Using Asynchronous Windows APIs

Edit

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

A common scenario for Native Modules is to call one or more native asynchronous methods from a JS asynchronous method. However it may not be immediately obvious how to properly bridge both asynchronous worlds, which can lead to unstable, difficult to debug code.

This document proposes some best patterns to follow when bridging asynchronous methods from JS to native code for React Native Windows. It assumes you've already familiar with the basics of setting up and writing Native Modules.

> The complete source for the examples below are provided within the Native Module Sample in `microsoft/react-native-windows-samples` .

Writing Native Modules that call Asynchronous Windows APIs

Let's write a native module which uses asynchronous Windows APIs to perform a simple HTTP request. We'll call it `SimpleHttpModule` and it needs a single, promise-based method

```
NativeModules.SimpleHttpModule.GetHttpResponse('https://microsoft.github.io/react-native-w
  .then(result => console.log(result))
  .catch(error => console.log(error));
```

`SimpleHttpModule` in C#

The native module support for C# supports the common asynchronous programming patterns established in C# using `async`, `await` and `Task<T>`.

To expose the module to JavaScript you need to declare a C# class. To indicate it should be exposed to JavaScript, you annotate with a `[ReactModule]` attribute like:

```
namespace NativeModuleSample
{
  [ReactModule]
  class SimpleHttpModule
  {
    // Methods go here.
  }
}
```

This makes an object available to JavaScript via the expression `NativeModules.SimpleHttpModule`. By default the JavaScript name will match the C# class name. If you don't want the name of the class to match the name in JavaScript, i.e. you want to access the module via the expression `NativeModules.CustomModule`. you can pass a custom name like:

```
  [ReactModule("CustomModule")]
  class SimpleHttpModule
  {
```

it is the default, to write these functions asynchronously. Writing asynchronous code in C# is pretty straight-forward and intuitive with the `async` and `await` keywords and the `Task<T>` types.

> If you're not familiar with writing asynchronous C# code, see Call asynchronous APIs in C# or Visual Basic and Asynchronous programming that will teach you the concepts if you are not familiar yet.

The function signature for a typical web request, annotated with the `[ReactMethod]` attribute will look like:

```
[ReactMethod]
public async Task<string> GetHttpResponseAsync(string uri) {
  ...
}
```

You are now free to fill in the logic like:

```
  // Create an HttpClient object
  var httpClient = new HttpClient();

  // Send the GET request asynchronously
  var httpResponseMessage = await httpClient.GetAsync(new Uri(uri));

  var content = await httpResponseMessage.Content.ReadAsStringAsync();

  return content;
```

The code takes the following steps:

1. Creates a `HttpClient` .
2. Asynchronously calls the `GetAsync` method to make an HTTP request for the URI.
3. Parses the status code out of the returned `HttpResponseMessage` object.

This code only returns a string. You might want to return a more complex object that contains both the content and the status code. For that you can simply declare a C# `struct` that will be marshaled to JavaScript like:

```
internal struct Result {
  public int statusCode { get; set; }
  public string content { get; set; }
}
```

It is recommended to follow JavaScript naming conventions here as of now there is no auto-mapping of names between the common style guides of C# and JS

To return the value you'll of course have to update the signature of the method from returning a `string` to the `Result` :

```
  public async Task<Result> GetHttpResponseAsync(string uri) {
```

as well as store the status code and update the return statement from `return content;` to:

```
    var statusCode = httpResponseMessage.StatusCode;

    return new Result()
    {
      statusCode = (int)statusCode,
      content = content,
    };
```

But wait, we've only discussed the success path, what happens if `GetHttpResponse` doesn't succeed? We don't handle any exceptions in this example. If an exception is thrown, how do we marshal an error back to JavaScript? That is actually taken care of for you by the framework: any exception in the task will be marshaled to the JavaScript side as a JavaScript exception.

Let's start with the asynchronous native method which performs the HTTP request:

```
static winrt::Windows::Foundation::IAsyncAction GetHttpResponseAsync(std::wstring   i) noe
{
  // Create an HttpClient object
  auto httpClient = winrt::Windows::Web::Http::HttpClient();

  // Send the GET request asynchronously
  auto httpResponseMessage = co_await httpClient.GetAsync(winrt::Windows::Foundation::Uri(

  // Parse response
  auto statusCode = httpResponseMessage.StatusCode();
  auto content = co_await httpResponseMessage.Content().ReadAsStringAsync();

  // TODO: How to return the result?
}
```

The `GetHttpResponseAsync` method is pretty straight-forward at this point, it takes a `wstring` URI and "returns" an `IAsyncAction` (which is to say, the method is asynchronous and doesn't actually return a value when it's done).

> If you're not familiar with writing asynchronous C++/WinRT code, see Concurrency and asynchronous operations with C++/WinRT.

Inside `GetHttpResponseAsync`, we see it:

1. Creates a `HttpClient`.
2. Asynchronous calls the `GetAsync` method to make an HTTP request for the URI.
3. Parses the status code out of the returned `HttpResponseMessage` object.
4. Asynchronously parses the content out of the returned `HttpResponseMessage` object.

```
namespace NativeModuleSample
{
  REACT_MODULE(SimpleHttpModule);
  struct SimpleHttpModule
  {
    REACT_METHOD(GetHttpResponse);
    void GetHttpResponse(std::wstring uri,
        winrt::Microsoft::ReactNative::ReactPromise<winrt::Microsoft::ReactNative::JSValue(
    {
    }
  };
}
```

Here we simply define `SimpleHttpModule` with an empty `GetHttpResponse` method.

Notice the method itself is `void` and that the last parameter in the signature is of type `ReactPromise<JSValueObject>`. This indicates to React Native Windows that we want a promise-based method in JS, and that the expected return value of a success is of type `JSValueObject`.

All method parameters before this final promise are the input parameters we expect to be marshaled in from the JS. In this case, we want a single string for the URI to request.

The `promise` object is our interface for handling the promise and marshaling a result to the JS. To do so we simply call `promise.Resolve()` with the result object (if the operation was a success) or `promise.Reject()` with an error (if the operation failed).

Now that we know how to return results, let's prep `GetHttpResponseAsync` to take in a `ReactPromise<JSValueObject>` parameter and use it:

```
static winrt::Windows::Foundation::IAsyncAction GetHttpResponseAsync(std::wstring i,
  winrt::Microsoft::ReactNative::ReactPromise<winrt::Microsoft::ReactNative::JSValueObject:
{
  auto capturedPromise = promise;
```

```
    auto httpResponseMessage = co_await httpClient.GetAsync(winrt::Windows::Foundation::Uri(

    // Parse response
    auto statusCode = httpResponseMessage.StatusCode();
    auto content = co_await httpResponseMessage.Content().ReadAsStringAsync();

    // Build result object
    auto resultObject = winrt::Microsoft::ReactNative::JSValueObject();

    resultObject["statusCode"] = static_cast<int>(statusCode);
    resultObject["content"] = winrt::to_string(content);

    capturedPromise.Resolve(resultObject);
  }
```

What have we done here? First off, we've "captured" the `promise` locally within the asynchronous method by copying it into `capturedPromise` . We do this because this is an

`ReactPromise` object getting deleted prematurely by React Native Windows.

> **Important:** Our only input parameter in this example is a `wstring` , but if your method uses `JSValue` , `JSValueArray` , or `JSValueObject` parameter types, you'll need to "capture" those with a copy too. Example:
>
> ```
> static winrt::Windows::Foundation::IAsyncAction MethodAsync(winrt::Microsoft::React
> {
>   auto captureOptions = options.Copy();
>   ...
> }
> ```

At the bottom of the method, we simply build the result object to be returned to JS, and pass it to `capturedPromise.Resolve()` . That's it for `GetHttpResponseAsync` - if the method

`GetHttpResponse` native module method.

```
REACT_METHOD(GetHttpResponse);
void GetHttpResponse(std::wstring uri,
    winrt::Microsoft::ReactNative::ReactPromise<winrt::Microsoft::ReactNative::JSValueObje
{
  auto asyncOp = GetHttpResponseAsync(uri, promise);
}
```

Looks simple enough, right? We call `GetHttpResponseAsync` with the `uri` and `promise` parameters, and get back an `IAsyncAction` object which we store in `asyncOp`. When this executes, `GetHttpResponseAsync` will return control when it hits its first `co_await`, which in turn will return control for the JS code to continue running. When everything in `GetHttpResponseAsync` succeeds, it itself is responsible for resolving the promise with the result.

But wait, what happens if `GetHttpResponseAsync` doesn't succeed? We don't handle any exceptions in this example, so if an exception is thrown, how do we marshal an error back to the JS? We have one more thing to do, and that's to check for unhandled exceptions:

```
REACT_METHOD(GetHttpResponse);
void GetHttpResponse(std::wstring uri,
    winrt::Microsoft::ReactNative::ReactPromise<winrt::Microsoft::ReactNative::JSValueObje
{
  auto asyncOp = GetHttpResponseAsync(uri, promise);
  asyncOp.Completed([promise](auto action, auto status)
  {
    if (status == winrt::Windows::Foundation::AsyncStatus::Error)
    {
      std::stringstream errorCode;
      errorCode << "0x" << std::hex << action.ErrorCode() << std::endl;

      auto error = winrt::Microsoft::ReactNative::ReactError();
      error.Message = "HRESULT " + errorCode.str() + ": " + std::system_category().message
      promise.Reject(error);
```

We've defined an `AsyncActionCompletedHandler` lambda and set it to be run when `asyncOp` completes. Here we check if the action failed (i.e. `status == AsyncStatus::Error`) and if so, we build a `ReactError` object where the message contains both the error code (a Windows `HRESULT`) and the error message for that code. Then we pass that error to `promise.Reject()`, thereby marshaling the error back to the JS.

> **Important:** This example shows the minimum case, where you don't handle any errors within `GetHttpResponseAsync`, but you're not limited to this. You're free to detect error conditions within your code and call `capturedPromise.Reject()` yourself with (more useful) error messages at any time. However you should *always* include this final
>
> especially when calling Windows APIs. Just be sure that you only call `Reject()` once and that nothing executes afterwards.

That's it! If you want to see the complete `SimpleHttpModule`, see `AsyncMethodExamples.h`.

Executing calls to API on the UI thread

Since version 0.64, calls to native modules no longer run on the UI thread. This means that each call to the APIs that must be executed on the UI thread now needs to be explicitly dispatched.

To do that the `UIDispatcher` should be used.

This section will cover the basic usage scenario of the `UIDispatcher` and its `Post()` method with the WinRT `FileOpenPicker` (for a description of opening files and folder with a picker on UWP please see the Open files and folders with a picker).

Using `UIDispatcher` with C#

Copy

```
[ReactMethod("openFile")]
public async void OpenFile()
{
  var picker = new Windows.Storage.Pickers.FileOpenPicker();
  // Other initialization code
  Windows.Storage.StorageFile file = await picker.PickSingleFileAsync();

  if (file != null)
  {
    // File opened successfully
  }
  else
  {
    // Error while opening the file
  }
}
```

However, starting with react-native-windows 0.64, this method would end up with `System.Exception: Invalid window handle`. Since the `FileOpenPicker` API requires running on the UI thread, we need to wrap this call with the `UIDispatcher.Post` method.

Copy

```
[ReactMethod("openFile")]
public void OpenFile()
{
  context.Handle.UIDispatcher.Post(async () => {
    var picker = new Windows.Storage.Pickers.FileOpenPicker();
    // Other initialization code
    Windows.Storage.StorageFile file = await picker.PickSingleFileAsync();

    if (file != null)
    {
      // File opened successfully
    }
    else
    {
      // Error while opening the file
    }
```

> **Note:** `UIDispatcher` is available via the `ReactContext`, which we can inject through a method marked as `ReactInitializer`
>
> ```
> [ReactInitializer]
> public void Initialize(ReactContext reactContext)
> {
>   context = reactContext;
> }
> ```

Now if we call the `openFile` method in our JS code the file picker's window will open.

Using `UIDispatcher` with C++/WinRT

Let's suppose we have the native module which opens and loads the file using the `FileOpenPicker`.

Following the official example the native module's method launching the picker would look like this:

```
REACT_METHOD(OpenFile, L"openFile");
winrt::fire_and_forget OpenFile() noexcept
{
  winrt::Windows::Storage::Pickers::FileOpenPicker openPicker;
  // Other initialization code
  winrt::Windows::Storage::StorageFile file = co_await openPicker.PickSingleFileAsync();

  if (file != nullptr)
  {
    // File opened successfully
  }
  else
  {
    // Error while opening the file
```

However, starting with react-native-windows 0.64, this method would end up with `ERROR_INVALID_WINDOW_HANDLE`. Since the `FileOpenPicker` API requires running on the UI thread, we need to wrap this call with the `UIDispatcher.Post` method.

```
REACT_METHOD(OpenFile, L"openFile");
void OpenFile() noexcept
{
  context.UIDispatcher().Post([]()->winrt::fire_and_forget {
    winrt::Windows::Storage::Pickers::FileOpenPicker openPicker;
    // Other initialization code
    winrt::Windows::Storage::StorageFile file = co_await openPicker.PickSingleFileAsync()

    if (file != nullptr)
    {
      // File opened successfully
    }
    else
    {
      // Error while opening the file
    }
  });
}
```

Note: `UIDispatcher` is available via the `ReactContext`, which we can inject through a method marked as `REACT_INIT`

```
REACT_INIT(Initialize);
void Initialize(const winrt::Microsoft::ReactNative::ReactContext& reactContext) n
{
  context = reactContext;
}
```

Now if we call the `openFile` method in our JS code the file picker's window will open.