React Native for Windows + macOS 0.72

Docs APIs Blog Resources Samples Support

NATIVE MODULES (WINDOWS)

Native Modules

Edit

> **This documentation and the underlying platform code is a work in progress. 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`

Sometimes an app needs access to a platform API that React Native doesn't have a corresponding module for yet. Maybe you want to reuse some existing .NET code without having to re-implement it in JavaScript, or write some high performance, multi-threaded code for image processing, a database, or any number of advanced extensions.

React Native was designed such that it is possible for you to write real native code and have access to the full power of the platform. This is a more advanced feature and we don't expect it to be part of the usual development process, however it is essential that it exists. If React Native doesn't support a native feature that you need, you should be able to build it yourself.

> NOTE: If you are building a widget that has a UI component, check out the Native UI Component guide.

Overview

   i. Add custom attributes to the class. These attributes allow you to define methods, properties, constants, and events that can be referenced from JavaScript.

2. Register your native module. Note that native modules defined within your app are automatically registered.
   i. Add the package to your React Native application.
3. Use your native module from your JavaScript code.

React Native for Windows supports authoring native modules in both C# and C++. Examples of both are provided below. Please see the Choosing C++ or C# for native code note for more information about which to choose.

> NOTE: If you are unable to use the reflection-based annotation approach, you can define native modules directly using the ABI. This is outlined in the Writing Native Modules without using Attributes document.

Initial Setup

Follow the Native Modules Setup Guide to create the Visual Studio infrastructure to author your own stand-alone native module for React Native Windows.

Once you have set up your development environment and project structure, you are ready to write code.

Open the Visual Studio solution in the `windows` folder and add the new files directly to the app project.

Sample Native Module

```
"codegenConfig": {
  "name": "NameOfYourApp",
  "type": "modules",
  "jsSrcsDir": "src",
  "windows": {
    "namespace": "YourAppCodegenNamespace"
  }
},
```

The values for `name`, `type`, `jsSrcsDir` are shared with react-native as documented here. The `windows` object will cause the windows-codegen task to generate windows specific codegen for any TurboModule spec files defined within your project. The `windows.namespace` property will control which C++ namespace these generated files will use.

2. Create JavaScript Specification

Modules should have a definition defined in a typed dialect of JavaScript (either TypeScript or Flow). Codegen will use these specifications to verify the interface provided by your native code.

There are two requirements the file containing this specification must meet:

- The file **must** be named `Native<MODULE_NAME>`, with a `.js` or `.jsx` extension when using Flow, or a `.ts`, or `.tsx` extension when using TypeScript.
- The file **must** export a `TurboModuleRegistrySpec` object.

Example Specification file `NativeFancyMath.ts`:

```
import type { TurboModule } from 'react-native/Libraries/TurboModule/RCTExport';
import { TurboModuleRegistry } from 'react-native';

export interface Spec extends TurboModule {

  getConstants: () => {
    E: number,
export default TurboModuleRegistry.get<Spec>(
  'FancyMath'
) as Spec | null;
```

> Note even through this file uses `TurboModuleRegistry` , Native Modules will still work with this JavaScript. The code is forward looking and will support Native Modules or TurboModules.

C# C++

Attributes

ATTRIBUTE	USE
`ReactModule`	Specifies the class is a native module.
`ReactMethod`	Specifies an asynchronous method.
`ReactSyncMethod`	Specifies a synchronous method.
`ReactConstant`	Specifies a field or property that represents a constant.
`ReactConstantProvider`	Specifies a method that provides a set of constants.
`ReactEvent`	Specifies a field or property that represents an event.
`ReactStruct`	Specifies a `struct` that can be used in native methods.
`ReactInit`	Specifies a class initialization module.
`ReactFunction`	Specifies a JavaScript function that you want exposed to your native code.

`FancyMath.cs` :

```
using System;
using Microsoft.ReactNative.Managed;

namespace NativeModuleSample
{
  [ReactModule]
  class FancyMath
  {
    [ReactConstant]
    public double E = Math.E;

    [ReactConstant("Pi")]
    public double PI = Math.PI;

    [ReactMethod("add")]
    public double Add(double a, double b)
    {
        double result = a + b;
        AddEvent(result);
        return result;
    }

    [ReactEvent]
    public ReactEvent<double> AddEvent { get; set; }
  }
}
```

First off, you see that we're making use of the `Microsoft.ReactNative.Managed` shared library, which provides the easiest (and recommended) experience for authoring native modules. `Microsoft.ReactNative.Managed` provides the mechanism that discovers the native module annotations to build bindings at runtime.

The `[ReactModule]` attribute says that the class is a React Native native module. It has an optional parameter for the module name visible to JavaScript and optionally the name of a

You can specify a different event emitter like this: `[ReactModule(EventEmitter = "mathEmitter")]` .

> NOTE: Using the default event emitter, `RCTDeviceEventEmitter` , all native event names must be **globally unique across all native modules** (even the ones built-in to RN). However, specifying your own event emitter means you'll need to create and register that too. This process is outlined in the Native Modules and React Native Windows (Advanced Topics) document.

The `[ReactConstant]` attribute is how you can define constants. Here `FancyMath` has defined two constants: `E` and `Pi` . When accessing these constants you should use `FancyMath.getConstants().E` . If you want to use another name in JS you could override the JS name like this: `[ReactConstant("e")]` .

The `[ReactMethod]` attribute is how you define methods. In `FancyMath` we have one method, `add` , which takes two doubles and returns their sum. As before, you can optionally customize the name like this: `[ReactMethod("add")]` .

The `[ReactEvent]` attribute is how you define events. In `FancyMath` we have one event, `AddEvent` , which uses the `ReactEvent<double>` delegate, where the double represents the type of the event data. Now whenever we invoke the `AddEvent` delegate in our native code (as we do above), an event named `"AddEvent"` will be raised in JavaScript. As before, you could have optionally customized the name in JS like this: `[ReactEvent("addEvent")]` .

4. Registering your Native Module

> IMPORTANT NOTE: When you create a new project via the CLI, the generated `ReactApplication` class will automatically register all native modules defined within

Now, we want to register our new `FancyMath` module with React Native so we can use it from JavaScript code. To do this, first we're going to create a `ReactPackageProvider` which implements `Microsoft.ReactNative.IReactPackageProvider`.

`ReactPackageProvider.cs`:

```
using Microsoft.ReactNative.Managed;

namespace NativeModuleSample
{
  public sealed class ReactPackageProvider : IReactPackageProvider
  {
    public void CreatePackage(IReactPackageBuilder packageBuilder)
    {
      packageBuilder.AddAttributedModules();
    }
  }
}
```

Here we've implemented the `CreatePackage` method, which receives `packageBuilder` to build contents of the package. Since we use reflection to discover and bind native module, we call `AddAttributedModules` extension method to register all native modules in our assembly that have the `ReactModule` attribute.

Now that we have the `ReactPackageProvider`, it's time to register it within our `ReactApplication`. We do that by simply adding the provider to the `PackageProviders` property.

`App.xaml.cs`:

```
using Microsoft.ReactNative;

namespace SampleApp
{
    sealed partial class App : ReactApplication
    {
            PackageProviders.Add(new NativeModuleSample.ReactPackageProvider());

            /* Other Init Code */
        }
    }
}
```

This example assumes that the `NativeModuleSample.ReactPackageProvider` we created above is in a different project (assembly) than our application. However you'll notice that by default we also added a `Microsoft.ReactNative.Managed.ReactPackageProvider`.

The `Microsoft.ReactNative.Managed.ReactPackageProvider` is a convenience that makes sure that all native modules and view managers defined within the app project automatically get registered. So if you're creating your native modules directly within the app project, you won't actually want to define a separate `ReactPackageProvider`.

5. Using your Native Module in JS

Now we have a Native Module which is registered with React Native Windows. How do we access it in JS? Here's a simple RN app:

`NativeModuleSample.js`:

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Alert,
  Text,
  View,
} from 'react-native';

import FancyMath from './NativeFancyMath'
import { NativeEventEmitter } from 'react-native';

const FancyMathEventEmitter = new NativeEventEmitter(FancyMath);
```

```
    FancyMathEventEmitter.addListener('AddEvent', eventHandler, this);
  }

  componentWillUnmount() {
    // Unsubscribing from FancyMath.AddEvent
    FancyMathEventEmitter.removeListener('AddEvent', eventHandler, this);
  }

  eventHandler(result) {
    console.log("Event was fired with: " + result);
  }

  _onPressHandler() {
    // Calling FancyMath.add method
    FancyMath.add(
      /* arg a */ FancyMath.getConstants().Pi,
      /* arg b */ FancyMath.E,
      /* callback */ function (result) {
        Alert.alert(
          'FancyMath',
          `FancyMath says ${FancyMath.getConstants().Pi} + ${FancyMath.getConstants().E} =
          [{ text: 'OK' }],
          {cancelable: false});
      });
  }

  render() {
    return (
      <View>
         <Text>FancyMath says PI = {FancyMath.getConstants().Pi}</Text>
         <Text>FancyMath says E = {FancyMath.getConstants().E}</Text>
         <Button onPress={this._onPressHandler} title="Click me!"/>
      </View>);
  }
}

AppRegistry.registerComponent('NativeModuleSample', () => NativeModuleSample);
```

To access your native modules, you need to import from your spec file, in this case `NativeFancyMath`. Since our modules fires events, we're also bringing in

Calls to methods are a little different due to the asynchronous nature of the JS engine. If the native method returns nothing, we can simply call the method. However, in this case `FancyMath.add()` returns a value, so in addition to the two necessary parameters we also include a callback function which will be called with the result of `FancyMath.add()`. In the example above, we can see that the callback raises an Alert dialog with the result value.

For events, you'll see that we created an instance of `NativeEventEmitter` passing in our `FancyMath` module, and called it `FancyMathEventEmitter`. We can then use the `FancyMathEventEmitter.addListener()` and `FancyMathEventEmitter.removeListener()` methods to subscribe to our `FancyMath.AddEvent`. In this case, when `AddEvent` is fired in the native code, `eventHandler` will get called, which logs the result to the console log.

‹ Publishing a React Native Windows App to the Microsoft Store

Native UI Components ›

REACT NATIVE DOCS

Getting Started

Tutorial

Components and APIs

More Resources

REACT NATIVE FOR WINDOWS + MACOS DOCS

Get Started with Windows

Get Started with macOS

React Native Windows Components and APIs

Native Modules

Native UI Components

CONNECT WITH US ON

Blog

Twitter

GitHub

Samples