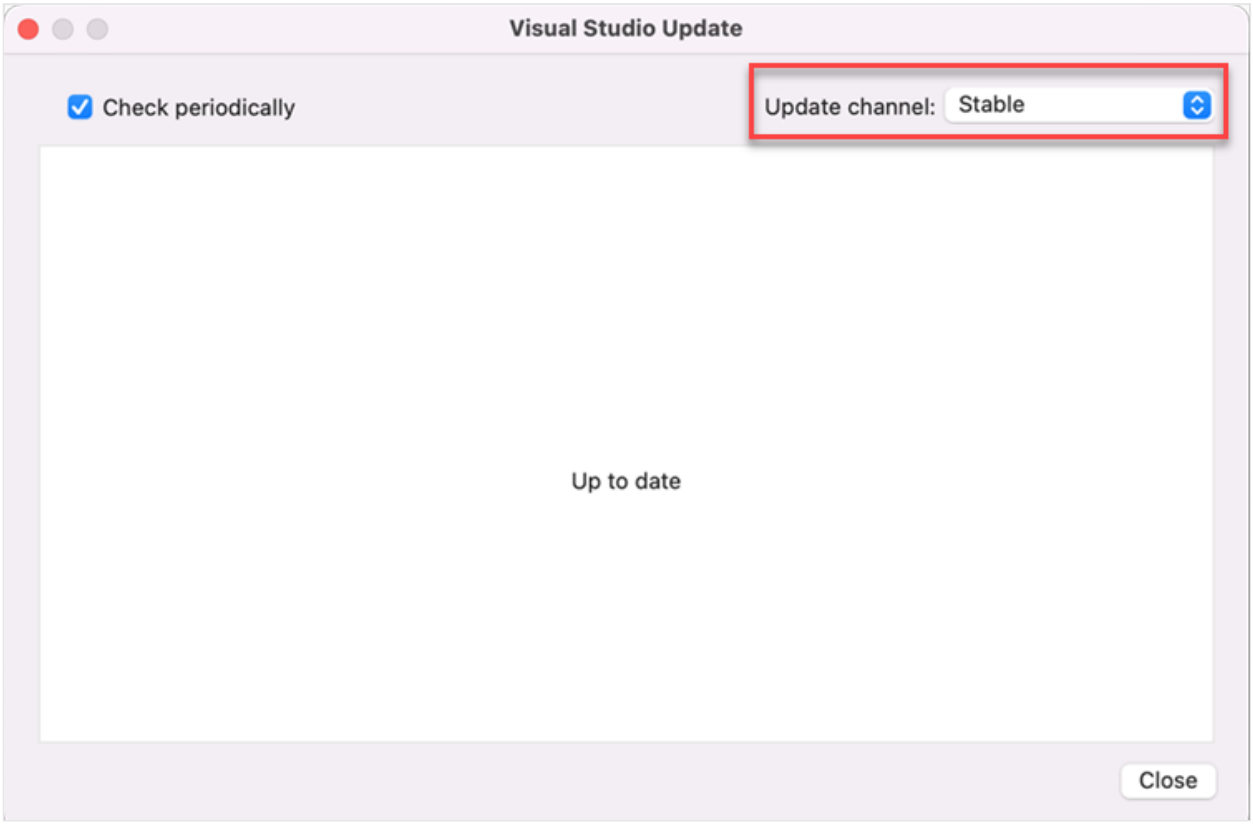


This displays the **Visual Studio Update** box:



Changing the update channel

To change the channel, select it from the **Update channel:** drop-down and select **Switch Channel**:

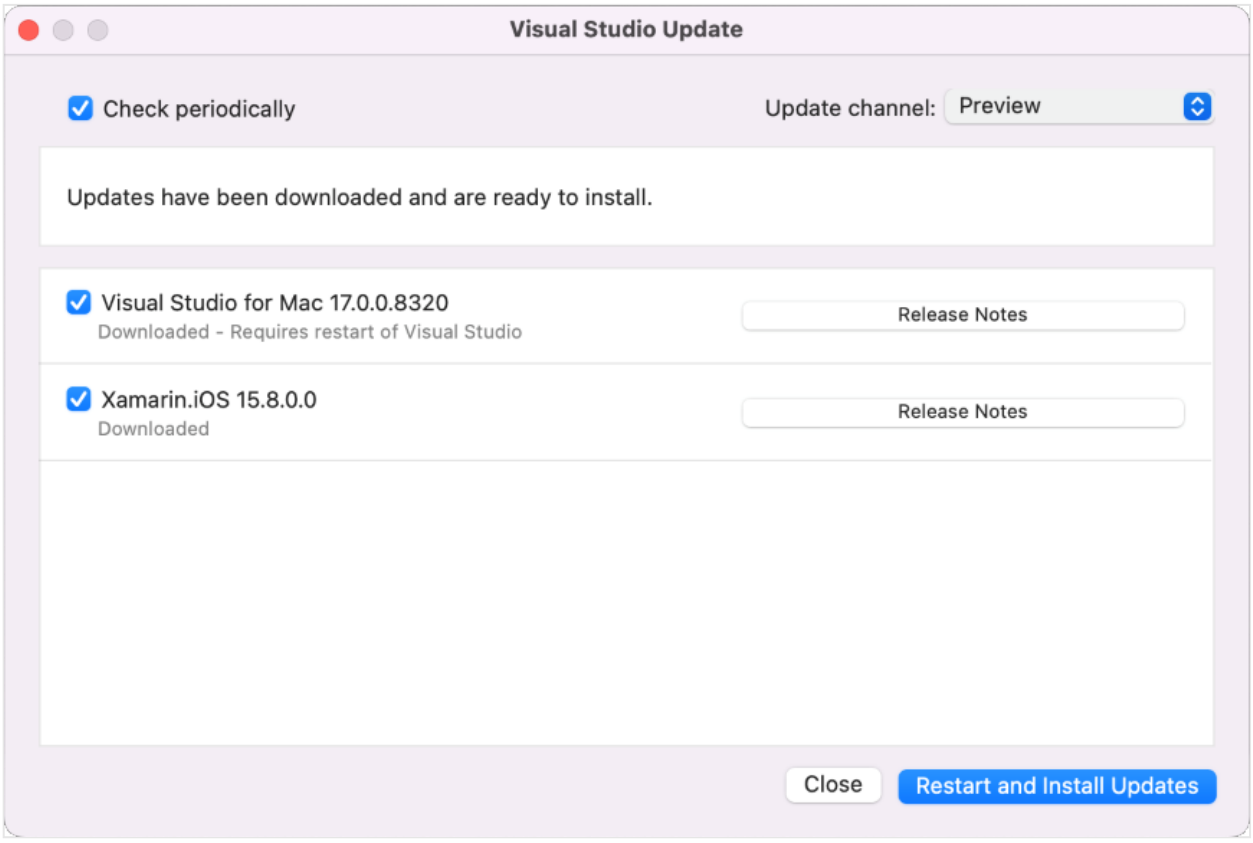


Downloading and installing updates

Switching channels automatically starts the download process of new updates.

If you've selected the option to **Check Automatically**, the **Visual Studio Update** box will pop up when Visual Studio for Mac is open to let you know that new updates are available. New updates will start downloading automatically to your machine when the updater box appears.

Once the updates are downloaded, select **Restart and Install Updates** to start installing the updates:



Depending on the components that need to be installed, you may need to accept more licenses or enter your machine's administrator username and password.

Troubleshooting

If you have issues with the updater, try following the steps in the Updater Troubleshooting guide.

See also

- Update Visual Studio (on Windows)

Uninstall Visual Studio for Mac

Article • 12/17/2022 • 5 minutes to read

Applies to: Visual Studio for Mac Visual Studio

You can use this guide to uninstall each component in Visual Studio for Mac individually by navigating to the relevant section. We recommend you use the scripts provided in the Uninstall scripts section to uninstall everything.

This article is for Visual Studio for Mac. If you're looking for info on VS Code, see Visual Studio Code set-up.

> **Note**
>
> We'd like to learn more about why you're uninstalling Visual Studio for Mac so we can make it better. If you have a few minutes, **please share your feedback**. Thank you!

Uninstall scripts

There are two scripts that can be used to uninstall Visual Studio for Mac and all components from your machine:

- Visual Studio and Xamarin script
- .NET Core script

The following sections provide information on downloading and using the scripts.

Visual Studio for Mac and Xamarin script

You can uninstall Visual Studio and Xamarin components in one go by using the uninstall script.

The uninstall script contains most of the commands that you'll find in the article. There are three main omissions from the script and aren't included due to possible external dependencies. To remove, jump to the relevant section below and remove them manually:

- **Uninstalling Mono**
- **Uninstalling Android AVD**
- **Uninstalling Android SDK and Java SDK**

To run the script, do the following steps:

1. Right-click on the script and select **Save As** to save the file on your Mac.
2. Open Terminal and change the working directory to where the script was downloaded:

   ```Bash
   cd /location/of/file
   ```

3. Make the script executable and run it with **sudo**:

   ```Bash
   chmod +x ./uninstall-vsmac.sh
   sudo ./uninstall-vsmac.sh
   ```

4. Finally, delete the uninstall script and remove Visual Studio for Mac from the dock (if it's there).

.NET Core script

The uninstall script for .NET Core is located in the dotnet cli repo

To run the script, do the following steps:

1. Right-click on the script and select **Save As** to save the file on your Mac.
2. Open Terminal and change the working directory to where the script was downloaded:

   ```Bash
   cd /location/of/file
   ```

3. Make the script executable and then run it with **sudo**:

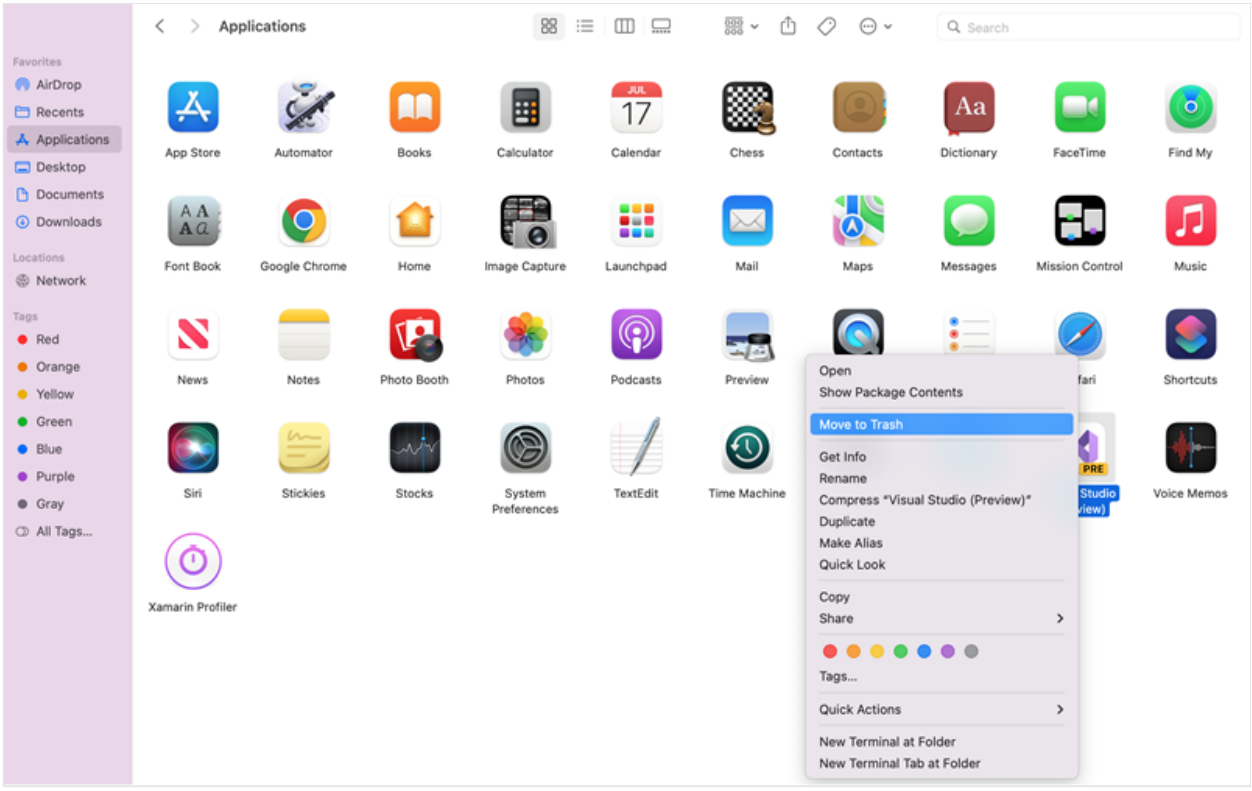
   ```Bash
   chmod +x ./dotnet-uninstall-pkgs.sh
   sudo ./dotnet-uninstall-pkgs.sh
   ```

4. Finally, delete the .NET Core uninstall script.

Manually removing Visual Studio for Mac

If you prefer to remove Visual Studio for Mac and its dependencies manually (instead of using the scripts from the prior section), this section summarizes the steps you should follow.

The first step in uninstalling Visual Studio from a Mac is to locate **Visual Studio** app in the **Applications** directory and drag it to **Trash**. Alternatively, control-click and select **Move to Trash** as illustrated in the following image:



Deleting this app bundle removes Visual Studio for Mac, but there may still be other files such as Xamarin SDKs, .NET SDKs, or iOS development tools on the file system.

To remove all traces of Visual Studio for Mac, run the following commands in Terminal:

Bash

```
sudo rm -rf "/Applications/Visual Studio.app"
rm -rf ~/Library/Caches/VisualStudio
rm -rf ~/Library/Preferences/VisualStudio
rm -rf ~/Library/Preferences/Visual\ Studio
rm -rf ~/Library/Logs/VisualStudio
rm -rf ~/Library/VisualStudio
rm -rf ~/Library/Preferences/Xamarin/
rm -rf ~/Library/Application\ Support/VisualStudio
```

You might also want to remove the following directory containing various Xamarin files and folders. However, this directory contains the Android signing keys. For more

information, see the section **Uninstalling Android SDK and Java SDK**:

```
Bash

rm -rf ~/Library/Developer/Xamarin
```

Uninstall Mono SDK (MDK)

Mono is an open-source implementation of Microsoft's .NET Framework and is used by all Xamarin Products—Xamarin.iOS, Xamarin.Android, and Xamarin.Mac to allow development of these platforms in C#.

> ⚠ **Warning**
>
> There are other applications outside of Visual Studio for Mac that also use Mono, such as Unity. Be sure that there are no other dependencies on Mono before uninstalling it.

To remove the Mono Framework from a machine, run the following commands in Terminal:

```
Bash

sudo rm -rf /Library/Frameworks/Mono.framework
sudo pkgutil --forget com.xamarin.mono-MDK.pkg
sudo rm -rf /etc/paths.d/mono-commands
```

Uninstall Xamarin.Android

Many items are required for the installation and use of Xamarin.Android, such as the Android SDK and Java SDK.

Use the following commands to remove Xamarin.Android:

```
Bash

sudo rm -rf /Developer/MonoDroid
rm -rf ~/Library/MonoAndroid
sudo pkgutil --forget com.xamarin.android.pkg
sudo rm -rf /Library/Frameworks/Xamarin.Android.framework
```

Uninstall Android SDK and Java SDK

The Android SDK is required for development of Android applications. To completely remove all parts of the Android SDK, locate the file at **~/Library/Developer/Xamarin/** and move it to **Trash**.

> ⚠ **Warning**
>
> Note that Android signing keys that are generated by Visual Studio for Mac are located in `~/Library/Developer/Xamarin/Keystore`. Make sure to back these up appropriately, or avoid removing this directory if you wish to keep your keystore.

The Java SDK (JDK) doesn't need to be uninstalled, as it's already pre-packaged as part of macOS.

Uninstall Android AVD

> ⚠ **Warning**
>
> There are other applications outside of Visual Studio for Mac that also use Android AVD and these additional android components, such as Android Studio. Removing this directory might cause projects to break in Android Studio.

To remove any Android AVDs and other Android components use the following command:

```Bash
rm -rf ~/.android
```

To remove only the Android AVDs, use the following command:

```Bash
rm -rf ~/.android/avd
```

Uninstall Xamarin.iOS

Xamarin.iOS allows iOS application development using C# or F# with Visual Studio for Mac.

Use the following commands in Terminal to remove all Xamarin.iOS files from a file system:

```Bash
rm -rf ~/Library/MonoTouch
sudo rm -rf /Library/Frameworks/Xamarin.iOS.framework
sudo rm -rf /Developer/MonoTouch
sudo pkgutil --forget com.xamarin.monotouch.pkg
sudo pkgutil --forget com.xamarin.xamarin-ios-build-host.pkg
sudo pkgutil --forget com.xamarin.xamarin.ios.pkg
```

Uninstall Xamarin.Mac

Xamarin.Mac can be removed from your machine by using the following two commands that remove the product and license from your Mac respectively:

```Bash
sudo rm -rf /Library/Frameworks/Xamarin.Mac.framework
rm -rf ~/Library/Xamarin.Mac
```

Uninstall Workbooks and Inspector

Starting with 1.2.2, Xamarin Workbooks & Inspector can be uninstalled by running the following command in Terminal:

```Bash
sudo
/Library/Frameworks/Xamarin.Interactive.framework/Versions/Current/uninstall
```

For older versions, you need to manually remove the following artifacts:

- Delete the Workbooks app at `"/Applications/Xamarin Workbooks.app"`
- Delete the Inspector app at `"Applications/Xamarin Inspector.app"`
- Delete the add-ins: `"~/Library/Application Support/XamarinStudio-6.0/LocalInstall/Addins/Xamarin.Interactive"` and `"~/Library/Application Support/XamarinStudio-6.0/LocalInstall/Addins/Xamarin.Inspector"`
- Delete Inspector and supporting files here: `/Library/Frameworks/Xamarin.Interactive.framework` and `/Library/Frameworks/Xamarin.Inspector.framework`

Uninstall the Xamarin Profiler

Bash

```
sudo rm -rf "/Applications/Xamarin Profiler.app"
```

Uninstall the Visual Studio Installer

Use the following commands to remove all traces of the Xamarin Universal Installer:

Bash

```
rm -rf ~/Library/Caches/XamarinInstaller/
rm -rf ~/Library/Caches/VisualStudioInstaller/
rm -rf ~/Library/Logs/XamarinInstaller/
rm -rf ~/Library/Logs/VisualStudioInstaller/
rm -rf ~/Library/Preferences/Xamarin/
rm -rf "~/Library/Preferences/Visual Studio/"
```

See also

- Uninstall Visual Studio (on Windows)
- Uninstall Visual Studio Code

Xamarin firewall configuration instructions

Article • 05/28/2020 • 2 minutes to read

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

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

Endpoints to allow

Xamarin installer

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

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

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

NuGet (including Xamarin.Forms)

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

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

Software updates

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

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

Xamarin Mac Agent

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

Xamarin.Forms supported platforms

Article • 07/08/2021 • 2 minutes to read

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

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

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

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

> **Note**
>
> Developing apps using Xamarin.Forms requires familiarity with **.NET Standard**.

Additional platform support

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

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

The status of these platforms is available on the Xamarin.Forms GitHub platform support wiki.

Android platform support

You should have the latest Android SDK Tools and Android API platform installed. You can update to the latest versions using the Android SDK Manager.

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

Visual Studio

Project Options > Application > Application Properties

Deprecated platforms

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

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

Build your first Xamarin.Forms App

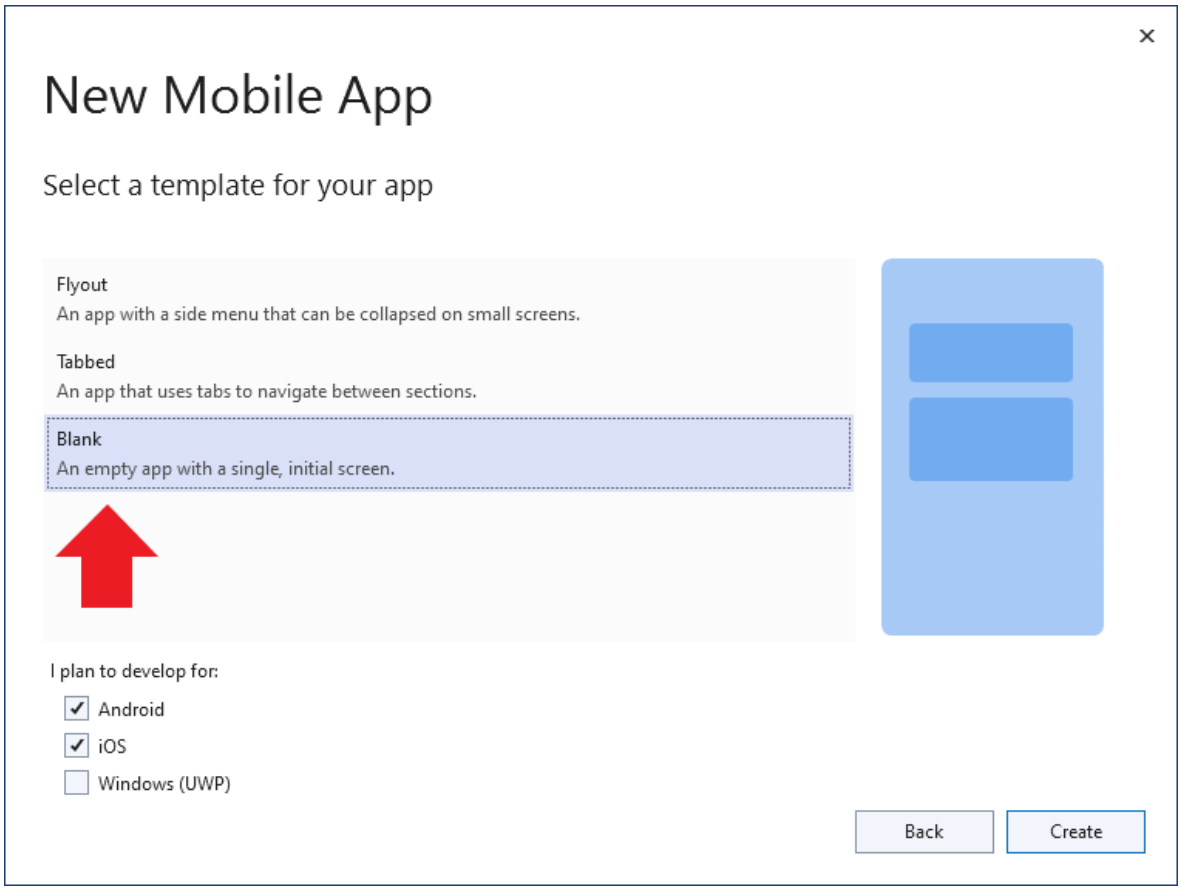
Article • 09/22/2022 • 4 minutes to read

Step-by-step instructions for Windows

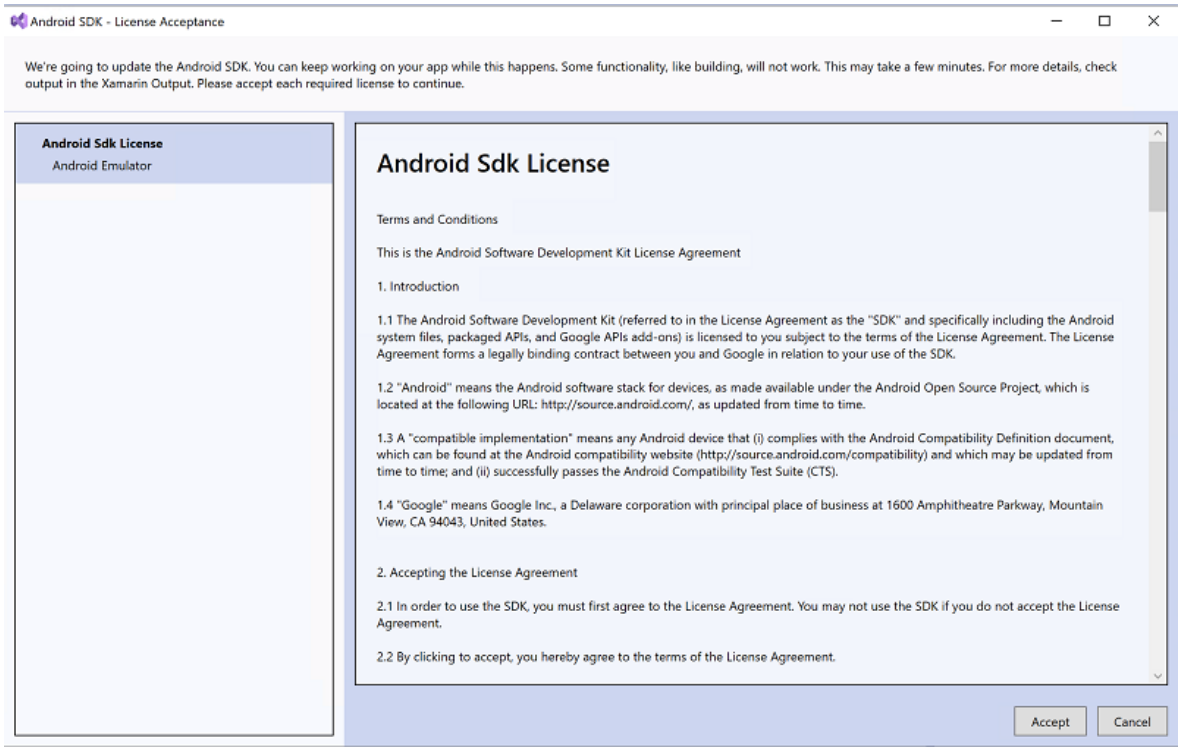
Download the sample

Follow these steps along with the video above:

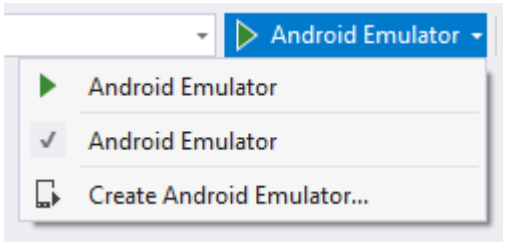
1. Choose **File** > **New** > **Project...** or press the **Create new project...** button.
2. Search for "Xamarin" or choose **Mobile** from the **Project type** menu. Select the **Mobile App (Xamarin.Forms)** project type.
3. Choose a project name – the example uses "AwesomeApp".
4. Click on the **Blank** project type and ensure **Android** and **iOS** are selected:



5. Wait until the NuGet packages are restored (a "Restore completed" message will appear in the status bar).
6. New Visual Studio 2022 installations won't have Android SDKs installed, you may be prompted to install the most recent Android SDK:

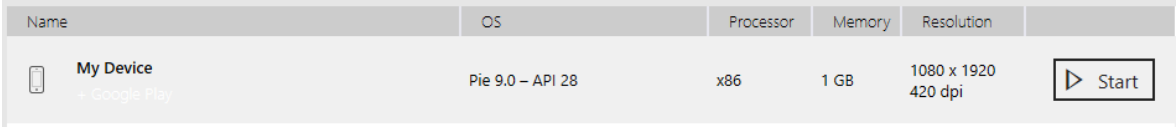


7. New Visual Studio 2022 installations won't have an Android emulator configured. Click the dropdown arrow on the **Debug** button and choose **Create Android Emulator** to launch the emulator creation screen:

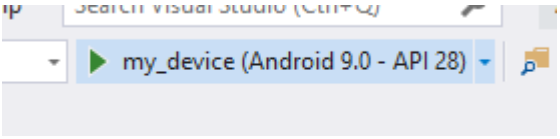


8. In the emulator creation screen, use the default settings and click the **Create** button:

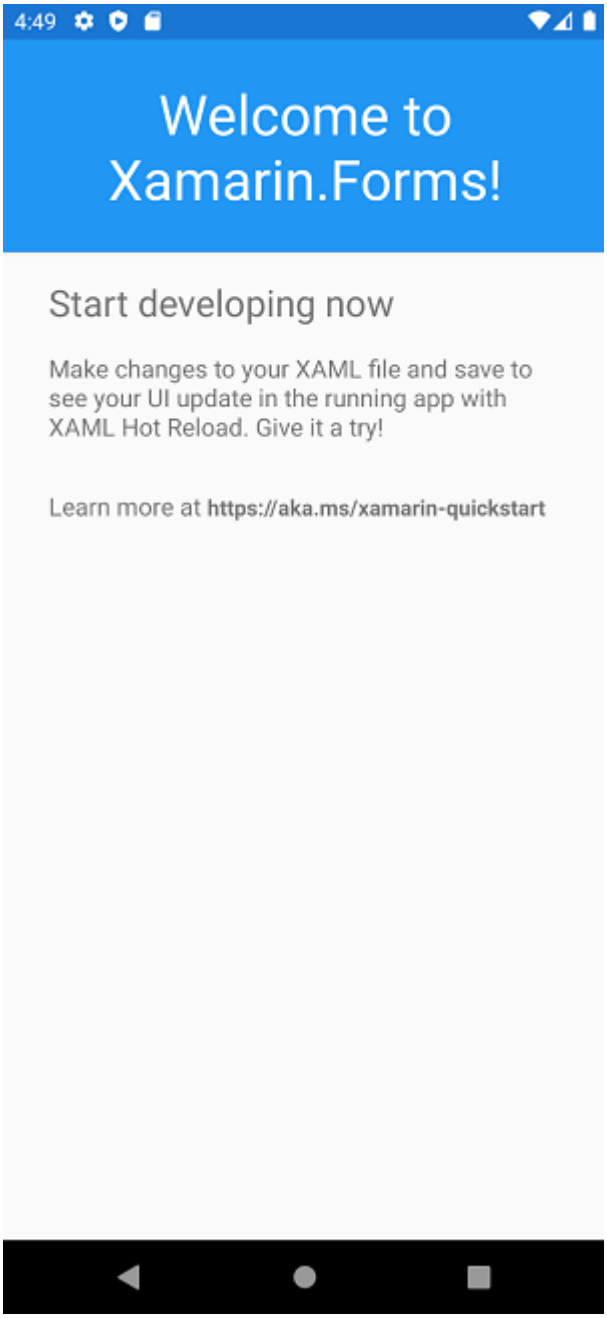
9. Creating an emulator will return you to the Device Manager window. Click the **Start** button to launch the new emulator:



10. Visual Studio 2022 should now show the name of the new emulator on the **Debug** button:



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



Customize the application

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

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

   XAML

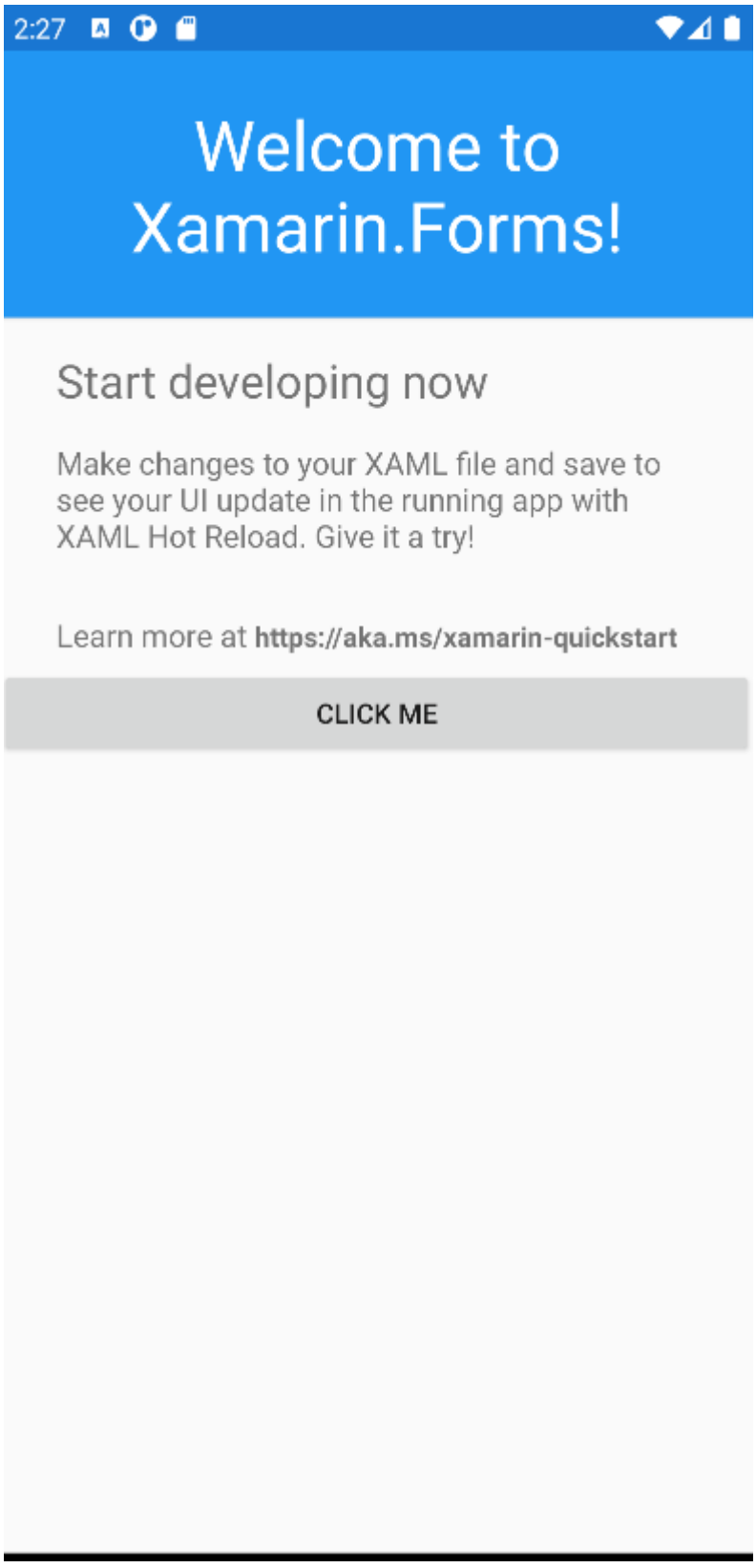
   ```
   <Button Text="Click Me" Clicked="Button_Clicked" />
   ```

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

   C#

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

3. Debug the app on Android:



Build an iOS app in Visual Studio 2022

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

You can download the completed code from the samples gallery or view it on GitHub.

Next Steps

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

Xamarin.Forms quickstarts

Article • 02/08/2021 • 2 minutes to read

Learn how to create mobile applications with Xamarin.Forms.

Create a Xamarin.Forms application

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

Perform navigation in a Xamarin.Forms application

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

Store data in a local SQLite.NET database

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

Style a cross-platform Xamarin.Forms application

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

Quickstart deep dive

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

Create a Xamarin.Forms application quickstart

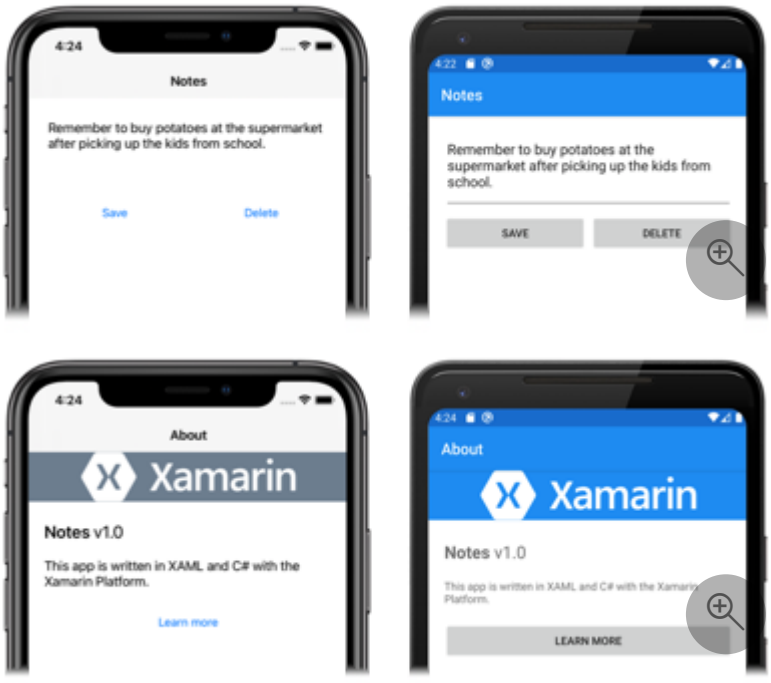
Article • 07/08/2021 • 17 minutes to read

Download the sample

In this quickstart, you will learn how to:

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

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



Prerequisites

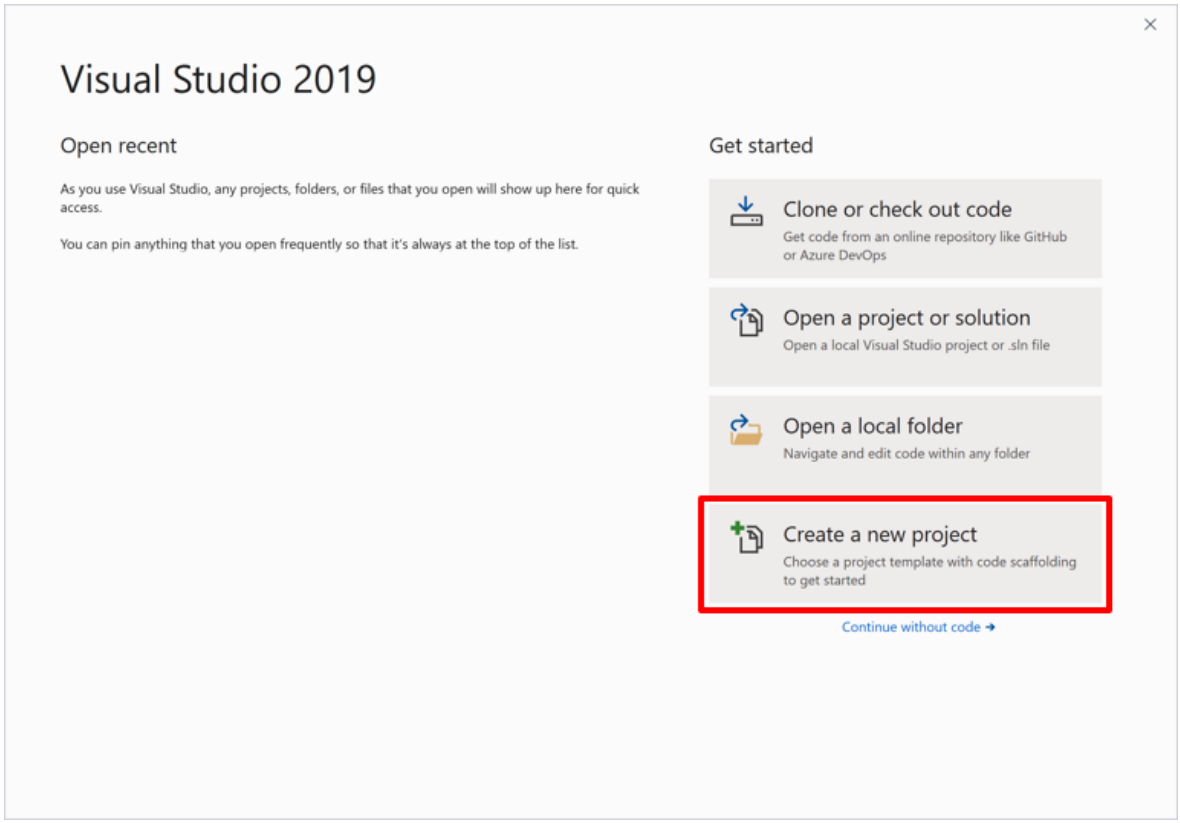
- Visual Studio 2019 (latest release), with the **Mobile development with .NET** workload installed.
- Knowledge of C#.
- (optional) A paired Mac to build the application on iOS.

For more information about these prerequisites, see Installing Xamarin. For information about connecting Visual Studio 2019 to a Mac build host, see Pair to Mac for

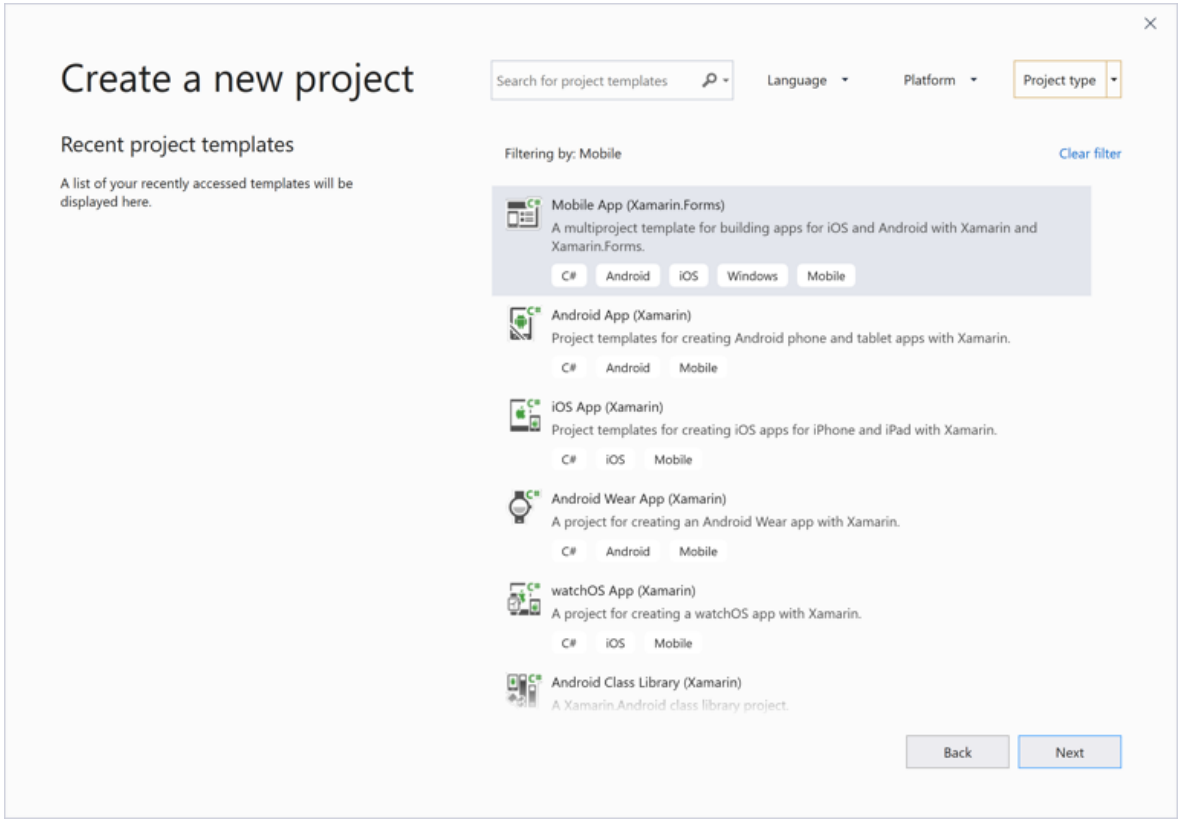
Xamarin.iOS development.

Get started with Visual Studio 2019

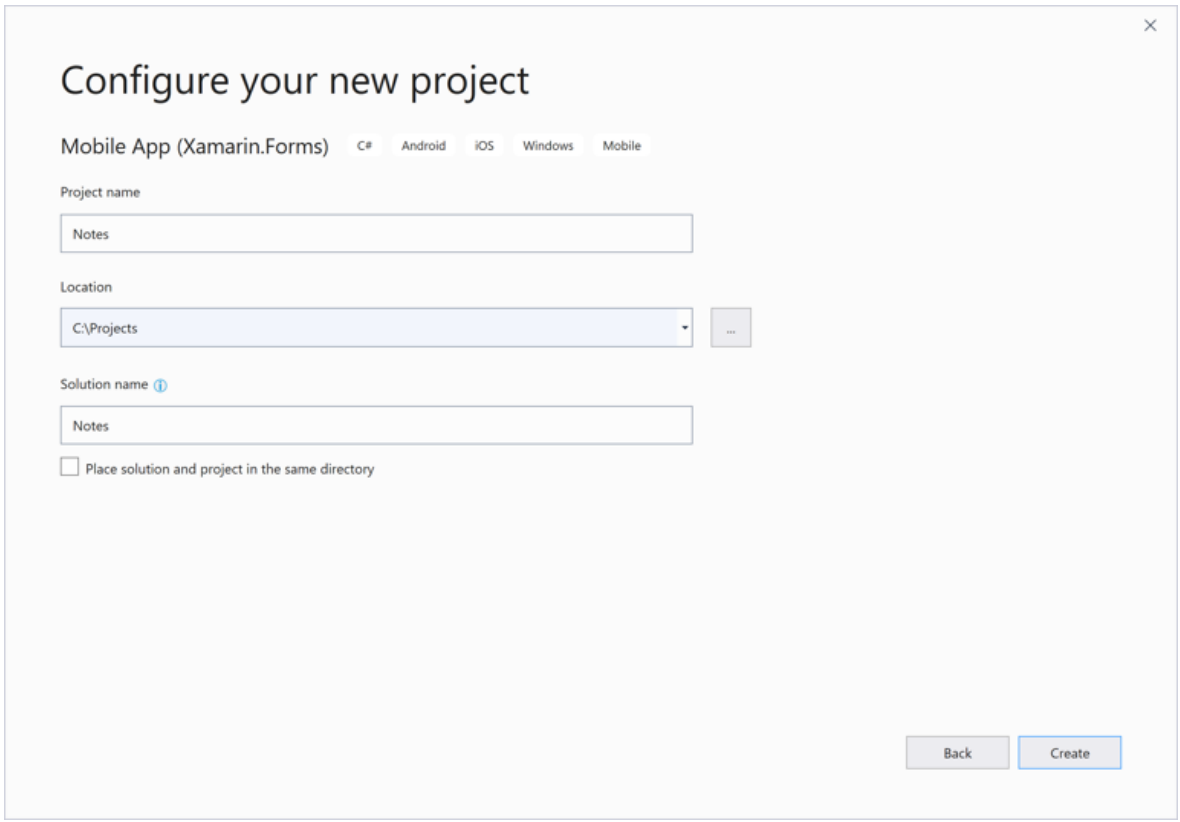
1. Launch Visual Studio 2019, and in the start window click **Create a new project** to create a new project:



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



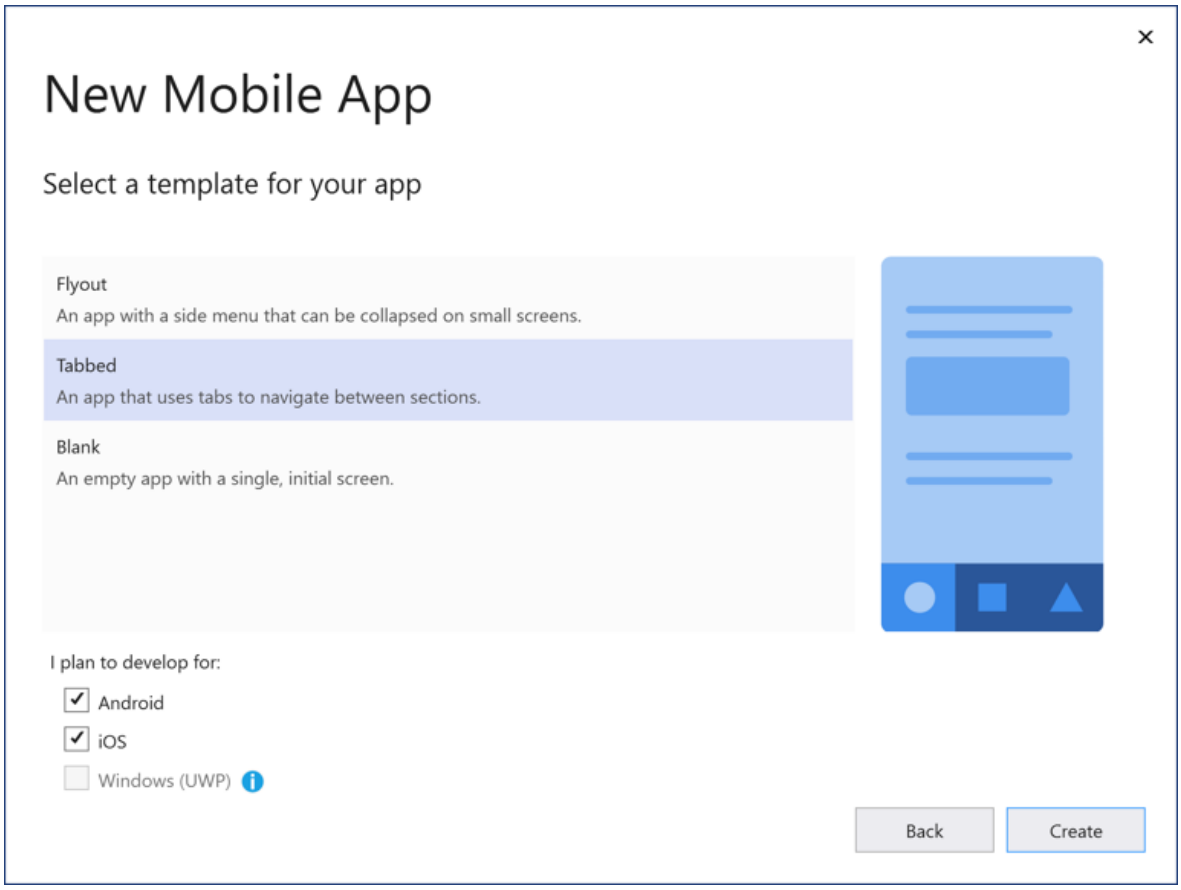
3. In the **Configure your new project** window, set the **Project name** to **Notes**, choose a suitable location for the project, and click the **Create** button:



> **ⓘ Important**
>
> The C# and XAML snippets in this quickstart requires that the solution and project are both named **Notes**. Using a different name will result in build

errors when you copy code from this quickstart into the project.

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



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

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

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

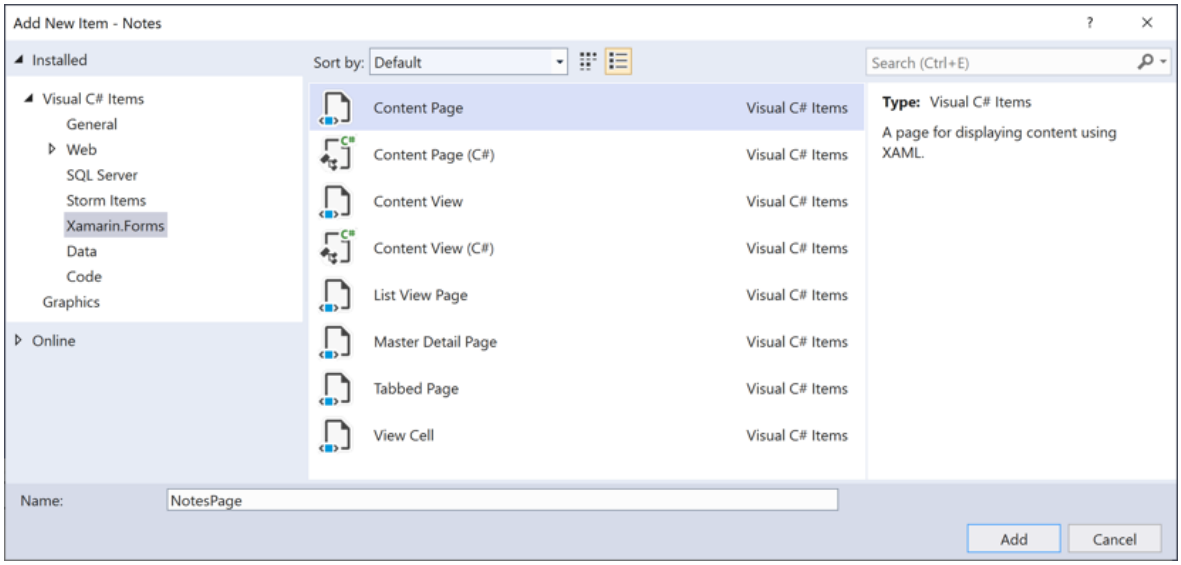
   - **Models**
   - **Services**
   - **ViewModels**
   - **Views**

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

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

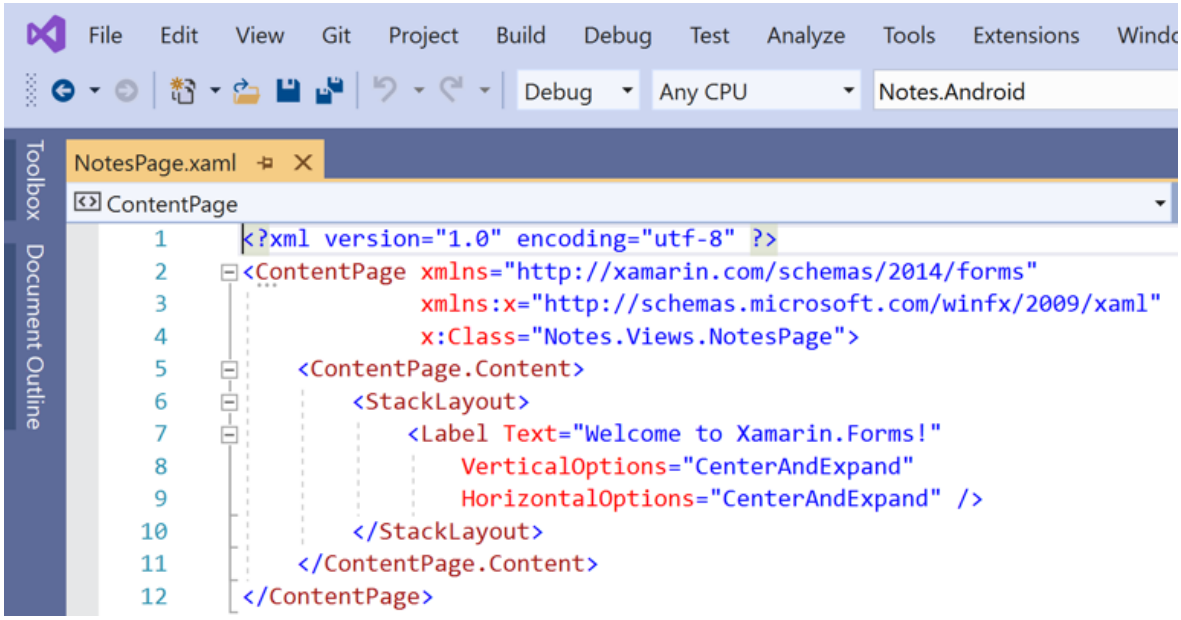
8. In **Solution Explorer**, in the **Notes** project, select the **Views** folder, right-click, and select **Add** > **New Item...**. In the **Add New Item** dialog, select **Visual C# Items** >

Xamarin.Forms > **Content Page**, name the new file **NotesPage**, and click the **Add** button:



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

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



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

XAML

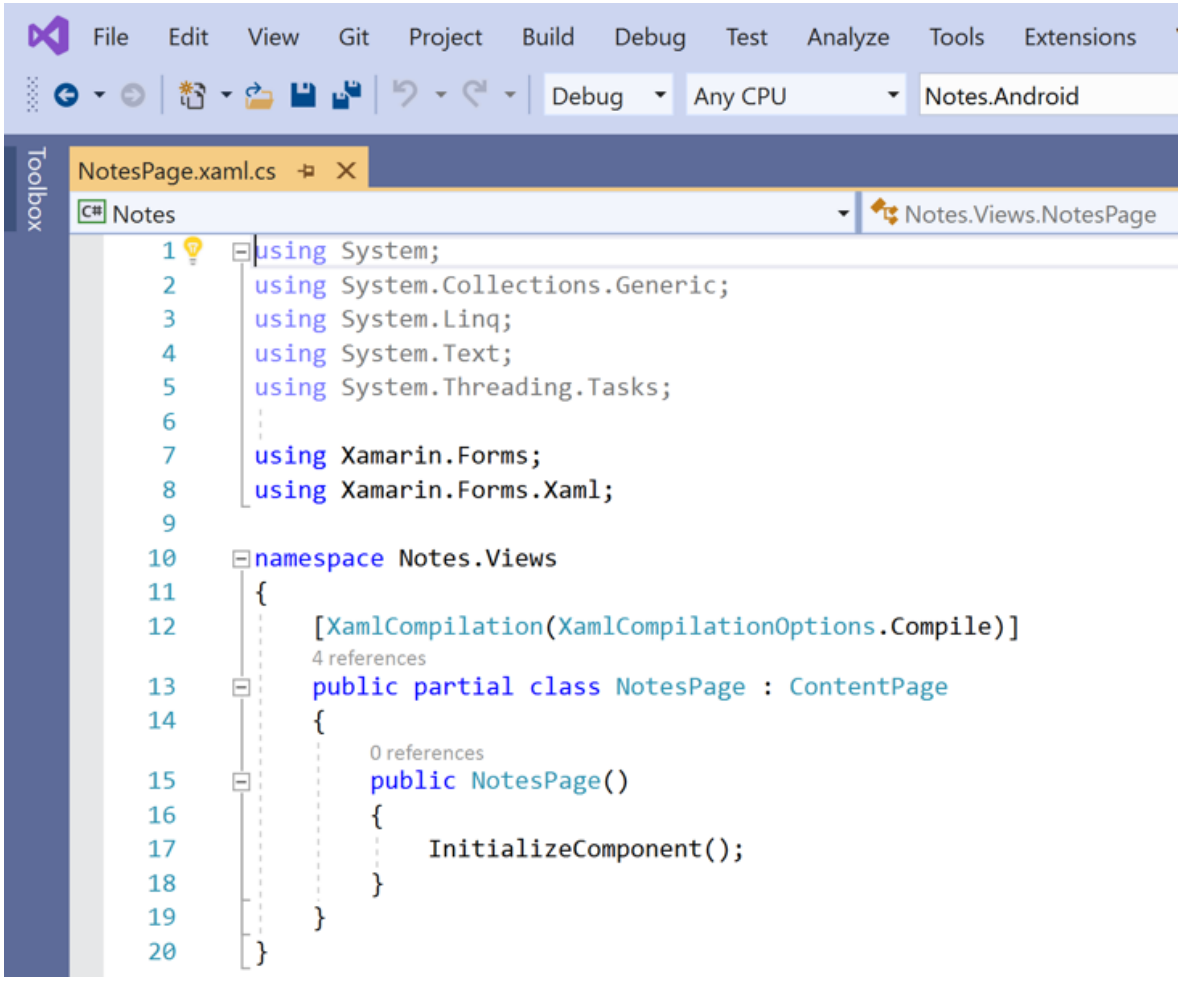
```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Notes.Views.NotesPage"
             Title="Notes">
    <!-- Layout children vertically -->
```

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

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

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

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



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

C#

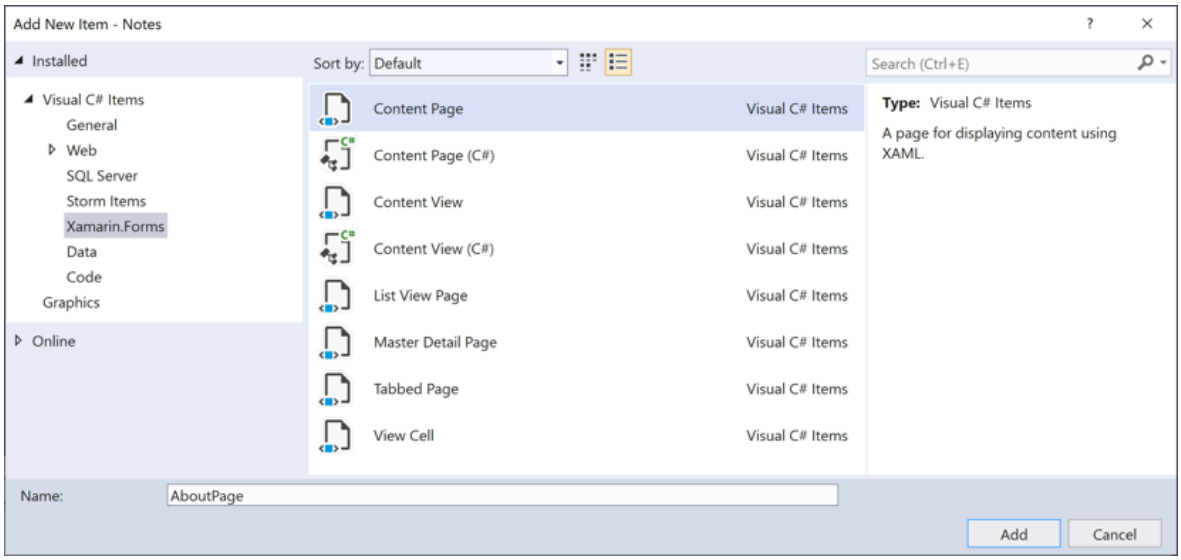
```
using System;
using System.IO;
using Xamarin.Forms;

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

        public NotesPage()
        {
            InitializeComponent();

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

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

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

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

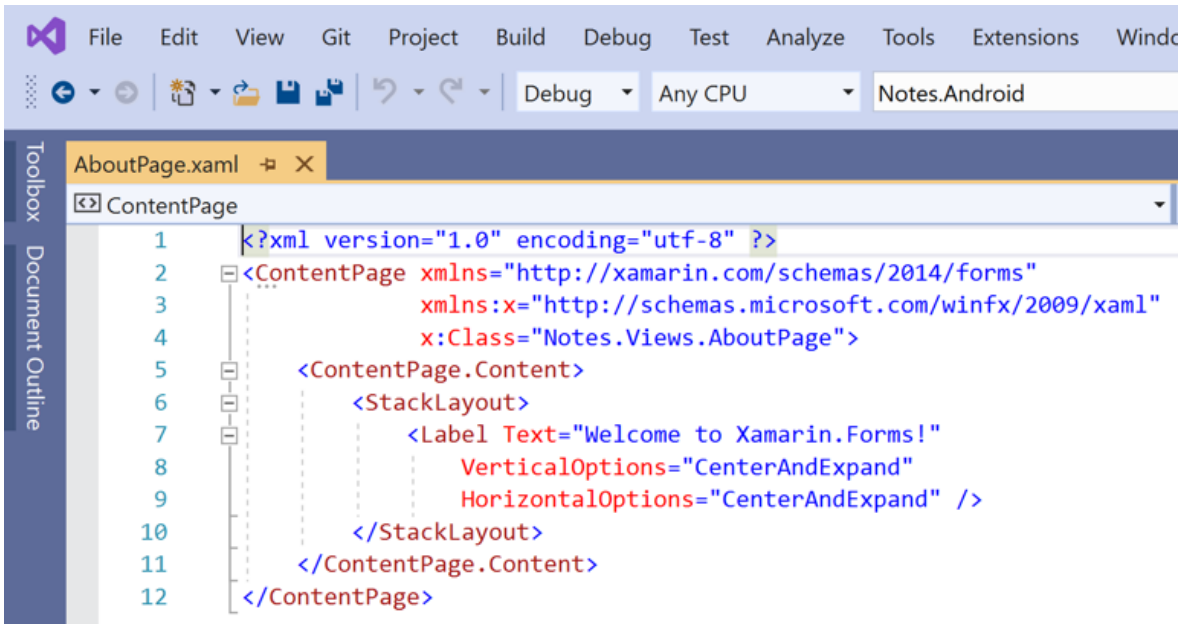
Save the changes to **NotesPage.xaml.cs** by pressing **CTRL+S**.

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



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

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



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

XAML

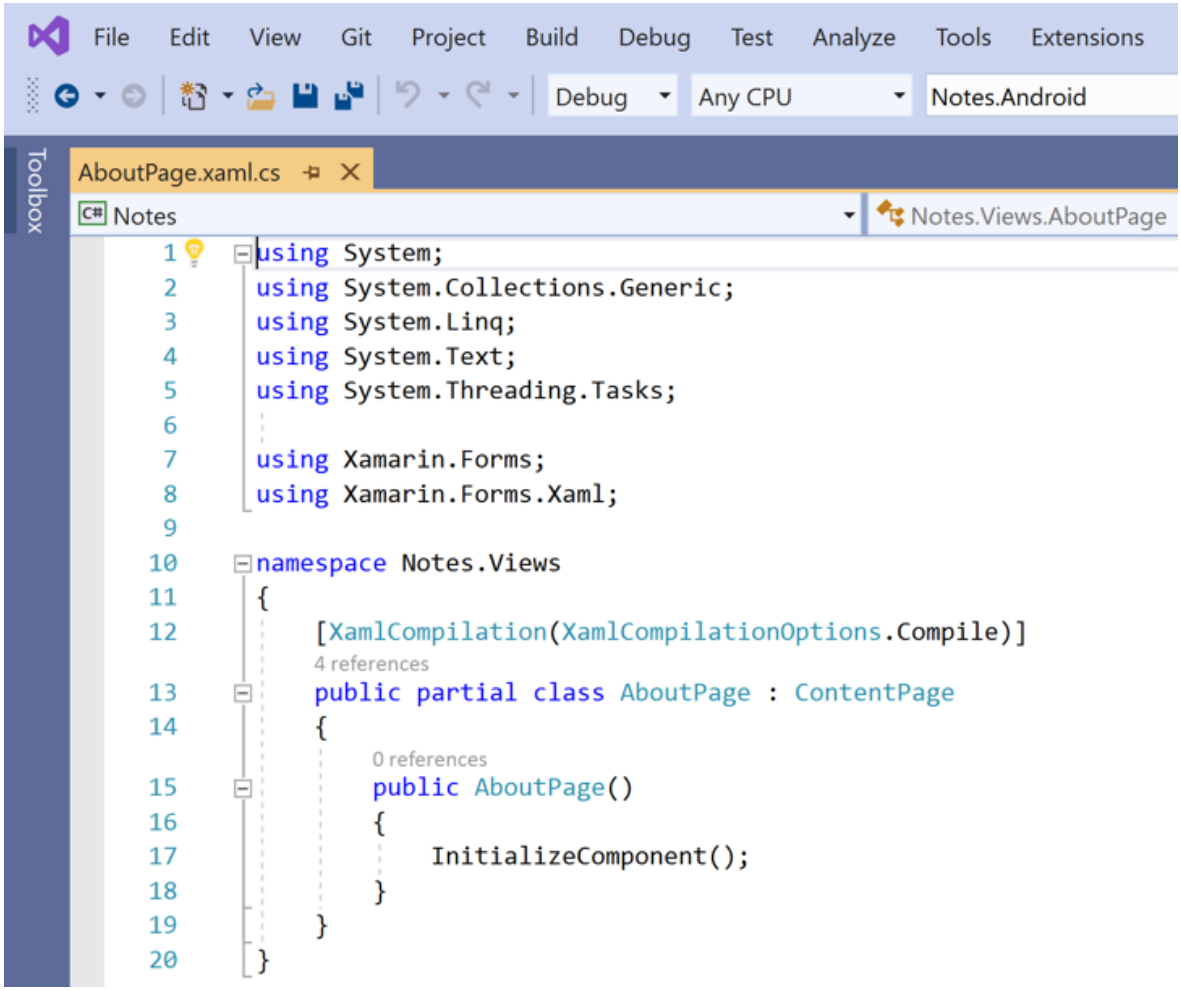
```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Notes.Views.AboutPage"
             Title="About">
    <!-- Layout children in two rows -->
    <Grid RowDefinitions="Auto,*">
        <Image Source="xamarin_logo.png"
               BackgroundColor="{OnPlatform iOS=LightSlateGray,
Android=#2196F3}"
```

```
                VerticalOptions="Center"
                HeightRequest="64" />
        <!-- Layout children vertically -->
        <StackLayout Grid.Row="1"
                     Margin="20"
                     Spacing="20">
            <Label FontSize="22">
                <Label.FormattedText>
                    <FormattedString>
                        <FormattedString.Spans>
                            <Span Text="Notes"
                                  FontAttributes="Bold"
                                  FontSize="22" />
                            <Span Text=" v1.0" />
                        </FormattedString.Spans>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
            <Label Text="This app is written in XAML and C# with the
Xamarin Platform." />
            <Button Text="Learn more"
                    Clicked="OnButtonClicked" />
        </StackLayout>
    </Grid>
</ContentPage>
```

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

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

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



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

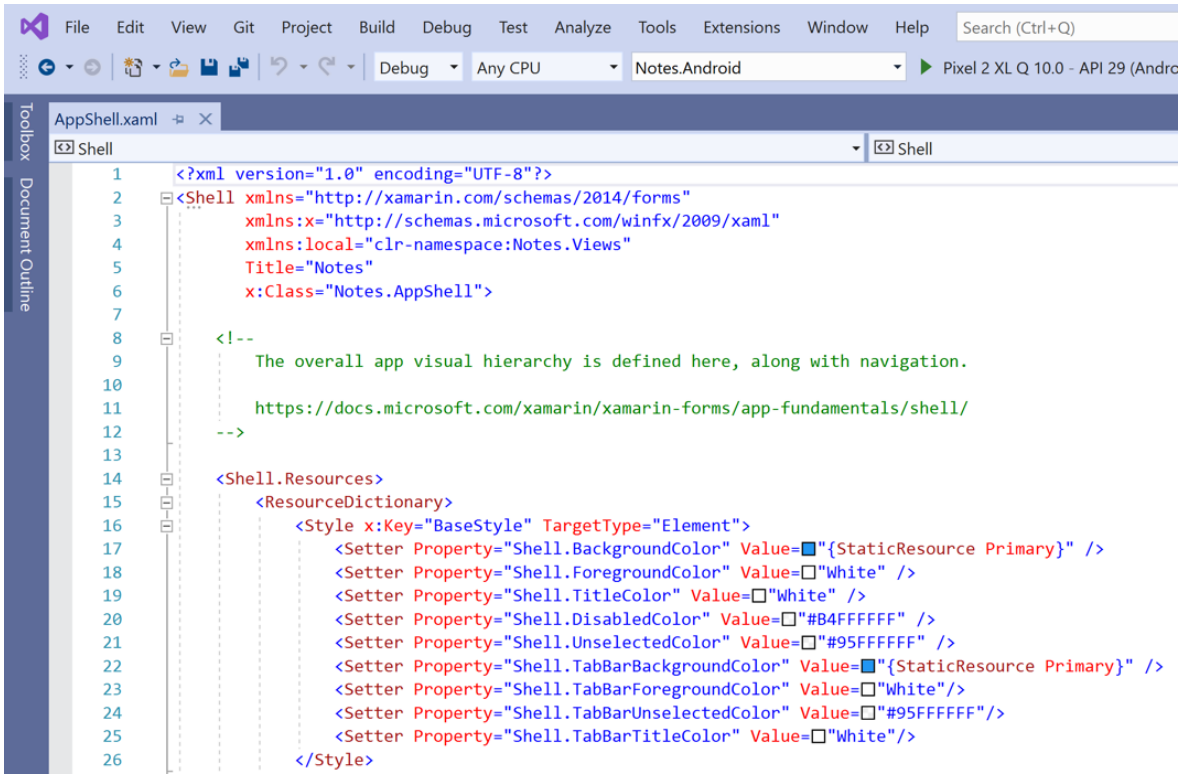
C#

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

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

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

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

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

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



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

XAML

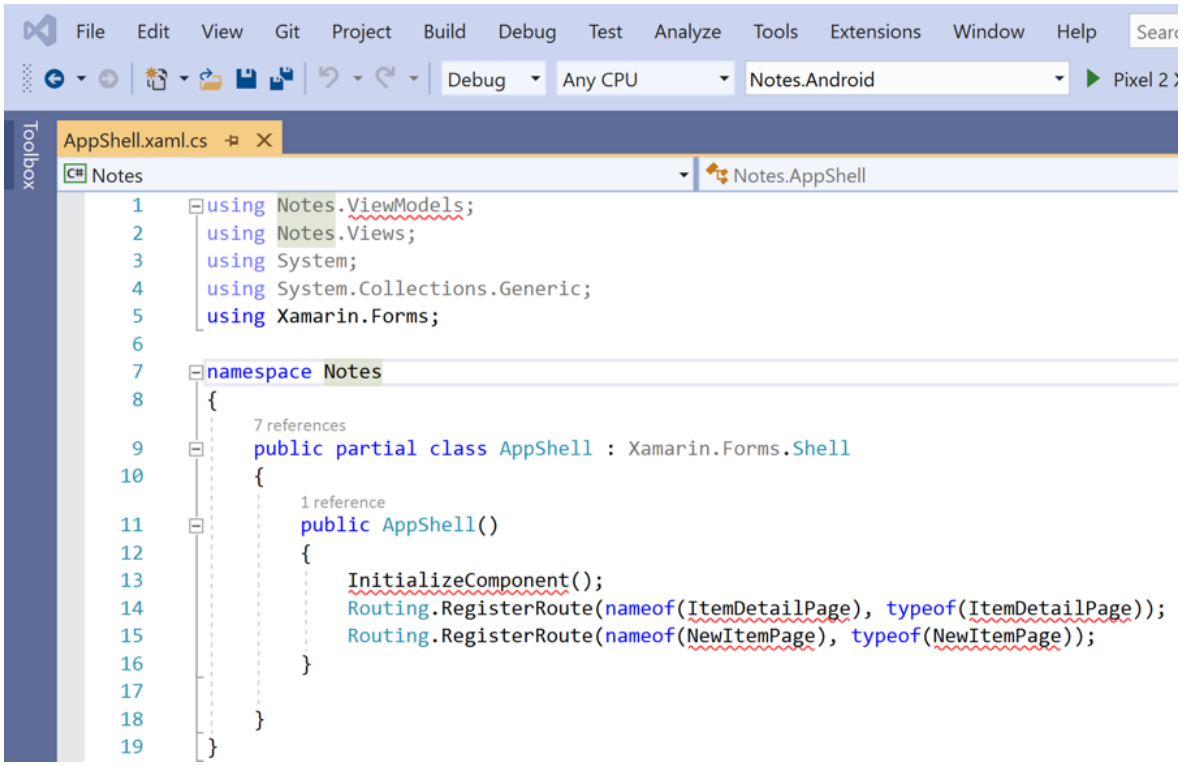
```
<?xml version="1.0" encoding="UTF-8"?>
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:views="clr-namespace:Notes.Views"
       x:Class="Notes.AppShell">
    <!-- Display a bottom tab bar containing two tabs -->
    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

```
    </TabBar>
</Shell>
```

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

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

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



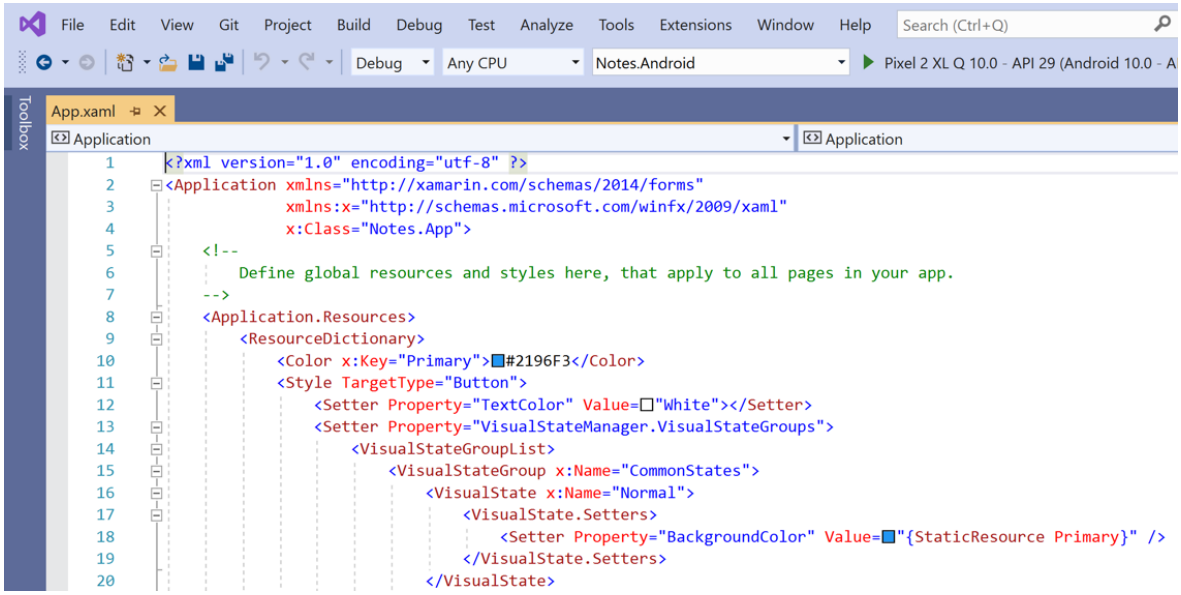
21. In **AppShell.xaml.cs**, remove all of the template code and replace it with the following code:

```
C#

using Xamarin.Forms;

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

```
        }
    }
}
```

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

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



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

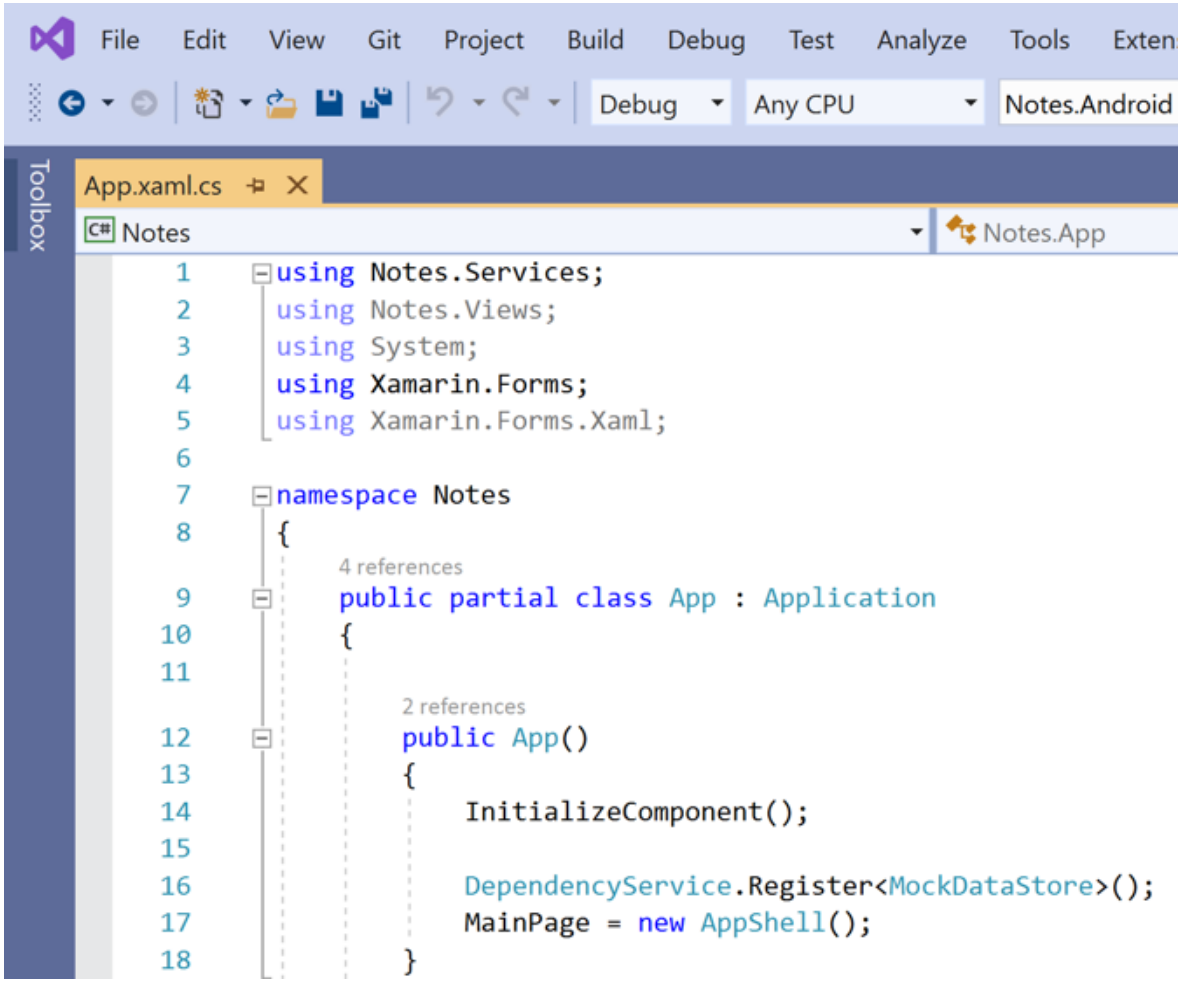
XAML

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

</Application>
```

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

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

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



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

C#

```
using Xamarin.Forms;

namespace Notes
{
    public partial class App : Application
    {

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

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

        protected override void OnResume()
```

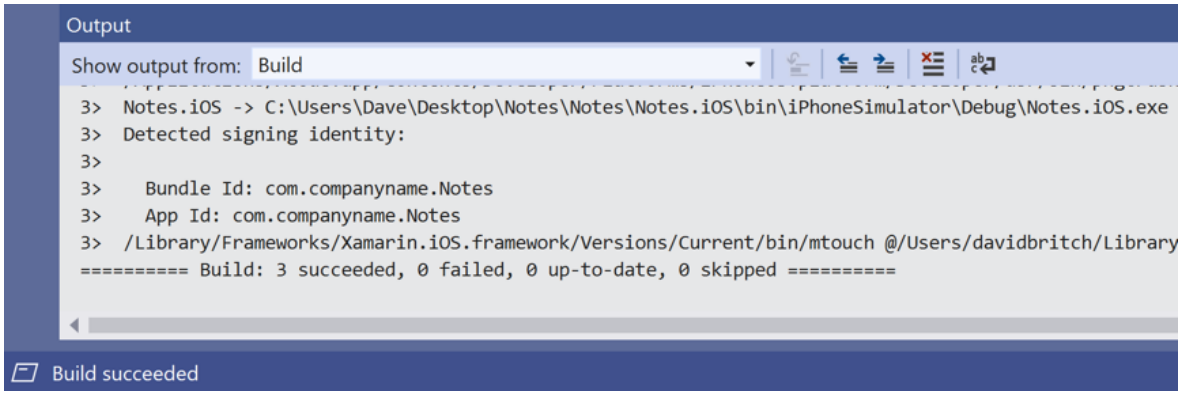
```
        {
        }
    }
}
```

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

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

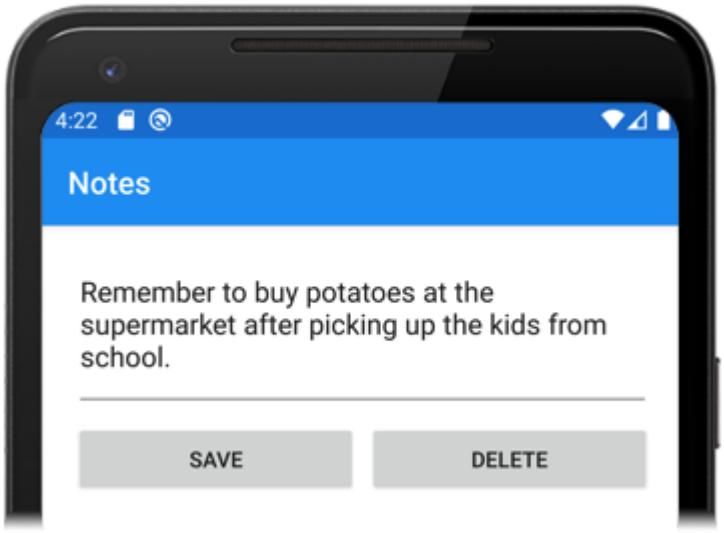
Building the quickstart

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



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

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



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

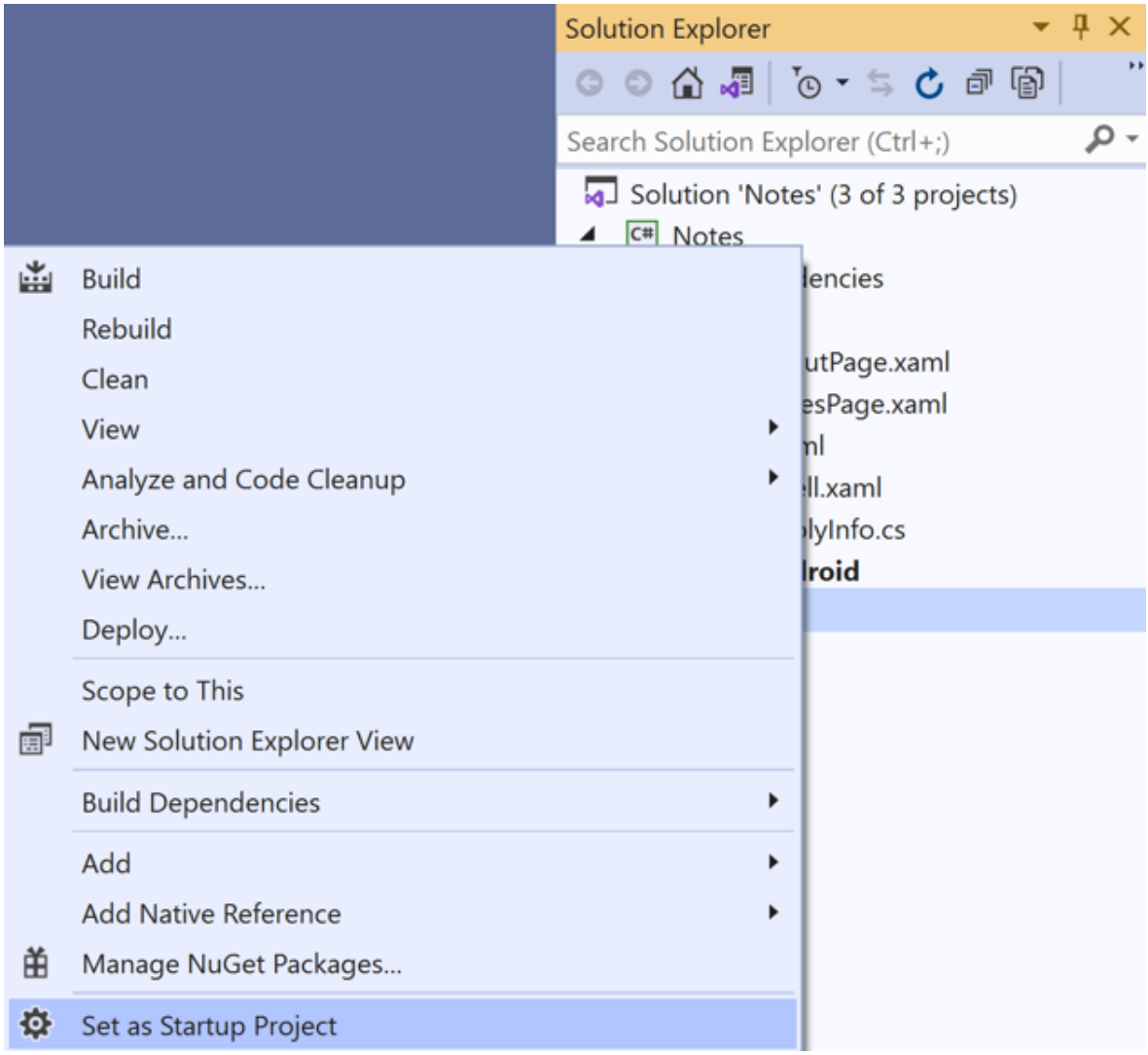
Press the **About** tab icon to navigate to the `AboutPage`:

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

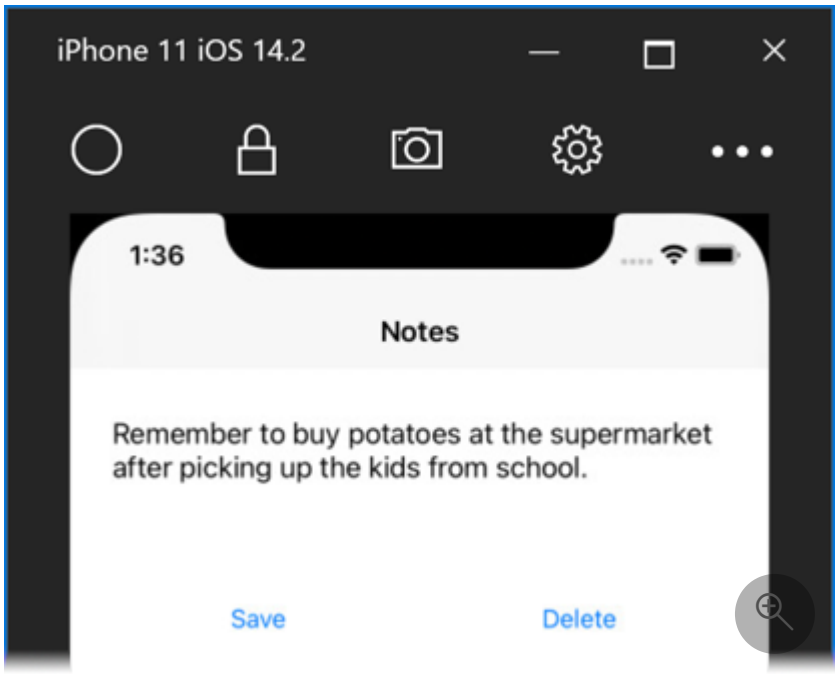
For more information about how the application is launched on each platform, see Launching the application on each platform in the Xamarin.Forms Quickstart Deep Dive.

> ⓘ **Note**
>
> The following steps should only be carried out if you have a **paired Mac** that meets the system requirements for Xamarin.Forms development.

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



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



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

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

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

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

Next steps

In this quickstart, you learned how to:

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

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

Next

Related links

- Notes (sample)
- Xamarin.Forms Shell quickstart deep dive

Perform navigation in a Xamarin.Forms application

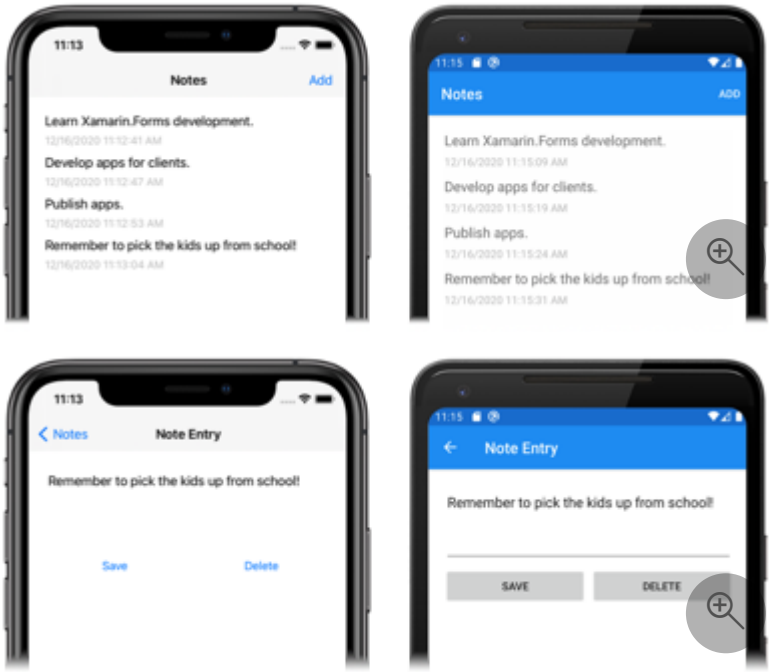
Article • 07/08/2021 • 17 minutes to read

Download the sample

In this quickstart, you will learn how to:

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

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

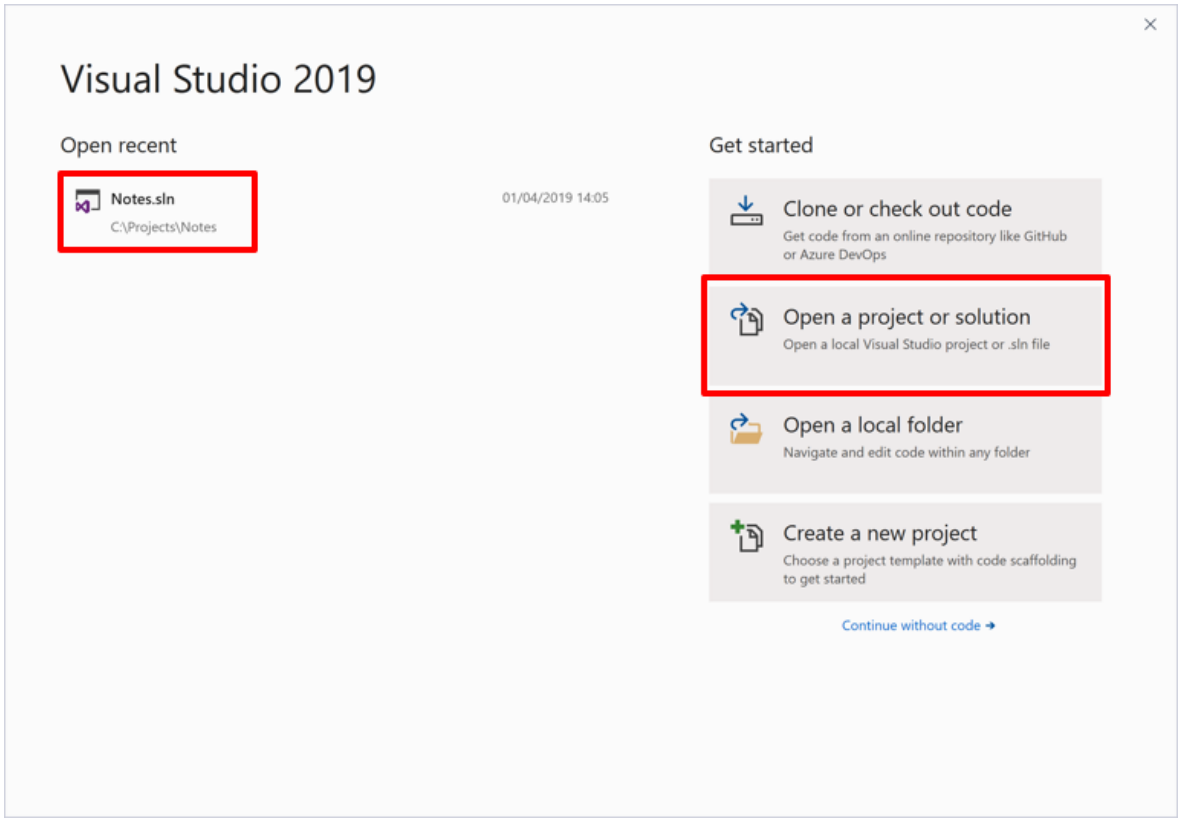


Prerequisites

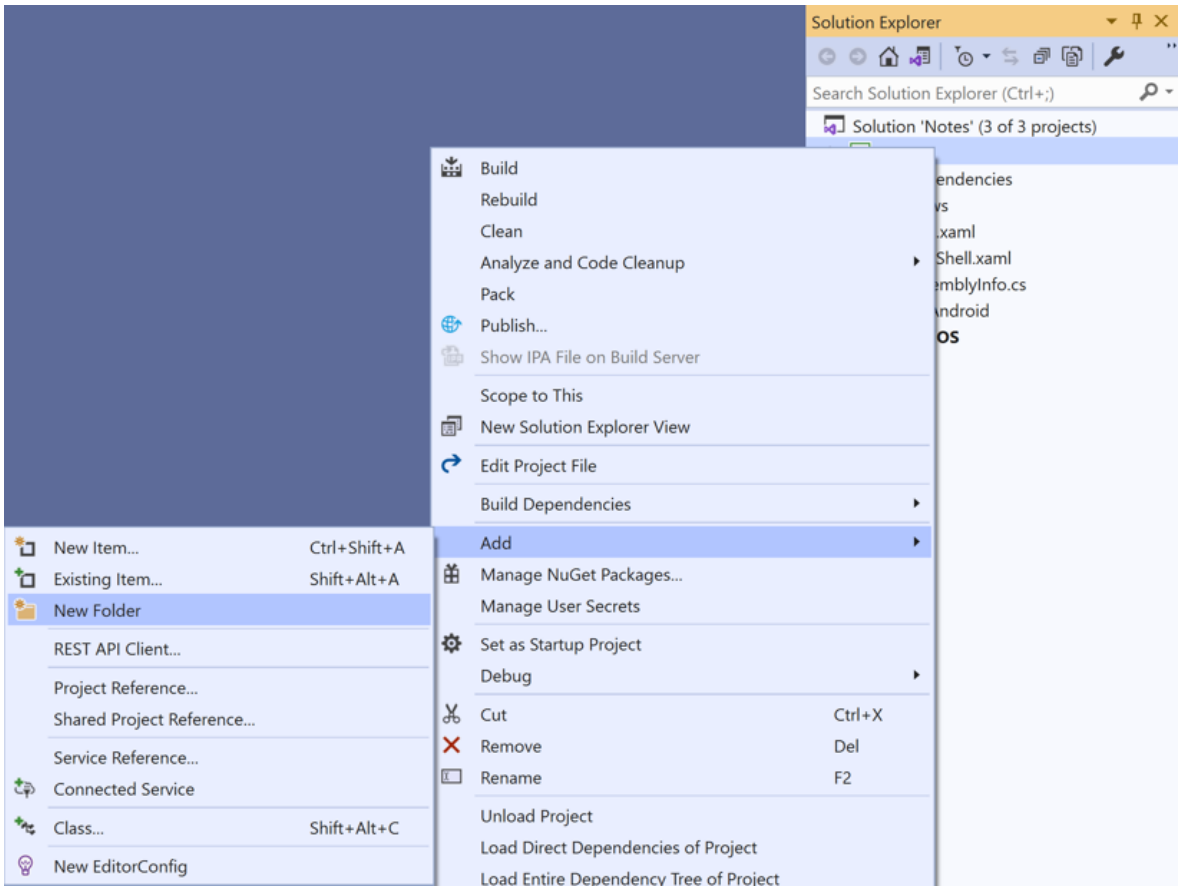
You should successfully complete the previous quickstart before attempting this quickstart. Alternatively, download the previous quickstart sample and use it as the starting point for this quickstart.

Update the app with Visual Studio

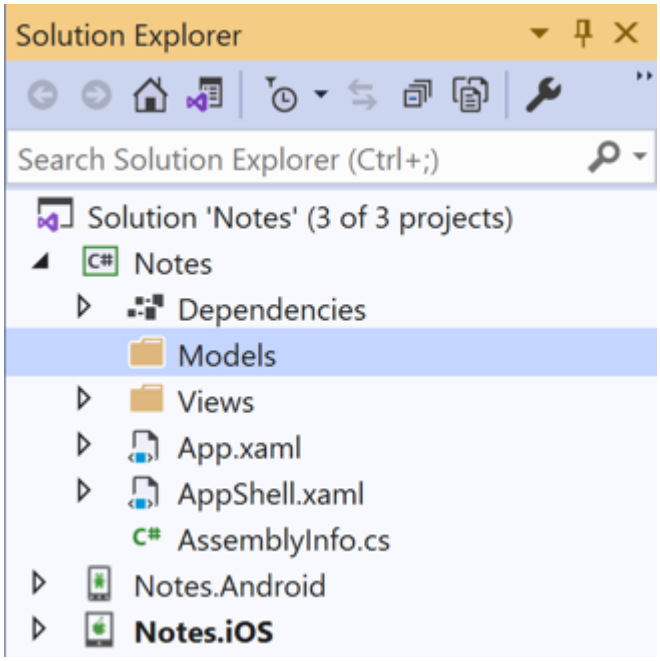
1. Launch Visual Studio. In the start window, click the **Notes** solution in the recent projects/solutions list, or click **Open a project or solution**, and in the **Open Project/Solution** dialog select the solution file for the Notes project:



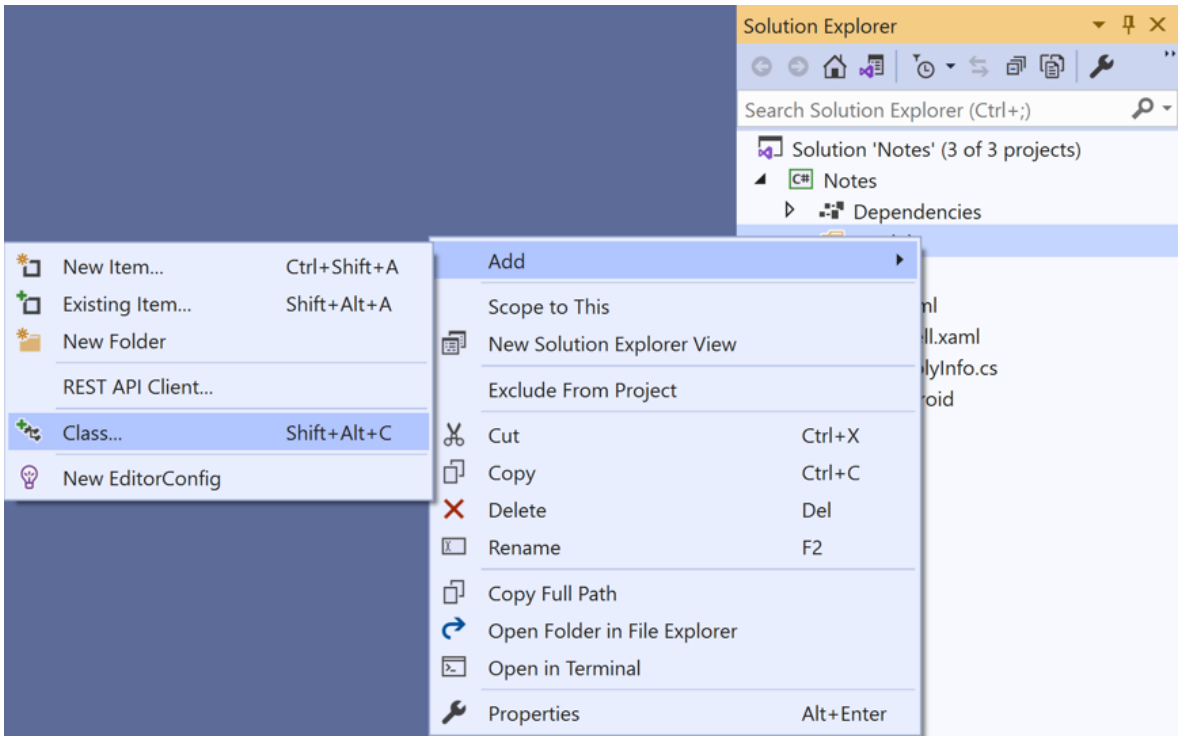
2. In **Solution Explorer**, right-click on the **Notes** project, and select **Add** > **New Folder**:



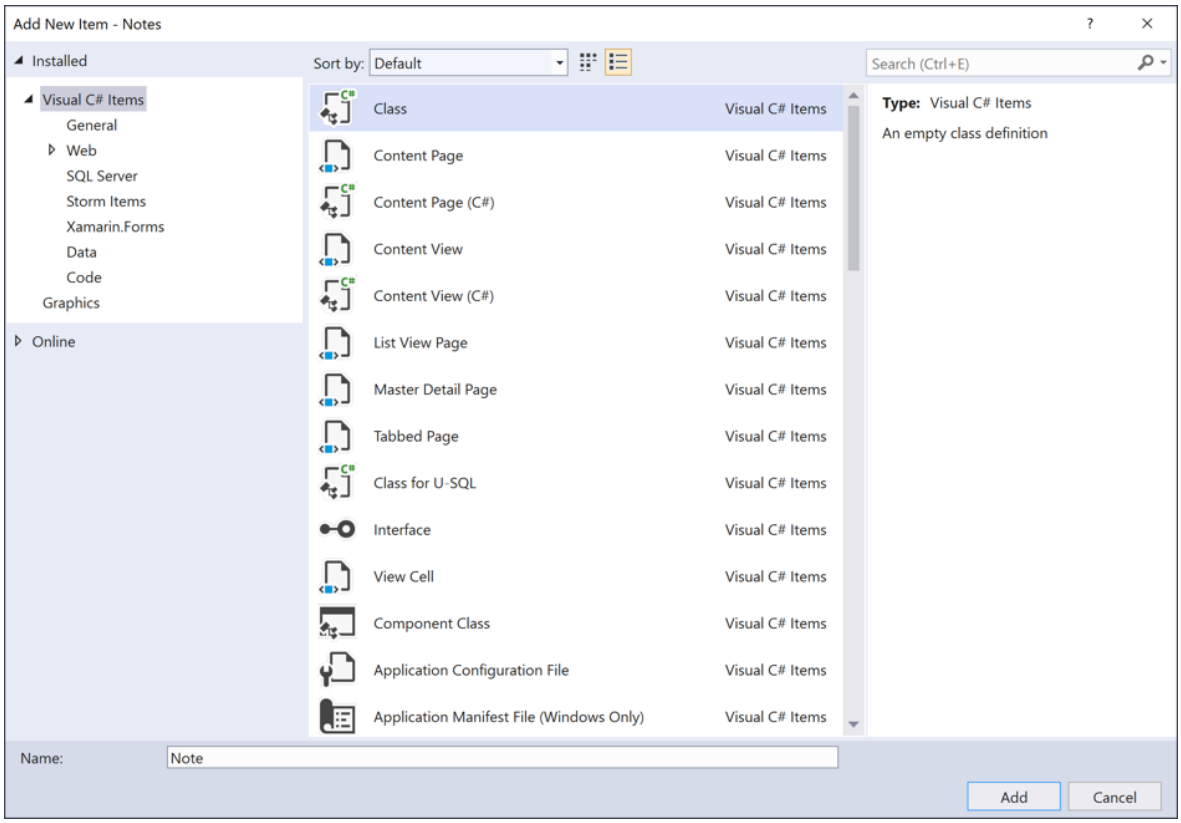
3. In **Solution Explorer**, name the new folder **Models**:



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



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



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

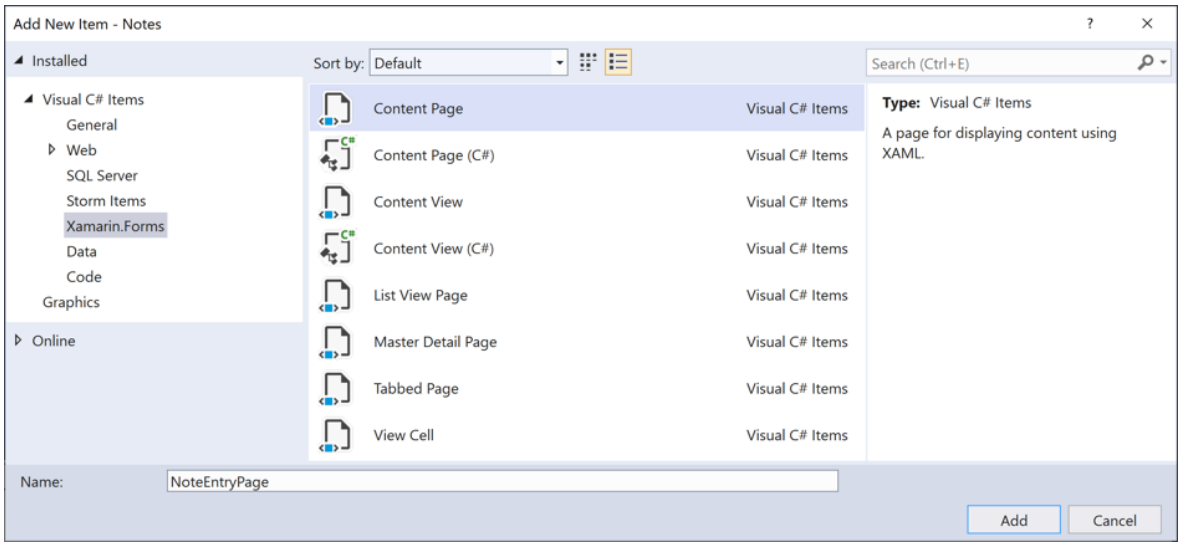
6. In **Note.cs**, remove all of the template code and replace it with the following code:

```
C#

using System;

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

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

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

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



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

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

XAML

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Notes.Views.NoteEntryPage"
             Title="Note Entry">
    <!-- Layout children vertically -->
    <StackLayout Margin="20">
        <Editor Placeholder="Enter your note"
                Text="{Binding Text}"
                HeightRequest="100" />
        <!-- Layout children in two columns -->
        <Grid ColumnDefinitions="*,*">
            <Button Text="Save"
                    Clicked="OnSaveButtonClicked" />
            <Button Grid.Column="1"
                    Text="Delete"
                    Clicked="OnDeleteButtonClicked"/>
        </Grid>
    </StackLayout>
</ContentPage>
```

This code declaratively defines the user interface for the page, which consists of an Editor for text input, and two Button objects that direct the application to save or delete a file. The two `Button` instances are horizontally laid out in a Grid, with the `Editor` and `Grid` being vertically laid out in a StackLayout. In addition, the `Editor` uses data binding to bind to the `Text` property of the `Note` model. For more

information about data binding, see Data binding in the Xamarin.Forms Quickstart Deep Dive.

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

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

```
C#
```

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

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

        public NoteEntryPage()
        {
            InitializeComponent();

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

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

```
            }
        }

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

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

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

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

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

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

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

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

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

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

10. In **Solution Explorer**, in the **Notes** project, open **NotesPage.xaml** in the **Views** folder.

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

XAML

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Notes.Views.NotesPage"
             Title="Notes">
    <!-- Add an item to the toolbar -->
    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Add"
                     Clicked="OnAddClicked" />
    </ContentPage.ToolbarItems>

    <!-- Display notes in a list -->
    <CollectionView x:Name="collectionView"
                    Margin="20"
                    SelectionMode="Single"
                    SelectionChanged="OnSelectionChanged">
        <CollectionView.ItemsLayout>
            <LinearItemsLayout Orientation="Vertical"
                               ItemSpacing="10" />
        </CollectionView.ItemsLayout>
        <!-- Define the appearance of each item in the list -->
        <CollectionView.ItemTemplate>
```

```
            <DataTemplate>
                <StackLayout>
                    <Label Text="{Binding Text}"
                           FontSize="Medium"/>
                    <Label Text="{Binding Date}"
                           TextColor="Silver"
                           FontSize="Small" />
                </StackLayout>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</ContentPage>
```

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

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

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

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

C#

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

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

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

```
            var notes = new List<Note>();

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

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

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

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

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

`Filename` property of the selected `Note` as a query parameter to the page. For more information about navigation, see Navigation in the Xamarin.Forms Quickstart Deep Dive.

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

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

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

```
C#
```

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

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

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

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

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

```
C#
```

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

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

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

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

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

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

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

16. Build and run the project on each platform. For more information, see Building the quickstart.

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

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

Next steps

In this quickstart, you learned how to:

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

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

Next

Related links

- Notes (sample)
- Xamarin.Forms Shell quickstart deep dive

Store data in a local SQLite.NET database

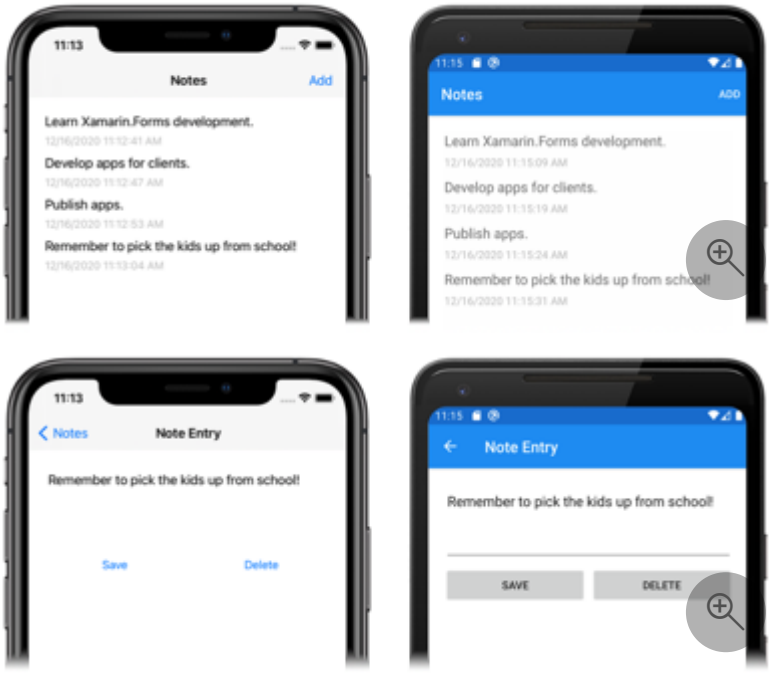
Article • 12/21/2022 • 11 minutes to read

Download the sample

In this quickstart, you will learn how to:

- Store data locally in a SQLite.NET database.

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

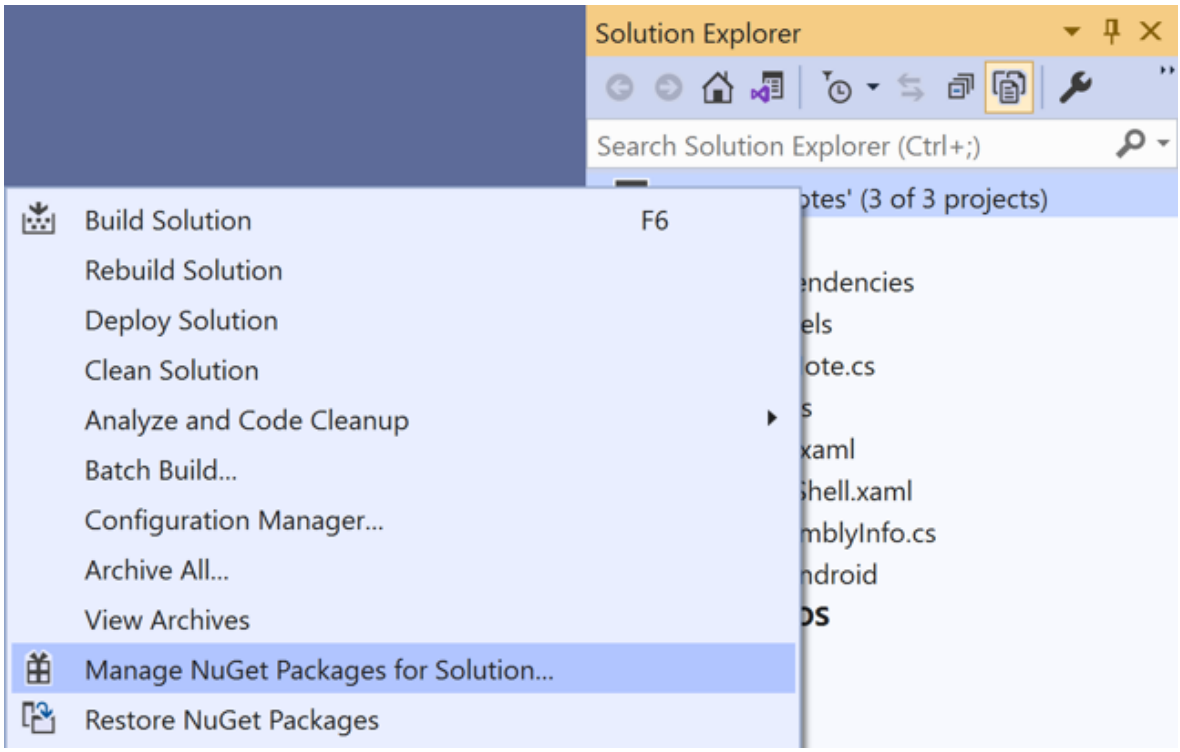


Prerequisites

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

Update the app with Visual Studio

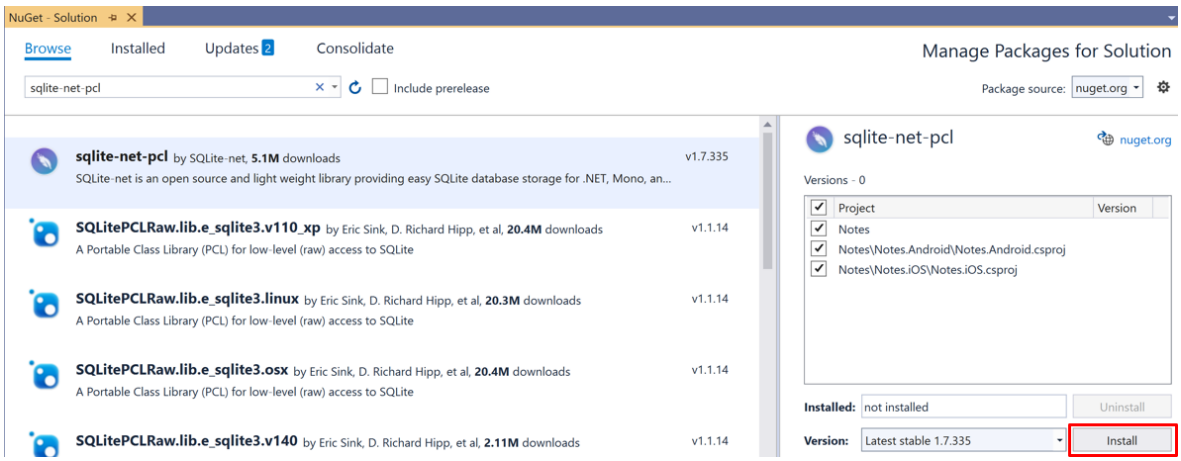
1. Launch Visual Studio and open the Notes solution.
2. In **Solution Explorer**, right-click the **Notes** solution and select **Manage NuGet Packages for Solution...**:



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

> ⚠ **Warning**
>
> There are many NuGet packages with similar names. The correct package has these attributes:
>
> - **Authors:** SQLite-net
> - **NuGet link:** sqlite-net-pcl
>
> Despite the package name, this NuGet package can be used in .NET Standard projects.

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



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

> **ⓘ Important**
>
> SQLite.NET is a third-party library that's supported from the **praeclarum/sqlite-net repo**.

Close the **NuGet Package Manager**.

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

```
C#

using System;
using SQLite;

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

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

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

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

5. In **Solution Explorer**, add a new folder named **Data** to the **Notes** project.

6. In **Solution Explorer**, in the **Notes** project, add a new class named **NoteDatabase** to the **Data** folder.

7. In **NoteDatabase.cs**, replace the existing code with the following code:

C#

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

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

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

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

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

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

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

```
    }
}
```

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

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

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

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

```
C#

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

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

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

        public App()
```

```
        {
            InitializeComponent();
            MainPage = new AppShell();
        }

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

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

This code defines a `Database` property that creates a new `NoteDatabase` instance as a singleton, passing in the filename of the database as the argument to the `NoteDatabase` constructor. The advantage of exposing the database as a singleton is that a single database connection is created that's kept open while the application runs, therefore avoiding the expense of opening and closing the database file each time a database operation is performed.

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

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

9. In **Solution Explorer**, in the **Notes** project, expand **NotesPage.xaml** in the **Views** folder and open **NotesPage.xaml.cs**. Then replace the `OnAppearing` and `OnSelectionChanged` methods with the following code:

C#

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

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

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

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

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

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

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

C#

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

async void OnSaveButtonClicked(object sender, EventArgs e)
{
```

```
    var note = (Note)BindingContext;
    note.Date = DateTime.UtcNow;
    if (!string.IsNullOrWhiteSpace(note.Text))
    {
        await App.Database.SaveNoteAsync(note);
    }

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

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

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

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

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

11. Build and run the project on each platform. For more information, see Building the quickstart.

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

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

Next steps

In this quickstart, you learned how to:

- Store data locally in a SQLite.NET database.

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

Next

Related links

- Notes (sample)
- Xamarin.Forms Shell Quickstart Deep Dive

Style a cross-platform Xamarin.Forms application

Article • 07/08/2021 • 10 minutes to read

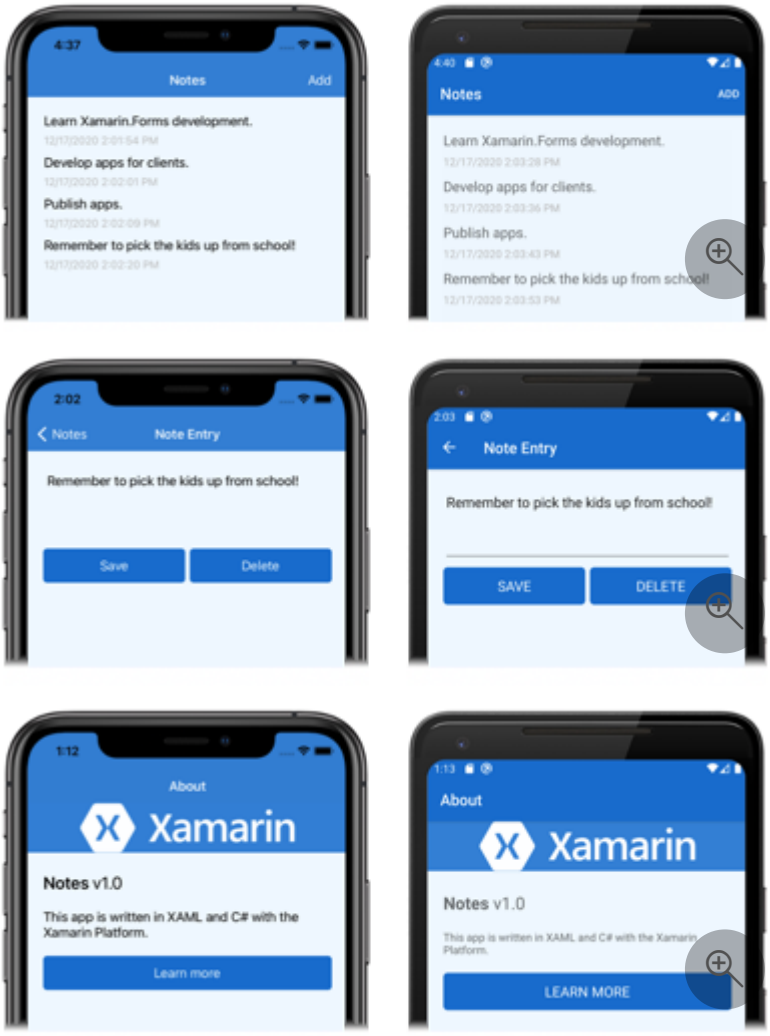
Download the sample

In this quickstart, you will learn how to:

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

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

The final application is shown below:



Prerequisites

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

Update the app with Visual Studio

1. Launch Visual Studio and open the Notes solution.

2. Build and run the project on your chosen platform. For more information, see Building the quickstart.

   Leave the application running and return to Visual Studio.

3. In **Solution Explorer**, in the **Notes** project, open **App.xaml**. Then replace the existing code with the following code:

XAML

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

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

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

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

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

        <Style TargetType="Button">
            <Setter Property="FontSize"
                    Value="Medium" />
            <Setter Property="BackgroundColor"
                    Value="{StaticResource AppPrimaryColor}" />
            <Setter Property="TextColor"
```

```
                    Value="{StaticResource SecondaryColor}" />
            <Setter Property="CornerRadius"
                    Value="5" />
        </Style>

    </Application.Resources>
</Application>
```

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

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

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

XAML

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

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

    <!-- Display a bottom tab bar containing two tabs -->
    <TabBar>
```

```
        <ShellContent Title="Notes"
                      Icon="icon_feed.png"
                      ContentTemplate="{DataTemplate views:NotesPage}"
/>
        <ShellContent Title="About"
                      Icon="icon_about.png"
                      ContentTemplate="{DataTemplate views:AboutPage}"
/>
    </TabBar>
</Shell>
```

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

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

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

XAML

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

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

    <!-- Add an item to the toolbar -->
```

```
    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Add"
                     Clicked="OnAddClicked" />
    </ContentPage.ToolbarItems>

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

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

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

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

XAML

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Notes.Views.NoteEntryPage"
             Title="Note Entry">
```

```
    <ContentPage.Resources>
        <!-- Implicit styles -->
        <Style TargetType="{x:Type Editor}">
            <Setter Property="BackgroundColor"
                    Value="{StaticResource AppBackgroundColor}" />
        </Style>
    </ContentPage.Resources>

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

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

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

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

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

   XAML

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Notes.Views.AboutPage"
             Title="About">
    <!-- Layout children in two rows -->
    <Grid RowDefinitions="Auto,*">
        <Image Source="xamarin_logo.png"
               BackgroundColor="{StaticResource AppPrimaryColor}"
               Opacity="0.85"
```

```
                VerticalOptions="Center"
                HeightRequest="64" />
        <!-- Layout children vertically -->
        <StackLayout Grid.Row="1"
                     Margin="{StaticResource PageMargin}"
                     Spacing="20">
            <Label FontSize="22">
                <Label.FormattedText>
                    <FormattedString>
                        <FormattedString.Spans>
                            <Span Text="Notes"
                                  FontAttributes="Bold"
                                  FontSize="22" />
                            <Span Text=" v1.0" />
                        </FormattedString.Spans>
                    </FormattedString>
                </Label.FormattedText>
            </Label>
            <Label Text="This app is written in XAML and C# with the
Xamarin Platform." />
            <Button Text="Learn more"
                    Clicked="OnButtonClicked" />
        </StackLayout>
    </Grid>
</ContentPage>
```

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

9. In the running application, navigate to the `AboutPage`.

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

Next steps

In this quickstart, you learned how to:

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

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

Next

Related links

- Notes (sample)
- XAML Hot Reload for Xamarin.Forms
- Xamarin.Forms Quickstart Deep Dive

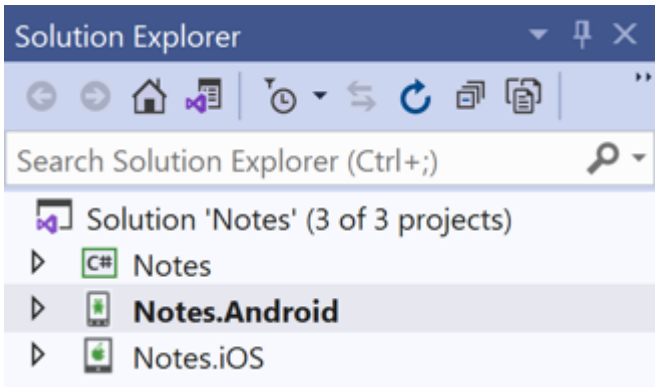
Xamarin.Forms Quickstart Deep Dive

Article • 09/20/2022 • 18 minutes to read

In the Xamarin.Forms Quickstart, the Notes application was built. This article reviews what has been built to gain an understanding of the fundamentals of how Xamarin.Forms Shell applications work.

Introduction to Visual Studio

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

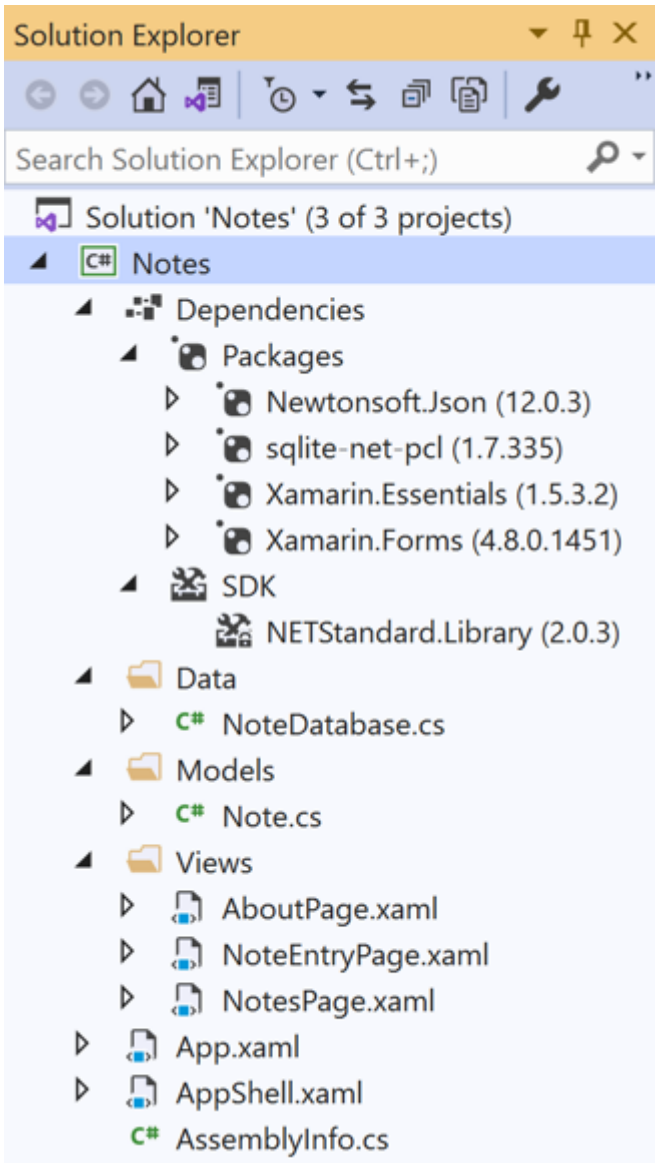


The projects are:

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

Anatomy of a Xamarin.Forms application

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



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

- **NuGet** – the Xamarin.Forms, Xamarin.Essentials, Newtonsoft.Json, and sqlite-net-pcl NuGet packages that have been added to the project.
- **SDK** – the `NETStandard.Library` metapackage that references the complete set of NuGet packages that define .NET Standard.

The project also consists of multiple files:

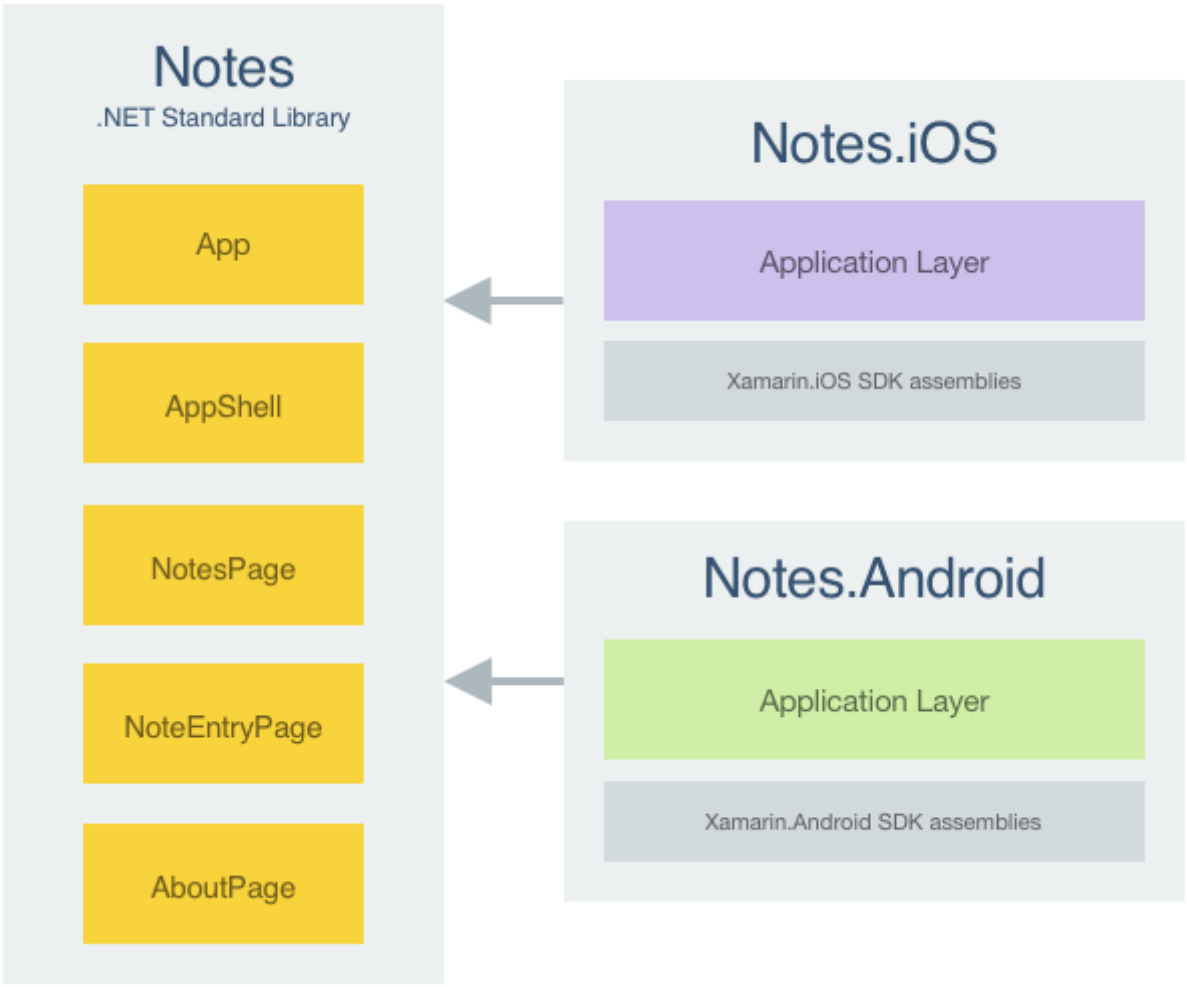
- **Data\NoteDatabase.cs** – This class contains code to create the database, read data from it, write data to it, and delete data from it.
- **Models\Note.cs** – This class defines a `Note` model whose instances store data about each note in the application.
- **Views\AboutPage.xaml** – The XAML markup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

- **Views\NotesPage.xaml** – The XAML markup for the `NotesPage` class, which defines the UI for the page shown when the application launches.
- **Views\NotesPage.xaml.cs** – The code-behind for the `NotesPage` class, which contains the business logic that is executed when the user interacts with the page.
- **Views\NoteEntryPage.xaml** – The XAML markup for the `NoteEntryPage` class, which defines the UI for the page shown when the user enters a note.
- **Views\NoteEntryPage.xaml.cs** – The code-behind for the `NoteEntryPage` class, which contains the business logic that is executed when the user interacts with the page.
- **App.xaml** – The XAML markup for the `App` class, which defines a resource dictionary for the application.
- **App.xaml.cs** – The code-behind for the `App` class, which is responsible for instantiating the Shell application, and for handling application lifecycle events.
- **AppShell.xaml** – The XAML markup for the `AppShell` class, which defines the visual hierarchy of the application.
- **AppShell.xaml.cs** – The code-behind for the `AppShell` class, which creates a route for the `NoteEntryPage` so that it can be navigated to programmatically.
- **AssemblyInfo.cs** – This file contains an application attribute about the project, that is applied at the assembly level.

For more information about the anatomy of a Xamarin.iOS application, see Anatomy of a Xamarin.iOS Application. For more information about the anatomy of a Xamarin.Android application, see Anatomy of a Xamarin.Android Application.

Architecture and application fundamentals

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



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

C#

```
using Xamarin.Forms;

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

This code sets the `MainPage` property of the `App` class to the `AppShell` object. The `AppShell` class defines the visual hierarchy of the application. Shell takes this visual

hierarchy and produces the user interface for it. For more information about defining the visual hierarchy of the application, see Application visual hierarchy.

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

```
C#

using Xamarin.Forms.Xaml;

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

The XamlCompilation attribute turns on the XAML compiler, so that XAML is compiled directly into intermediate language. For more information, see XAML Compilation.

Launch the application on each platform

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

iOS

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

```
C#

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

The `FinishedLaunching` override initializes the Xamarin.Forms framework by calling the `Init` method. This causes the iOS-specific implementation of Xamarin.Forms to be

loaded in the application before the root view controller is set by the call to the `LoadApplication` method.

Android

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

C#

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

            base.OnCreate(savedInstanceState);
            global::Xamarin.Forms.Forms.Init(this, savedInstanceState);
            LoadApplication(new App());
        }
    }
}
```

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

Application visual hierarchy

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

XAML

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

```
        xmlns:views="clr-namespace:Notes.Views"
        x:Class="Notes.AppShell">
    <TabBar>
        <ShellContent Title="Notes"
                      Icon="icon_feed.png"
                      ContentTemplate="{DataTemplate views:NotesPage}" />
        <ShellContent Title="About"
                      Icon="icon_about.png"
                      ContentTemplate="{DataTemplate views:AboutPage}" />
    </TabBar>
</Shell>
```

This XAML consists of two main objects:

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

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

For more information about Shell applications, see Xamarin.Forms Shell.

User interface

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

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

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

Layout

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

The following XAML code shows an example of using a StackLayout to layout the `NoteEntryPage`:

XAML

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Notes.Views.NoteEntryPage"
             Title="Note Entry">
    ...
    <StackLayout Margin="{StaticResource PageMargin}">
        <Editor Placeholder="Enter your note"
                Text="{Binding Text}"
                HeightRequest="100" />
        <Grid>
            ...
        </Grid>
    </StackLayout>
</ContentPage>
```

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

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

For more information about the StackLayout class, see Xamarin.Forms StackLayout.

Responding to user interaction

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

C#

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

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

> **Note**
>
> The code-behind file for a XAML class can access an object defined in XAML using the name assigned to it with the `x:Name` attribute. The value assigned to this attribute has the same rules as C# variables, in that it must begin with a letter or underscore and contain no embedded spaces.

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

XAML

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

Lists

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

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

XAML

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

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

C#

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

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

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

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

C#

```
async void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
{
```

```
    if (e.CurrentSelection != null)
    {
        // ...
    }
}
```

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

For more information about the CollectionView class, see Xamarin.Forms CollectionView.

Navigation

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

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

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

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

For more information about Shell navigation, see Xamarin.Forms Shell navigation.

Register routes

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

C#

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

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

Perform navigation

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

C#

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

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

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

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

C#

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

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

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

C#

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

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

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

C#

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

For more information about backwards navigation, see Backwards navigation.

Data binding

Data binding is used to simplify how a Xamarin.Forms application displays and interacts with its data. It establishes a connection between the user interface and the underlying

application. The BindableObject class contains much of the infrastructure to support data binding.

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

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

Establishing data binding is a two-step process:

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

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

C#

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

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

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

C#

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

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

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

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

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

XAML

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

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

For more information about data binding, see Xamarin.Forms Data Binding.

Styling

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

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

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

> **ⓘ Important**
>
> Any styles that are used throughout the application are stored in the application's resource dictionary to avoid duplication. However, XAML that's specific to a page shouldn't be included in the application's resource dictionary, as the resources will then be parsed at application startup instead of when required by a page. For more information, see **Reduce the application resource dictionary size**.

Each Style instance contains a collection of one or more Setter objects, with each `Setter` having a Property and a Value. The `Property` is the name of the bindable property of the element the style is applied to, and the `Value` is the value that is applied to the property. The following code example shows a style from `NoteEntryPage`:

XAML

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

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

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

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

For more information about XAML styles, see Styling Xamarin.Forms Apps using XAML Styles.

Test and deployment

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

Simulators are a good place to start deploying and testing an application, and feature useful functionality for testing applications. However, users will not consume the final application in a simulator, so applications should be tested on real devices early and often. For more information about iOS device provisioning, see Device Provisioning. For more information about Android device provisioning, see Set Up Device for Development.

Next steps

This deep dive has examined the fundamentals of application development using Xamarin.Forms Shell. Suggested next steps include reading about the following functionality:

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

Related links

- Xamarin.Forms Shell
- eXtensible Application Markup Language (XAML)
- Data Binding
- Controls Reference
- Get Started Samples
- Xamarin.Forms Samples

- Xamarin.Forms API reference

Related video

https://learn.microsoft.com/shows/Xamarin-101/Xamarin-Solution-Architecture-4-of-11

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Forms tutorials

Learn about the basics of creating mobile applications with Xamarin.Forms.

Layouts

HOW-TO GUIDE

StackLayout

Grid

Controls

HOW-TO GUIDE

Label

Button

Entry

Editor

Image

CollectionView

Pop-ups

App fundamentals

HOW-TO GUIDE

App lifecycle

Local database

Web services

Cross-Platform for Desktop Developers

Article • 07/12/2021 • 2 minutes to read

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

App Lifecycle Comparison

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

UI Controls Comparison

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

Porting Guidance

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

Samples

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

Learn More

Xamarin for Java developers

Article • 01/06/2023 • 24 minutes to read

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

Overview

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

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

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

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

Both Java and C# are compiled to an intermediate language that is run in a managed execution environment. Both C# and Java are statically-typed, and both languages treat strings as immutable types. Both languages use a single-rooted class hierarchy. Like Java, C# supports only single inheritance and does not allow for global methods. In both

languages, objects are created on the heap using the `new` keyword, and objects are garbage-collected when they are no longer used. Both languages provide formal exception handling support with `try`/`catch` semantics. Both provide thread management and synchronization support.

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

- In Java, you can pass parameters only by value, while in C# you can pass by reference as well as by value. (C# provides the `ref` and `out` keywords for passing parameters by reference; there is no equivalent to these in Java).
- Java does not support preprocessor directives like `#define`.
- Java does not support unsigned integer types, while C# provides unsigned integer types such as `ulong`, `uint`, `ushort` and `byte`.
- Java does not support operator overloading; in C# you can overload operators and conversions.
- In a Java `switch` statement, code can fall through into the next switch section, but in C# the end of every `switch` section must terminate the switch (the end of each section must close with a `break` statement).
- In Java, you specify the exceptions thrown by a method with the `throws` keyword, but C# has no concept of checked exceptions – the `throws` keyword is not supported in C#.
- C# supports Language-Integrated Query (LINQ), which lets you use the reserved words `from`, `select`, and `where` to write queries against collections in a way that is similar to database queries.

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

- Going from Java to C# Development provides an introduction to the fundamental differences between C# and Java.
- Object-Oriented Programming Features outlines the most important object-oriented feature differences between the two languages.

- Keyword Differences provides a table of useful keyword equivalents, C#-only keywords, and links to C# keyword definitions.

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

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

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

> **Note**
>
> Android programming uses a specific version of the Java language that supports all Java 7 features **and a subset of Java 8**.
>
> Some features mentioned on this page (such as the `var` keyword in C#) are available in newer versions of Java (e.g. **var in Java 10**), but are still not available to Android developers.

Going from Java to C# development

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

Libraries vs. assemblies

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

For more information about assemblies, see the Assemblies and the Global Assembly Cache topic.

Packages vs. namespaces

C# uses the `namespace` keyword to group related types together; this is similar to Java's `package` keyword. Typically, a Xamarin.Android app will reside in a namespace created for that app. For example, the following C# code declares the `WeatherApp` namespace wrapper for a weather-reporting app:

C#

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

Importing types

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

Java

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

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

Java

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

The C# `using` statement works in a very similar way, but it allows you to import an entire package without specifying a wildcard. For example, you will often see a series of

`using` statements at the beginning of Xamarin.Android source files, as seen in this example:

```
C#

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

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

Generics

Both Java and C# support *generics*, which are placeholders that let you plug in different types at compile time. However, generics work slightly differently in C#. In Java, type erasure makes type information available only at compile time, but not at run time. By contrast, the .NET common language runtime (CLR) provides explicit support for generic types, which means that C# has access to type information at runtime. In day-to-day Xamarin.Android development, the importance of this distinction is not often apparent, but if you are using reflection, you will depend on this feature to access type information at run time.

In Xamarin.Android, you will often see the generic method `FindViewById` used to get a reference to a layout control. This method accepts a generic type parameter that specifies the type of control to look up. For example:

```
C#

TextView label = FindViewById<TextView> (Resource.Id.Label);
```

In this code example, `FindViewById` gets a reference to the `TextView` control that is defined in the layout as **Label**, then returns it as a `TextView` type.

For more information about generics, see the Generics topic. Note that there are some limitations in Xamarin.Android support for generic C# classes; for more information, see

Limitations.

Object-oriented programming features

Both Java and C# use very similar object-oriented programming idioms:

- All classes are ultimately derived from a single root object – all Java objects derive from `java.lang.Object`, while all C# objects derive from `System.Object`.
- Instances of classes are reference types.
- When you access the properties and methods of an instance, you use the "`.`" operator.
- All class instances are created on the heap via the `new` operator.
- Because both languages use garbage collection, there is no way to explicitly release unused objects (i.e., there is not a `delete` keyword as there is in C++).
- You can extend classes through inheritance, and both languages only allow a single base class per type.
- You can define interfaces, and a class can inherit from (i.e., implement) multiple interface definitions.

However, there are also some important differences:

- Java has two powerful features that C# does not support: anonymous classes and inner classes. (However, C# does allow nesting of class definitions – C#'s nested classes are like Java's static nested classes.)
- C# supports C-style structure types (`struct`) while Java does not.
- In C#, you can implement a class definition in separate source files by using the `partial` keyword.
- C# interfaces cannot declare fields.
- C# uses C++-style destructor syntax to express finalizers. The syntax is different from Java's `finalize` method, but the semantics are nearly the same. (Note that in C#, destructors automatically call the base-class destructor – in contrast to Java where an explicit call to `super.finalize` is used.)

Class inheritance

To extend a class in Java, you use the `extends` keyword. To extend a class in C#, you use a colon (`:`) to indicate derivation. For example, in Xamarin.Android apps, you will often see class derivations that resemble the following code fragment:

```
C#

public class MainActivity : Activity
{
    ...
```

In this example, `MainActivity` inherits from the `Activity` class.

To declare support for an interface in Java, you use the `implements` keyword. However, in C#, you simply add interface names to the list of classes to inherit from, as shown in this code fragment:

```
C#

public class SensorsActivity : Activity, ISensorEventListener
{
    ...
```

In this example, `SensorsActivity` inherits from `Activity` and implements the functionality declared in the `ISensorEventListener` interface. Note that the list of interfaces must come after the base class (or you will get a compile-time error). By convention, C# interface names are prepended with an upper-case "I"; this makes it possible to determine which classes are interfaces without requiring an `implements` keyword.

When you want to prevent a class from being further subclassed in C#, you precede the class name with `sealed` – in Java, you precede the class name with `final`.

For more about C# class definitions, see the Classes and Inheritance topics.

Properties

In Java, mutator methods (setters) and inspector methods (getters) are often used to control how changes are made to class members while hiding and protecting these members from outside code. For example, the Android `TextView` class provides `getText` and `setText` methods. C# provides a similar but more direct mechanism known as *properties*. Users of a C# class can access a property in the same way as they would access a field, but each access actually results in a method call that is transparent to the

caller. This "under the covers" method can implement side effects such as setting other values, performing conversions, or changing object state.

Properties are often used for accessing and modifying UI (user interface) object members. For example:

```
C#

int width = rulerView.MeasuredWidth;
int height = rulerView.MeasuredHeight;
...
rulerView.DrawingCacheEnabled = true;
```

In this example, width and height values are read from the `rulerView` object by accessing its `MeasuredWidth` and `MeasuredHeight` properties. When these properties are read, values from their associated (but hidden) field values are fetched behind the scenes and returned to the caller. The `rulerView` object may store width and height values in one unit of measurement (say, pixels) and convert these values on-the-fly to a different unit of measurement (say, millimeters) when the `MeasuredWidth` and `MeasuredHeight` properties are accessed.

The `rulerView` object also has a property called `DrawingCacheEnabled` – the example code sets this property to `true` to enable the drawing cache in `rulerView`. Behind the scenes, an associated hidden field is updated with the new value, and possibly other aspects of `rulerView` state are modified. For example, when `DrawingCacheEnabled` is set to `false`, `rulerView` may also erase any drawing cache information already accumulated in the object.

Access to properties can be read/write, read-only, or write-only. Also, you can use different access modifiers for reading and writing. For example, you can define a property that has public read access but private write access.

For more information about C# properties, see the Properties topic.

Calling base class methods

To call a base-class constructor in C#, you use a colon (`:`) followed by the `base` keyword and an initializer list; this `base` constructor call is placed immediately after the derived constructor parameter list. The base-class constructor is called on entry to the derived constructor; the compiler inserts the call to the base constructor at the start of the method body. The following code fragment illustrates a base constructor called from a derived constructor in a Xamarin.Android app:

C#

```
public class PictureLayout : ViewGroup
{
    ...
    public PictureLayout (Context context)
           : base (context)
    {
        ...
    }
    ...
}
```

In this example, the `PictureLayout` class is derived from the `ViewGroup` class. The `PictureLayout` constructor shown in this example accepts a `context` argument and passes it to the `ViewGroup` constructor via the `base(context)` call.

To call a base-class method in C#, use the `base` keyword. For example, Xamarin.Android apps often make calls to base methods as shown here:

C#

```
public class MainActivity : Activity
{
    ...
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
```

In this case, the `OnCreate` method defined by the derived class (`MainActivity`) calls the `OnCreate` method of the base class (`Activity`).

Access modifiers

Java and C# both support the `public`, `private`, and `protected` access modifiers. However, C# supports two additional access modifiers:

- **`internal`** – The class member is accessible only within the current assembly.
- **`protected internal`** – The class member is accessible within the defining assembly, the defining class, and derived classes (derived classes both inside and outside the assembly have access).

For more information about C# access modifiers, see the Access Modifiers topic.

Virtual and override methods

Both Java and C# support *polymorphism*, the ability to treat related objects in the same manner. In both languages, you can use a base-class reference to refer to a derived-class object, and the methods of a derived class can override the methods of its base classes. Both languages have the concept of a *virtual* method, a method in a base class that is designed to be replaced by a method in a derived class. Like Java, C# supports `abstract` classes and methods.

However, there are some differences between Java and C# in how you declare virtual methods and override them:

- In C#, methods are non-virtual by default. Parent classes must explicitly label which methods are to be overridden by using the `virtual` keyword. By contrast, all methods in Java are virtual methods by default.
- To prevent a method from being overridden in C#, you simply leave off the `virtual` keyword. By contrast, Java uses the `final` keyword to mark a method with "override is not allowed."
- C# derived classes must use the `override` keyword to explicitly indicate that a virtual base-class method is being overridden.

For more information about C#'s support for polymorphism, see the Polymorphism topic.

Lambda expressions

C# makes it possible to create *closures*: inline, anonymous methods that can access the state of the method in which they are enclosed. Using lambda expressions, you can write fewer lines of code to implement the same functionality that you might have implemented in Java with many more lines of code.

Lambda expressions make it possible for you to skip the extra ceremony involved in creating a one-time-use class or anonymous class as you would in Java – instead, you can just write the business logic of your method code inline. Also, because lambdas have access to the variables in the surrounding method, you don't have to create a long parameter list to pass state to your method code.

In C#, lambda expressions are created with the `=>` operator as shown here:

```
C#
```

```
(arg1, arg2, ...) => {
    // implementation code
};
```

In Xamarin.Android, lambda expressions are often used for defining event handlers. For example:

```
C#

button.Click += (sender, args) => {
    clickCount += 1;    // access variable in surrounding code
    button.Text = string.Format ("Clicked {0} times.", clickCount);
};
```

In this example, the lambda expression code (the code within the curly braces) increments a click count and updates the `button` text to display the click count. This lambda expression is registered with the `button` object as a click event handler to be called whenever the button is tapped. (Event handlers are explained in more detail below.) In this simple example, the `sender` and `args` parameters are not used by the lambda expression code, but they are required in the lambda expression to meet the method signature requirements for event registration. Under the hood, the C# compiler translates the lambda expression into an anonymous method that is called whenever button click events take place.

For more information about C# and lambda expressions, see the Lambda Expressions topic.

Event handling

An *event* is a way for an object to notify registered subscribers when something interesting happens to that object. Unlike in Java, where a subscriber typically implements a `Listener` interface that contains a callback method, C# provides language-level support for event handling through *delegates*. A *delegate* is like an object-oriented type-safe function pointer – it encapsulates an object reference and a method token. If a client object wants to subscribe to an event, it creates a delegate and passes the delegate to the notifying object. When the event occurs, the notifying object invokes the method represented by the delegate object, notifying the subscribing client object of the event. In C#, event handlers are essentially nothing more than methods that are invoked through delegates.

For more information about delegates, see the Delegates topic.

In C#, events are *multicast*; that is, more than one listener can be notified when an event takes place. This difference is observed when you consider the syntactical differences between Java and C# event registration. In Java you call `SetXXXListener` to register for event notifications; in C# you use the `+=` operator to register for event notifications by "adding" your delegate to the list of event listeners. In Java, you call `SetXXXListener` to unregister, while in C# you use the `-=` to "subtract" your delegate from the list of listeners.

In Xamarin.Android, events are often used for notifying objects when a user does something to a UI control. Normally, a UI control will have members that are defined using the `event` keyword; you attach your delegates to these members to subscribe to events from that UI control.

To subscribe to an event:

1. Create a delegate object that refers to the method that you want to be invoked when the event occurs.
2. Use the `+=` operator to attach your delegate to the event you are subscribing to.

The following example defines a delegate (with an explicit use of the `delegate` keyword) to subscribe to button clicks. This button-click handler starts a new activity:

C#

```
startActivityButton.Click += delegate {
    Intent intent = new Intent (this, typeof (MyActivity));
    StartActivity (intent);
};
```

However, you also can use a lambda expression to register for events, skipping the `delegate` keyword altogether. For example:

C#

```
startActivityButton.Click += (sender, e) => {
    Intent intent = new Intent (this, typeof (MyActivity));
    StartActivity (intent);
};
```

In this example, the `startActivityButton` object has an event that expects a delegate with a certain method signature: one that accepts sender and event arguments and returns void. However, because we don't want to go to the trouble to explicitly define

such a delegate or its method, we declare the signature of the method with `(sender, e)` and use a lambda expression to implement the body of the event handler. Note that we have to declare this parameter list even though we aren't using the `sender` and `e` parameters.

It is important to remember that you can unsubscribe a delegate (via the `-=` operator), but you cannot unsubscribe a lambda expression – attempting to do so can cause memory leaks. Use the lambda form of event registration only when your handler will not unsubscribe from the event.

Typically, lambda expressions are used to declare event handlers in Xamarin.Android code. This shorthand way to declare event handlers may seem cryptic at first, but it saves an enormous amount of time when you are writing and reading code. With increasing familiarity, you become accustomed to recognizing this pattern (which occurs frequently in Xamarin.Android code), and you spend more time thinking about the business logic of your application and less time wading through syntactical overhead.

Asynchronous programming

Asynchronous programming is a way to improve the overall responsiveness of your application. Asynchronous programming features make it possible for the rest of your app code to continue running while some part of your app is blocked by a lengthy operation. Accessing the web, processing images, and reading/writing files are examples of operations that can cause an entire app to appear to freeze up if it is not written asynchronously.

C# includes language-level support for asynchronous programming via the `async` and `await` keywords. These language features make it very easy to write code that performs long-running tasks without blocking the main thread of your application. Briefly, you use the `async` keyword on a method to indicate that the code in the method is to run asynchronously and not block the caller's thread. You use the `await` keyword when you call methods that are marked with `async`. The compiler interprets the `await` as the point where your method execution is to be moved to a background thread (a task is returned to the caller). When this task completes, execution of the code resumes on the caller's thread at the `await` point in your code, returning the results of the `async` call. By convention, methods that run asynchronously have `Async` suffixed to their names.

In Xamarin.Android applications, `async` and `await` are typically used to free up the UI thread so that it can respond to user input (such as the tapping of a **Cancel** button) while a long-running operation takes place in a background task.

In the following example, a button click event handler causes an asynchronous operation to download an image from the web:

```
C#

downloadButton.Click += downloadAsync;
...
async void downloadAsync(object sender, System.EventArgs e)
{
    webClient = new WebClient ();
    var url = new Uri
("http://photojournal.jpl.nasa.gov/jpeg/PIA15416.jpg");
    byte[] bytes = null;

    bytes = await webClient.DownloadDataTaskAsync(url);

    // display the downloaded image ...
```

In this example, when the user clicks the `downloadButton` control, the `downloadAsync` event handler creates a `WebClient` object and a `Uri` object to fetch an image from the specifed URL. Next, it calls the `WebClient` object's `DownloadDataTaskAsync` method with this URL to retrieve the image.

Notice that the method declaration of `downloadAsync` is prefaced by the `async` keyword to indicate that it will run asynchronously and return a task. Also note that the call to `DownloadDataTaskAsync` is preceded by the `await` keyword. The app moves the execution of the event handler (starting at the point where `await` appears) to a background thread until `DownloadDataTaskAsync` completes and returns. Meanwhile, the app's UI thread can still respond to user input and fire event handlers for the other controls. When `DownloadDataTaskAsync` completes (which may take several seconds), execution resumes where the `bytes` variable is set to the result of the call to `DownloadDataTaskAsync`, and the remainder of the event handler code displays the downloaded image on the caller's (UI) thread.

For an introduction to `async`/`await` in C#, see the Asynchronous Programming with Async and Await topic. For more information about Xamarin support of asynchronous programming features, see Async Support Overview.

Keyword differences

Many language keywords used in Java are also used in C#. There are also a number of Java keywords that have an equivalent but differently-named counterpart in C#, as listed in this table:

Java	C#	Description
`boolean`	bool	Used for declaring the boolean values true and false.
`extends`	`:`	Precedes the class and interfaces to inherit from.
`implements`	`:`	Precedes the class and interfaces to inherit from.
`import`	using	Imports types from a namespace, also used for creating a namespace alias.
`final`	sealed	Prevents class derivation; prevents methods and properties from being overridden in derived classes.
`instanceof`	is	Evaluates whether an object is compatible with a given type.
`native`	extern	Declares a method that is implemented externally.
`package`	namespace	Declares a scope for a related set of objects.
`T...`	params T	Specifies a method parameter that takes a variable number of arguments.
`super`	base	Used to access members of the parent class from within a derived class.
`synchronized`	lock	Wraps a critical section of code with lock acquisition and release.

Also, there are many keywords that are unique to C# and have no counterpart in the Java used on Android. Xamarin.Android code often makes use of the following C# keywords (this table is useful to refer to when you are reading through Xamarin.Android sample code):

C#	Description
as	Performs conversions between compatible reference types or nullable types.
async	Specifies that a method or lambda expression is asynchronous.
await	Suspends the execution of a method until a task completes.
byte	Unsigned 8-bit integer type.
delegate	Used to encapsulate a method or anonymous method.
enum	Declares an enumeration, a set of named constants.
event	Declares an event in a publisher class.
fixed	Prevents a variable from being relocated.

C#	Description
`get`	Defines an accessor method that retrieves the value of a property.
in	Enables a parameter to accept a less derived type in a generic interface.
object	An alias for the Object type in the .NET framework.
out	Parameter modifier or generic type parameter declaration.
override	Extends or modifies the implementation of an inherited member.
partial	Declares a definition to be split into multiple files, or splits a method definition from its implementation.
readonly	Declares that a class member can be assigned only at declaration time or by the class constructor.
ref	Causes an argument to be passed by reference rather than by value.
set	Defines an accessor method that sets the value of a property.
string	Alias for the String type in the .NET framework.
struct	A value type that encapsulates a group of related variables.
typeof	Obtains the type of an object.
var	Declares an implicitly-typed local variable.
value	References the value that client code wants to assign to a property.
virtual	Allows a method to be overridden in a derived class.

Interoperating with existing java code

If you have existing Java functionality that you do not want to convert to C#, you can reuse your existing Java libraries in Xamarin.Android applications via two techniques:

- **Create a Java Bindings Library** – Using this approach, you use Xamarin tools to generate C# wrappers around Java types. These wrappers are called *bindings*. As a result, your Xamarin.Android application can use your *.jar* file by calling into these wrappers.

- **Java Native Interface** – The *Java Native Interface* (JNI) is a framework that makes it possible for C# apps to call or be called by Java code.

For more information about these techniques, see Java Integration Overview.

Further reading

The MSDN C# Programming Guide is a great way to get started in learning the C# programming language, and you can use the C# Reference to look up particular C# language features.

In the same way that Java knowledge is at least as much about familiarity with the Java class libraries as knowing the Java language, practical knowledge of C# requires some familiarity with the .NET framework. Microsoft's Moving to C# and the .NET Framework, for Java Developers learning packet is a good way to learn more about the .NET framework from a Java perspective (while gaining a deeper understanding of C#).

When you are ready to tackle your first Xamarin.Android project in C#, our Hello, Android series can help you build your first Xamarin.Android application and further advance your understanding of the fundamentals of Android application development with Xamarin.

Summary

This article provided an introduction to the Xamarin.Android C# programming environment from a Java developer's perspective. It pointed out the similarities between C# and Java while explaining their practical differences. It introduced assemblies and namespaces, explained how to import external types, and provided an overview of the differences in access modifiers, generics, class derivation, calling base-class methods, method overriding, and event handling. It introduced C# features that are not available in Java, such as properties, `async`/`await` asynchronous programming, lambdas, C# delegates, and the C# event handling system. It included tables of important C# keywords, explained how to interoperate with existing Java libraries, and provided links to related documentation for further study.

Related links

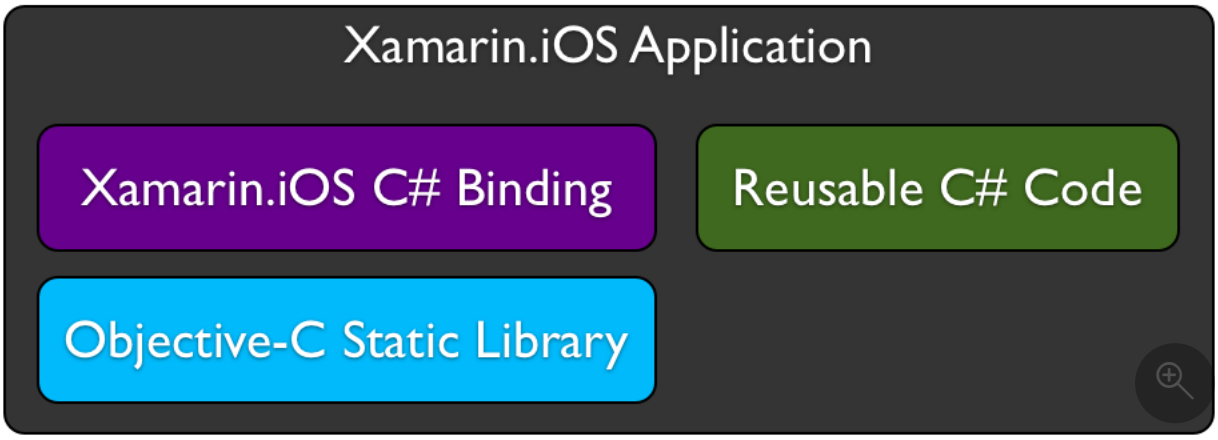
- Java Integration Overview
- C# Programming Guide
- C# Reference
- Moving to C# and the .NET Framework, for Java Developers

Xamarin for Objective-C Developers

Article • 07/12/2021 • 2 minutes to read

Xamarin offers a path for developers targeting iOS to move their non-user interface code to platform agnostic C# so that it can be used anywhere C# is available, including Android via Xamarin.Android and the various flavors of Windows. However, just because you use C# with Xamarin doesn't mean you can't leverage existing skills and Objective-C code. In fact, knowing Objective-C makes you a better Xamarin.iOS developer because Xamarin exposes all the native iOS and OS X platform APIs you know and love, such as UIKit, Core Animation, Core Foundation and Core Graphics to name a few. At the same time, you get the power of the C# language, including features like LINQ and Generics, as well as rich .NET base class libraries to use in your native applications.

Additionally, Xamarin allows you to leverage existing Objective-C assets via a technology know as bindings. You simply create a static library in Objective-C and expose it to C# via a binding, as illustrated in the following diagram:



This doesn't need to be limited to non-UI code. Bindings can expose user interface code developed in Objective-C as well.

Transitioning from Objective-C

You'll find a plethora of information on our documentation site to help ease the transition to Xamarin, showing how to integrate C# code with what you already know. Some highlights to get you started include:

- C# Primer for Objective-C Developers - A short primer for Objective-C developers looking to move to Xamarin and the C# language.
- Walkthrough: Binding an Objective-C Library - A step-by-step walkthrough for reusing existing Objective-C code in a Xamarin.iOS application.

Binding Objective-C

Once you have a grasp of how C# compares to Objective-C and have worked through the binding walkthrough above, you'll be in good shape for transitioning to the Xamarin platform. As a follow up, more detailed information on Xamarin.iOS binding technologies, including a comprehensive binding reference is available in the Binding Objective-C section.

Cross-Platform Development

Finally, after moving to Xamarin.iOS, you'll want to check out the cross-platform guidance we have, including case studies of reference applications we have developed, along with best practices for creating reusable, cross-platform code contained in the Building Cross-Platform Applications section.

Data & Azure cloud services

Learn how to store data, and build connected Xamarin.Forms applications with Azure cloud services.

Local data storage

CONCEPT

Files

Local databases

Azure services

CONCEPT

Azure mobile apps

Azure Cosmos DB document database

Azure notification hubs

Azure storage

Azure search

Azure functions

Azure cognitive services

CONCEPT

Introduction

Speech recognition

Spell check

Text translation

Emotion recognition

Web services

CONCEPT

Introduction

ASMX

WCF

REST

Authentication

CONCEPT

REST

Identity provider

Azure Active Directory B2C

Azure Cosmos DB document database authentication

Cross-platform guidance

Xamarin provides a cross-platform development solution for mobile, tablet, and desktop applications. This section covers details that apply no matter which platforms you're targeting.

Build cross-platform apps

GET STARTED

Requirements

Xamarin.Forms

.NET Standard

Cross-platform development on Q&A

HOW-TO GUIDE

App development lifecycle

Walkthrough

Language support

CONCEPT

Overview

C#

F#

Visual Basic

Razor HTML templates

Performance & security

CONCEPT

Xamarin.Forms performance

Android performance

iOS performance

Mac performance

Transport Layer Security (TLS) 1.2

Deployment & debugging

CONCEPT

Custom linker configuration

HOW-TO GUIDE

Connect to local web services

Multi-process debugging

Desktop developers

CONCEPT

App lifecycle comparison

UI controls comparison

Porting guidance