# Native Module Setup

Edit

> **This documentation is a work in progress and version-specific. Please check that the version of this document (top of page) matches the version of RN/RNW you're targeting. Examples (C# and C++/WinRT):**
>
> - Native Module Sample in `microsoft/react-native-windows-samples`
> - Sample App in `microsoft/react-native-windows/packages/microsoft-reactnative-sampleapps`

This guide will help set you up with the Visual Studio infrastructure to author your own stand-alone native module for React Native Windows. In this document we'll be creating the scaffolding for a `NativeModuleSample` native module.

## Development Environment

Make sure you have installed all of the development dependencies.

## Choose your own adventure

Once your development environment has been correctly configured, you have several options about how to access native APIs. You can either:

- Reference the APIs directly from within a React Native for Windows project

# Referencing Windows APIs within a React Native for Windows app project

If you are only planning on adding a native module to your existing React Native Windows app, i.e.:

1. You followed the Getting Started guide, where
2. You ran `npx react-native-windows-init` to add Windows to your project, and
3. You are just adding your native code to the app project under the `windows` folder.

Then you can simply open the Visual Studio solution in the `windows` folder and add the new files directly to the app project.

# Creating a new native module library project

The steps to create a new native module library project are:

1. Follow the official React Native instructions to create a blank native module project
2. Add Windows support to the newly created library

## Creating a blank native module project

Follow the official React Native instructions at https://reactnative.dev/docs/native-modules-setup,

or execute the following commands:

Now you'll have a new native module project under `NativeModuleSample`. Be sure to look at the command output for further steps you'll want to do before publishing the project.

At this point, follow the steps below to add Windows support to the newly created library.

# Adding Windows support to an existing library

> The steps below are written as if you're working with the `NativeModuleSample` example above, in the root folder of the project. Substitute the name of the library you're actually working on where appropriate, and ensure that you're working in the appropriate root folder of the library.

## Updating your `package.json`

Many native module libraries (including the default library template) target older versions of `react` and `react-native` than Windows supports, so you'll need to upgrade to newer versions in order to add support for `react-native-windows`.

> Properly defining your NPM dependencies is an essential part of creating and maintaining a React Native library, especially one that supports multiple platforms. The instructions here represent the minimum steps required to start targeting `react-native-windows`. If you're adding Windows support to a library you don't own, you'll need to work with the library owners to make sure any changes made to `package.json` are appropriate.

You can use the `npm info` command to find the correct versions to use. Let's assume you plan on building against the latest stable version of `react-native-windows`.

Use the following command to find the matching versions of `react`:

```
npm info react-native-windows@latest devDependencies.react
```

Copy

Take the result of that command (let's say it's `x.y.z`) and use it to upgrade the dev dependency:

```
yarn upgrade react@x.y.z --dev
```

Copy

You'll need to repeat the steps for `react-native`, i.e.:

```
npm info react-native-windows@latest devDependencies.react-native
```

Copy

Again, take the result of that command (let's say it's `0.x.y`) and use it to upgrade the dev dependency:

```
yarn upgrade react-native@0.x.y --dev
```

Copy

Now you should be ready to add Windows support with `react-native-windows-init`. The process is similar to adding Windows support to an app project, but you'll need to specify `--projectType lib`:

```
npx react-native-windows-init --version latest --projectType lib --overwrite
```

Copy

This defaults to a C++/WinRT project. If you want to create a C# based native module project, use:

That's it, you should be able to open `windows\NativeModuleSample.sln` and start working on your project.

## Testing your Build

To make sure that everything is working, you'll want to try building `NativeModuleSample`. First you'll want to make sure you've chosen a supported platform:

1. At the top, change the `Solution Platform` to `x86` or `x64`.
2. In the `Build` menu, select `Build Solution`.

## Next Steps

You have now created the scaffolding to build a native module or view manager. Now it's time to add the business logic to the module - follow the steps described in the Native Modules and View Managers documents.

## Making your module ready for consumption in an app

If you've followed the steps above, your module should be ready for consumption thanks to Autolinking.

However, there are some things you may need to check:

#### 1. Fixing relative NuGet paths

If you are writing a C++/WinRT module and have added any NuGet package dependencies, you'll see references to those packages in your vcxproj file as relative references e.g. `..\packages\...`. We need these to use the solution directory instead, so replace all mentions of `..\packages\` with `$(SolutionDir)\`.

**Example:**

# Testing the module before it gets published

**Option 1: Create a new test app**

1. Follow the getting started guide to create a new React Native Windows app.
2. Run `npm i <module-local-path> --save` (e.g. `npm i D:\NativeModuleSample --save`) to install the local module.
3. Link the native module.

**Option 2: Adding Windows support to existing sample app**

If you are working on an existing module that already has iOS and Android samples, and want to add Windows support to the existing test app, follow these steps (example of WebView module test app can be found here).

1. In a different directory, follow the getting started guide and create a new React Native Windows app.
2. Copy the `Windows` folder from the blank RNW app into the existing sample app's sample app's folder. (The RNW CLI helps create the correct project setup that you can then copy directly into the sample app.)
3. Open `sln` and `vxcproj` files and check `node_module` reference paths. Fix the paths if necessary based on how the folders are structured in native module repo (example).
4. Open the solution with Visual Studio and link native module.

> The project should build correctly at this point, but we still need to setup some special metro configurations for Windows in order to run the app without breaking iOS and Android bundling.

5. Add `metro.config.windows` for Windows bundling (example). Make sure the config file is at the root of the repo (see Metro bug #588).

8. Update JS main module path (relative path to metro `projectRoot`) in `App.cpp` if necessary (example).

## Adding tests for your module

We are using WebdriverIO + WinAppDriver for UI testing. More details here. For real world examples, check out `react-native-webview` or progress-view.

## Setup CI (continuous integration) pipeline for your module

When done developing your module, it's good practice to setup a CI pipeline with automated build and tests to avoid any future regressions. See the Setup Continuous Integration Pipeline for an RNW App for more information.

## Documenting Your Module

Once your module is complete, update react-native-community/directory so that its information on your native module is up to date. If you are building a native module which will be maintained by Microsoft, please update the Supported Community Modules documentation in react-native-windows-samples with your native module's information.

‹ Native UI Components

Native Modules vs Turbo Modules ›