NATIVE MODULES (WINDOWS)

Native Modules (Advanced)

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`

Writing Native Modules without using Attributes (C#)

The Native Modules page describes how you can author Native Modules in both C# and C++ using an attribute-based approach. The attribute-based approach makes it easy for you to author your native modules because it uses reflection to help React Native understand your native module.

In rare cases, you may need to write a native module against the ABI directly without using reflection. The high-level steps remain the same:

1. Author your native module
2. Register your native module
3. Use your native module

1. Authoring your Native Module

Here is the same `FancyMath` native module, but we do not use custom attributes and we're instead going to implement registration of its members ourselves without using reflection.

`FancyMath.cs` :

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

namespace NativeModuleSample
{
  class FancyMath
  {
    public double E = Math.E;

    public double PI = Math.PI;

    public double Add(double a, double b)
    {
        return a + b;
    }
  }
}
```

2. Registering your Native Module

Here's our `FancyMathPackageProvider` where we manually register our native module's members.

`FancyMathPackageProvider.cs` :

```
namespace NativeModuleSample
{
  public sealed class FancyMathPackageProvider : IReactPackageProvider
  {
    public void CreatePackage(IReactPackageBuilder packageBuilder)
```

```
          writer.WriteDouble(module.E);
          writer.WritePropertyName("Pi");
          writer.WriteDouble(module.PI);
        });
        moduleBuilder.AddMethod("add", MethodReturnType.Callback,
          (IJSValueReader inputReader,
          IJSValueWriter outputWriter,
          MethodResultCallback resolve,
          MethodResultCallback reject) => {
            double a = inputReader.GetNextArrayItem() ? inputReader.GetDouble() : throw new
            double b = inputReader.GetNextArrayItem() ? inputReader.GetDouble() : throw new
            double result = module.Add(a, b);
            outputWriter.WriteArrayBegin();
            outputWriter.WriteDouble(result);
            outputWriter.WriteArrayEnd();
            resolve(outputWriter);
          });
        return module;
      });
    }
  }
}
```

As you can see, it is possible to use the API directly, but the code looks a little bit more complicated. You are responsible for creating your own constant provider, serializing each constant into the `IJSValueWriter`. For methods, you are also responsible for de-serializing arguments, calling your code, and then serializing any return values. There are some (de)serialization helper methods to make that a little bit simpler:

`FancyMathPackageProvider.cs`:

```
namespace NativeModuleSample
{
  public sealed class FancyMathPackageProvider : IReactPackageProvider
  {
    public void CreatePackage(IReactPackageBuilder packageBuilder)
    {
      packageBuilder.AddModule("FancyMath", (IReactModuleBuilder moduleBuilder) => {
        moduleBuilder.AddMethod("add", MethodReturnType.Callback,
          (IJSValueReader inputReader,
          IJSValueWriter outputWriter,
          MethodResultCallback resolve,
          MethodResultCallback reject) => {
            double[] args;
            inputReader.ReadArgs(out args[0], out args[1]);
            double result = module.Add(args[0], args[1]);
            outputWriter.WriteArgs(result);
            resolve(outputWriter);
          });
        return module;
      });
    }
  }
}
```

It is possible to simplify the code even more by hiding the use of the value reader and writer interfaces:

`FancyMathPackageProvider.cs` :

```
namespace NativeModuleSample
{
  public sealed class FancyMathPackageProvider : IReactPackageProvider
  {
    public void CreatePackage(IReactPackageBuilder packageBuilder)
    {
      packageBuilder.AddModule("FancyMath", (IReactModuleBuilder moduleBuilder) => {
        var module = new FancyMath();
        moduleBuilder.AddConstantProvider(() => new Dictionary<string, object> {
          ["E"] = module.E,
          ["Pi"] = module.PI
        });
        moduleBuilder.AddMethod("add", MethodReturnType.Callback, (MethodCallContext callC
          double result = module.Add((double)callContext.Args[0], (double)callContext.Args[
          callContext.resolve(result);
        });
        return module;
```

This code looks much better, but we are getting the overhead of boxing values that involves memory allocation for each call. The code generation that we do using LINQ Expression avoids this extra overhead. Though, the initial use of reflection and code generation has some penalty too. From the maintenance point of view, the attributed code is much simple to support because we do not need to describe the same things in two different places.

And as for the rest of the code, once we have the `IReactPackageProvider` , registering that package is the same as in the example above that uses custom attributes.

3. Using your Native Module in JS

Using your native module in JS is the exact same as if the native module was defined using

`NativeFancyMath.ts` :

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

export interface Spec extends TurboModule {

  getConstants: () => {
    E: number,
    PI: number,
  };

  add(a: number, b: number): Promise<number>;
}

export default TurboModuleRegistry.get<Spec>(
  'FancyMath'
) as Spec | null;
```

`Sample.js` :

```
  Text,
  View,
} from 'react-native';
import FancyMath from './NativeFancyMath';

class NativeModuleSample extends Component {
  _onPressHandler() {
    FancyMath.add(
      /* arg a */ FancyMath.getConstants().Pi,
      /* arg b */ FancyMath.getConstants().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);
```

Native Modules with Custom Event Emitters

By default, native modules share a common `RCTDeviceEventEmitter` which emits the actual events into JavaScript. However, that comes with the limitation that all of the native modules

So say we have our `FancyMath` module, where we've specified `"MathEmitter"` as the name of the `EventEmitter`:

`FancyMath.cs`:

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

namespace NativeModuleSample
{
  [ReactModule(EventEmitterName = "MathEmitter")]
  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; }
  }
}
```

Now, when the native code calls `AddEvent`, that will be essentially translated into a JS call of `MathEmitter.emit("AddEvent", result)`.

So in order for this to work, you will need to create and register a `MathEmitter` module. You can create the module by and you can use the existing `EventEmitter` class.

```
const BatchedBridge = require('BatchedBridge');

BatchedBridge.registerLazyCallableModule("MathEmitter", () => {
  return new EventEmitter();
});
```

C# Native Modules with Initializer and as a way to access `ReactContext`

If your native module needs to perform some initialization logic on the native (C#) side, there is an easy mechanism for you to do so when the app is setting up. All you need to do is add a method that takes a `ReactContext` and has `[ReactInitializer]` attribute. If your native module needs to perform some operation periodically, you can do so by setting up a timer during your module's initialization as in the following example:

```
[ReactModule]
internal sealed class NativeModuleSample
{
  private ThreadPoolTimer m_timer;

  [ReactInitializer]
  public void Initialize(ReactContext reactContext)
  {
    m_timer = ThreadPoolTimer.CreatePeriodicTimer(
      new TimerElapsedHandler((timer) =>
      {
          // Do something every 5 seconds
      }),
      TimeSpan.FromSeconds(5)
    );
  }

  ~NativeModuleSample()
  {
    _timer?.Cancel();
```

on to the context passed onto the method that is marked `[ReactInitializer]`.

```
[ReactModule]
internal sealed class NativeModuleSample
{
  private ReactContext m_reactContext;

  [ReactInitializer]
  public void Initialize(ReactContext reactContext)
  {
    m_reactContext = reactContext;
  }

  [ReactMethod]
  public Task SampleAccessToHost()
  {
    var reactNativeHost = ReactNativeHost.FromContext(m_reactContext.Handle);
    // Use debugging api that reloads the instance
    reactNativeHost.ReloadInstance()
  }

  [ReactMethod]
  public Task EmitRCTDeviceEvent()
  {
    m_reactContext.EmitJSEvent("RCTDeviceEventEmitter", "MyCustomJsEvent", 42);
  }
}
```

‹ Autolinking Native Modules

Supported Community Modules ›

React Native for Windows + macOS 0.72

Docs APIs Blog Resources Samples Support

Native Modules

Native UI Components