Working with native code on Windows

Edit

What is a React Native for Windows app?

When you create a React Native for Windows app with the CLI, you will get a Universal Windows Platform app (aka UWP app).

The Universal Windows Platform allows you to access a set of common functionality on all Windows devices via the Windows Runtime (WinRT). WinRT APIs can be accessed from C++ (via C++/WinRT), or via .NET C#.

WinRT support in .NET

The current publicly supported version of .NET (.NET UWP) has built-in support for WinRT.

Win32 Desktop apps vs. RNW apps

Whether you are new to Windows development, or you are a Win32 desktop app veteran, the following FAQs should answer some common questions.

When you add Windows support to a react native app via the steps described in the Install the windows extension section, the react-native-windows CLI will create a UWP app for you.

> those issues are important to you, it's highly recommended that you read Choosing C++ or C# for native code.

Regardless of the language of your app **RNW apps are UWP apps** and therefore have the following characteristics:

API surface

The set of APIs these app can access are a subset of all Windows APIs (i.e. those accessible via WinRT). See:

- Win32 and COM APIs for UWP apps
- CRT functions not supported in Universal Windows Platform apps
- Alternatives to Windows APIs in Universal Windows Platform (UWP) apps

Isolation

The app runs inside of an app container - a type of sandbox. This provides apps with a secure way to install, access system resources like the filesystem, and lets the system manage their lifetime (e.g. suspending the app when it isn't on the foreground). This means that by default an RNW app cannot access arbitrary filesystem locations, start arbitrary processes, etc. UWP apps that need to access these kinds of capabilities may be able to do so via App capability declarations.

Packaging

React Native Windows apps are signed and packaged. Packaging is a mechanism through which an app and its dependencies acquire an identity, which is used to determine whether API calls that require system capabilities (e.g. filesystem access) should succeed or not.

- via the Microsoft Store.
- via your private Store if you are a business or educational organization. See also Distribute LOB apps to enterprises.
- using App Installer.

It's worth noting that you cannot just "*copy an EXE*" as the app package contains more than just the main executable, including an app manifest, assets, dependent framework libraries, etc.

In addition, the Store submission process has these requirements:

- UWP apps submitted to the store must pass Windows App Certification Kit (WACK) validation.
- UWP apps written in C# or other managed languages submitted to the store must be built using the .NET Native toolchain. This is the default when building C# apps in Release mode, **but not in Debug**, so apps built in Debug will be rejected by the Store.

Use of non-WinRT libraries

Any libraries you use should be built as WinRT components. In other words, you cannot easily link libraries built for Win32 desktop apps without additional work.

- C++/CX is a dialect of C++ that allows writing UWP apps, however this is **not supported** for writing a RNW app. The article How to use existing C++ code in a Universal Windows Platform app talks about how to consume non-WinRT libraries in a WinRT context using C++/CX, but most of the content should be applicable to using C++/WinRT which is the supported way to write RNW apps.
- See also the guide for moving from C++/CX to C++/WinRT.
- Libraries built for .NET desktop framework cannot be directly accessed by UWP. You can create a .NET Standard library that calls into the .NET framework one, and call from the UWP app into the .NET Standard middleware.

`npx react-native run-windows` performs loose-file registration of your app in order to install it locally. When running `npx react-native run-windows` with the `--release` switch, the CLI will install the real package onto your local machine. This requires the app to be signed and for the certificate it uses for signing to be trusted by the machine the app is going to be installed on. See Create a certificate for package signing and Intro to certificates.

Debugging crashes and reporting issues

If your app is "hard crashing" (the native code hits an error condition and your app closes), you will want to investigate the native side of the code. If the issue is in the `Microsoft.ReactNative` layer, please file a bug in the React Native for Windows repo, and provide a native stack trace, and ideally a crash dump with symbols. For your convenience, you can use a script to collect a native crash dump and stack traces. Here are the instructions:

1. Download the script at https://aka.ms/RNW/analyze-crash.ps1, for example to C:\temp
2. Open an admin PowerShell
3. If you haven't enabled running unsigned scripts yet, do that by running: `Set-ExecutionPolicy Unrestricted`
4. Run the script and pass it the name of your app's exe (usually it will be your app's name): `C:\temp\analyze-crash.ps1 -ExeName MyApp`

The script will set up automatic crash dump collection for your app, download the native debugging tools (including the command line debugger cdb), and ask you to reproduce the crash.

At this point you can launch the app (e.g. from Start menu if you've already deployed it to the local device). When the app crashes, it will generate a crash dump. You can then press enter to resume execution of the script, and the script will use cdb to automatically analyze the crash dump, and output the results to a file `analyze.log`.

‹ Supported Community Modules

Choosing C++ or C# for native code ›