

iOS Native Modules

! INFO

Native Module and Native Components are our stable technologies used by the legacy architecture. They will be deprecated in the future when the New Architecture will be stable. The New Architecture uses [Turbo Native Module](#) and [Fabric Native Components](#) to achieve similar results.

Welcome to Native Modules for iOS. Please start by reading the [Native Modules Intro](#) for an intro to what native modules are.

Create a Calendar Native Module

In the following guide you will create a native module, `CalendarModule`, that will allow you to access Apple's calendar APIs from JavaScript. By the end you will be able to call `CalendarModule.createCalendarEvent('Dinner Party', 'My House');` from JavaScript, invoking a native method that creates a calendar event.

Setup

To get started, open up the iOS project within your React Native application in Xcode. You can find your iOS project here within a React Native app:

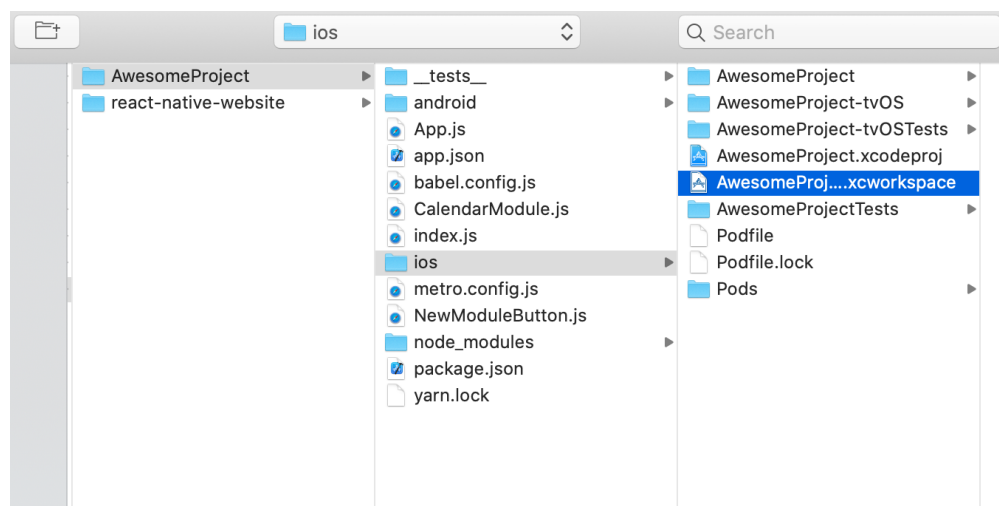




Image of where you can find your iOS project

We recommend using Xcode to write your native code. Xcode is built for iOS development, and using it will help you to quickly resolve smaller errors like code syntax.

Create Custom Native Module Files

The first step is to create our main custom native module header and implementation files. Create a new file called `RCTCalendarModule.h`

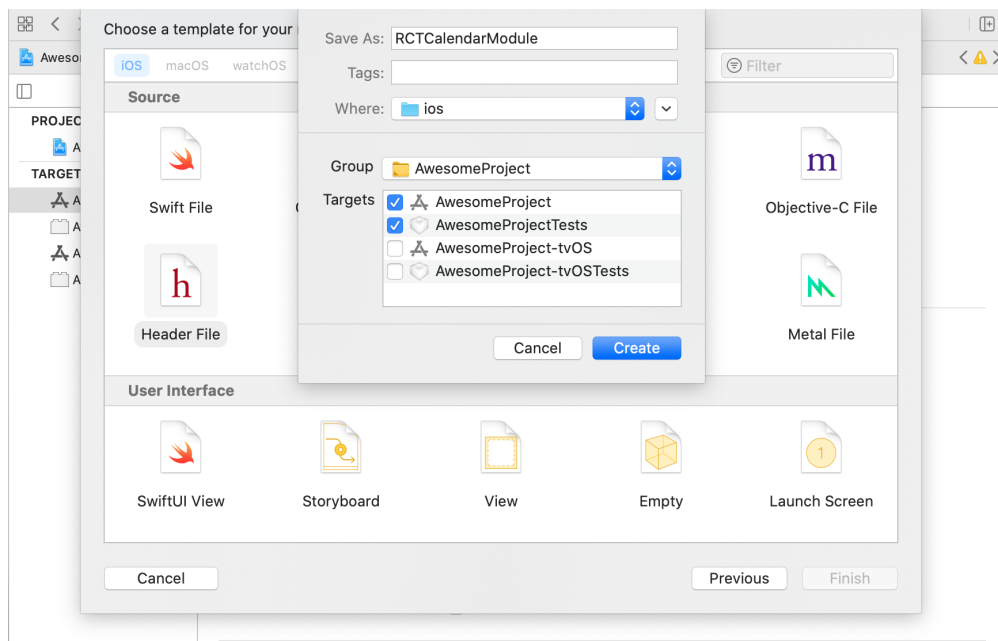


Image of creating a custom native module file within the same folder as AppDelegate

and add the following to it:

```
// RCTCalendarModule.h
#import <React/RCTBridgeModule.h>
@interface RCTCalendarModule : NSObject <RCTBridgeModule>
@end
```

You can use any name that fits the native module you are building. Name the class `RCTCalendarModule` since you are creating a calendar native module. Since ObjC does not have language-level support for namespaces like Java or C++, convention is to prepend the class name with a substring. This could be an abbreviation of your application name or your infra name. `RCT`, in this example, refers to React.

As you can see below, the `CalendarModule` class implements the `RCTBridgeModule` protocol. A native module is an Objective-C class that implements the `RCTBridgeModule` protocol.

Next up, let's start implementing the native module. Create the corresponding implementation file, `RCTCalendarModule.m`, in the same folder and include the following content:

```
// RCTCalendarModule.m
#import "RCTCalendarModule.h"

@implementation RCTCalendarModule

// To export a module named RCTCalendarModule
RCT_EXPORT_MODULE();

@end
```

Module Name

For now, your `RCTCalendarModule.m` native module only includes a `RCT_EXPORT_MODULE` macro, which exports and registers the native module class with React Native. The `RCT_EXPORT_MODULE` macro also takes an optional argument that specifies the name that the module will be accessible as in your JavaScript code.

This argument is not a string literal. In the example below `RCT_EXPORT_MODULE(CalendarModuleFoo)` is passed, not `RCT_EXPORT_MODULE("CalendarModuleFoo")`.

```
// To export a module named CalendarModuleFoo
RCT_EXPORT_MODULE(CalendarModuleFoo);
```

The native module can then be accessed in JS like this:

```
const {CalendarModuleFoo} = ReactNative.NativeModules;
```

If you do not specify a name, the JavaScript module name will match the Objective-C class name, with any "RCT" or "RK" prefixes removed.

Let's follow the example below and call `RCT_EXPORT_MODULE` without any arguments. As a result, the module will be exposed to React Native using the name `CalendarModule`, since that is the Objective-C class name, with RCT removed.

```
// Without passing in a name this will export the native module name as the Objective-C
class name with "RCT" removed
RCT_EXPORT_MODULE();
```

The native module can then be accessed in JS like this:

```
const {CalendarModule} = ReactNative.NativeModules;
```

Export a Native Method to JavaScript

React Native will not expose any methods in a native module to JavaScript unless explicitly told to. This can be done using the `RCT_EXPORT_METHOD` macro. Methods written in the `RCT_EXPORT_METHOD` macro are asynchronous and the return type is therefore always void. In order to pass a result from a `RCT_EXPORT_METHOD` method to JavaScript you can use callbacks or emit events (covered below). Let's go ahead and set up a native method for our `CalendarModule` native module using the `RCT_EXPORT_METHOD` macro. Call it `createCalendarEvent()` and for now have it take in name and location arguments as strings. Argument type options will be covered shortly.

```
RCT_EXPORT_METHOD(createCalendarEvent:(NSString *)name location:(NSString *)location)
{
}
```

Please note that the `RCT_EXPORT_METHOD` macro will not be necessary with TurboModules unless your method relies on RCT argument conversion (see argument types below). Ultimately, React Native will remove `RCT_EXPORT_MACRO`, so we discourage people from using `RCTConvert`. Instead, you can do the argument conversion within the method body.

Before you build out the `createCalendarEvent()` method's functionality, add a console log in the method so you can confirm it has been invoked from JavaScript in your React Native application. Use the `RCTLog` APIs from React. Let's import that header at the top of your file and then add the log call.

```
#import <React/RCTLog.h>
RCT_EXPORT_METHOD(createCalendarEvent:(NSString *)name location:(NSString *)location)
{
  RCTLogInfo(@"Pretending to create an event %@ at %@", name, location);
}
```

Synchronous Methods

You can use the `RCT_EXPORT_BLOCKING_SYNCHRONOUS_METHOD` to create a synchronous native method.

```
RCT_EXPORT_BLOCKING_SYNCHRONOUS_METHOD(getName)
{
  return [[UIDevice currentDevice] name];
}
```

The return type of this method must be of object type (id) and should be serializable to JSON. This means that the hook can only return nil or JSON values (e.g. `NSNumber`, `NSString`, `NSArray`, `NSDictionary`).

At the moment, we do not recommend using synchronous methods, since calling methods synchronously can have strong performance penalties and introduce threading-related bugs to your native modules. Additionally, please note that if you choose to use `RCT_EXPORT_BLOCKING_SYNCHRONOUS_METHOD`, your app can no longer use the Google Chrome debugger. This is because synchronous methods require the JS VM to share memory with the app. For the Google Chrome debugger, React Native runs inside the JS VM in Google Chrome, and communicates asynchronously with the mobile devices via WebSockets.

Test What You Have Built

At this point you have set up the basic scaffolding for your native module in iOS. Test that out by accessing the native module and invoking it's exported method in JavaScript.

Find a place in your application where you would like to add a call to the native module's `createCalendarEvent()` method. Below is an example of a component, `NewModuleButton` you can add in your app. You can invoke the native module inside `NewModuleButton`'s `onPress()` function.

```
import React from 'react';
import {NativeModules, Button} from 'react-native';

const NewModuleButton = () => {
  const onPress = () => {
    console.log('We will invoke the native module here!');
  };

  return (
    <Button
      title="Click to invoke your native module!"
      color="#841584"
      onPress={onPress}
    />
  );
};

export default NewModuleButton;
```

In order to access your native module from JavaScript you need to first import `NativeModules` from React Native:

```
import {NativeModules} from 'react-native';
```

You can then access the `CalendarModule` native module off of `NativeModules`.

```
const {CalendarModule} = NativeModules;
```

Now that you have the `CalendarModule` native module available, you can invoke your native method `createCalendarEvent()`. Below it is added to the `onPress()` method in `NewModuleButton`:

```
const onPress = () => {  
  CalendarModule.createCalendarEvent('testName', 'testLocation');  
};
```

The final step is to rebuild the React Native app so that you can have the latest native code (with your new native module!) available. In your command line, where the react native application is located, run the following:

`npm` **`Yarn`**

```
yarn ios
```

Building as You Iterate

As you work through these guides and iterate on your native module, you will need to do a native rebuild of your application to access your most recent changes from JavaScript. This is because the code that you are writing sits within the native part of your application. While React Native's metro bundler can watch for changes in JavaScript and rebuild JS bundle on the fly for you, it will not do so for native code. So if you want to test your latest native changes you need to rebuild by using the above command.

Recap ✨

You should now be able to invoke your `createCalendarEvent()` method on your native module in JavaScript. Since you are using `RCTLog` in the function, you can confirm your native method is being invoked by enabling debug mode in your app and looking at the JS console in Chrome or the mobile app debugger Flipper. You should see your `RCTLogInfo(@"Pretending to create an event %@ at %@", name, location); message` each time you invoke the native module method.

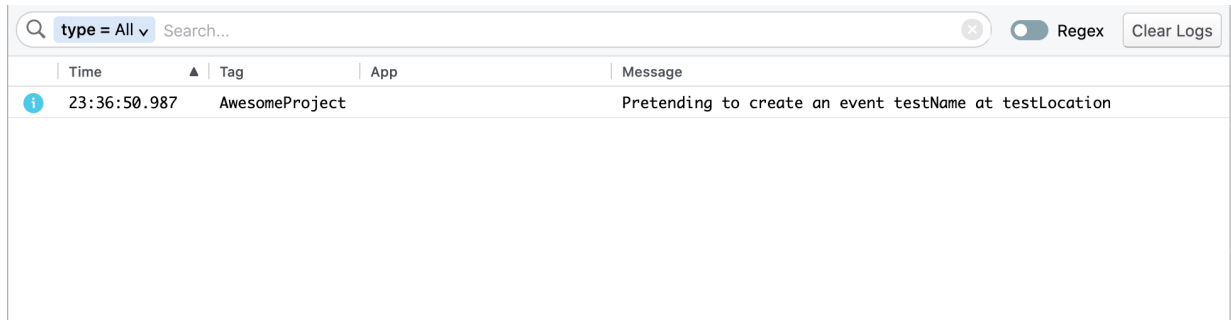


Image of iOS logs in Flipper

At this point you have created an iOS native module and invoked a method on it from JavaScript in your React Native application. You can read on to learn more about things like what argument types your native module method takes and how to setup callbacks and promises within your native module.

Beyond a Calendar Native Module

Better Native Module Export

Importing your native module by pulling it off of `NativeModules` like above is a bit clunky.

To save consumers of your native module from needing to do that each time they want to access your native module, you can create a JavaScript wrapper for the module. Create a new JavaScript file named `NativeCalendarModule.js` with the following content:

```
/**
 * This exposes the native CalendarModule module as a JS module. This has a
 * function 'createCalendarEvent' which takes the following parameters:
 *
 * 1. String name: A string representing the name of the event
```



```

* 2. String location: A string representing the location of the event
*/
import {NativeModules} from 'react-native';
const {CalendarModule} = NativeModules;
export default CalendarModule;

```

This JavaScript file also becomes a good location for you to add any JavaScript side functionality. For example, if you use a type system like TypeScript you can add type annotations for your native module here. While React Native does not yet support Native to JS type safety, with these type annotations, all your JS code will be type safe. These annotations will also make it easier for you to switch to type-safe native modules down the line. Below is an example of adding type safety to the Calendar Module:

```

/**
 * This exposes the native CalendarModule module as a JS module. This has a
 * function 'createCalendarEvent' which takes the following parameters:
 *
 * 1. String name: A string representing the name of the event
 * 2. String location: A string representing the location of the event
 */
import {NativeModules} from 'react-native';
const {CalendarModule} = NativeModules;
interface CalendarInterface {
  createCalendarEvent(name: string, location: string): void;
}
export default CalendarModule as CalendarInterface;

```

In your other JavaScript files you can access the native module and invoke its method like this:

```

import NativeCalendarModule from './NativeCalendarModule';
NativeCalendarModule.createCalendarEvent('foo', 'bar');

```

Note this assumes that the place you are importing `CalendarModule` is in the same hierarchy as `CalendarModule.js`. Please update the relative import as necessary.

Argument Types

When a native module method is invoked in JavaScript, React Native converts the arguments from JS objects to their Objective-C/Swift object analogues. So for example, if your Objective-C Native Module method accepts a `NSNumber`, in JS you need to call the method with a number. React Native will handle the conversion for you. Below is a list of the argument types supported for native module methods and the JavaScript equivalents they map to.

OBJECTIVE-C	JAVASCRIPT
<code>NSString</code>	<code>string</code> , <code>?string</code>
<code>BOOL</code>	<code>boolean</code>
<code>double</code>	<code>number</code>
<code>NSNumber</code>	<code>?number</code>
<code>NSArray</code>	<code>Array</code> , <code>?Array</code>
<code>NSDictionary</code>	<code>Object</code> , <code>?Object</code>
<code>RCTResponseSenderBlock</code>	<code>Function (success)</code>
<code>RCTResponseSenderBlock</code> , <code>RCTResponseErrorBlock</code>	<code>Function (failure)</code>
<code>RCTPromiseResolveBlock</code> , <code>RCTPromiseRejectBlock</code>	<code>Promise</code>

The following types are currently supported but will not be supported in TurboModules. Please avoid using them.

- `Function (failure)` -> `RCTResponseErrorBlock`
- `Number` -> `NSInteger`
- `Number` -> `CGFloat`
- `Number` -> `float`

For iOS, you can also write native module methods with any argument type that is supported by the `RCTConvert` class (see [RCTConvert](#) for details about what is supported). The `RCTConvert` helper functions all accept a JSON value as input and map it to a native Objective-C type or class.

Exporting Constants

A native module can export constants by overriding the native method `constantsToExport()`. Below `constantsToExport()` is overridden, and returns a Dictionary that contains a default event name property you can access in JavaScript like so:

```
- (NSDictionary *)constantsToExport
{
    return @{ @"DEFAULT_EVENT_NAME": @"New Event" };
}
```

The constant can then be accessed by invoking `getConstants()` on the native module in JS like so:

```
const {DEFAULT_EVENT_NAME} = CalendarModule.getConstants();
console.log(DEFAULT_EVENT_NAME);
```

Technically, it is possible to access constants exported in `constantsToExport()` directly off the `NativeModule` object. This will no longer be supported with TurboModules, so we encourage the community to switch to the above approach to avoid necessary migration down the line.

Note that the constants are exported only at initialization time, so if you change `constantsToExport()` values at runtime it won't affect the JavaScript environment.

For iOS, if you override `constantsToExport()` then you should also implement `+ requiresMainQueueSetup` to let React Native know if your module needs to be initialized on the main thread, before any JavaScript code executes. Otherwise you will see a warning that in the future your module may be initialized on a background thread unless you explicitly opt out with `+ requiresMainQueueSetup: NO`. If your module does not require access to UIKit, then you should respond to `+ requiresMainQueueSetup` with `NO`.

Callbacks

Native modules also support a unique kind of argument - a callback. Callbacks are used to pass data from Objective-C to JavaScript for asynchronous methods. They can also be used to asynchronously execute JS from the native side.

For iOS, callbacks are implemented using the type `RCTResponseSenderBlock`. Below the callback parameter `myCallback` is added to the `createCalendarEventMethod()`:

```
RCT_EXPORT_METHOD(createCalendarEvent:(NSString *)title
                  location:(NSString *)location
                  myCallback:(RCTResponseSenderBlock)callback)
```

You can then invoke the callback in your native function, providing whatever result you want to pass to JavaScript in an array. Note that `RCTResponseSenderBlock` accepts only one argument - an array of parameters to pass to the JavaScript callback. Below you will pass back the ID of an event created in an earlier call.

It is important to highlight that the callback is not invoked immediately after the native function completes—remember the communication is asynchronous.

```
RCT_EXPORT_METHOD(createCalendarEvent:(NSString *)title location:(NSString *)location
                  callback:(RCTResponseSenderBlock)callback)
{
    NSInteger eventId = ...
    callback(@[@(eventId)]);

    RCTLogInfo(@"Pretending to create an event %@ at %@", title, location);
}
```

This method could then be accessed in JavaScript using the following:

```
const onSubmit = () => {
  CalendarModule.createCalendarEvent(
    'Party',
    '04-12-2020',
    eventId => {
```

```

    console.log(`Created a new event with id ${eventId}`);
  },
);
};

```

A native module is supposed to invoke its callback only once. It can, however, store the callback and invoke it later. This pattern is often used to wrap iOS APIs that require delegates— see `RCTAlertManager` for an example. If the callback is never invoked, some memory is leaked.

There are two approaches to error handling with callbacks. The first is to follow Node’s convention and treat the first argument passed to the callback array as an error object.

```

RCT_EXPORT_METHOD(createCalendarEventCallback:(NSString *)title location:(NSString
*)location callback: (RCTResponseSenderBlock)callback)
{
  NSNumber *eventId = [NSNumber numberWithInt:123];
  callback(@[ [NSNull null], eventId]);
}

```

In JavaScript, you can then check the first argument to see if an error was passed through:

```

const onPress = () => {
  CalendarModule.createCalendarEventCallback(
    'testName',
    'testLocation',
    (error, eventId) => {
      if (error) {
        console.error(`Error found! ${error}`);
      }
      console.log(`event id ${eventId} returned`);
    },
  );
};

```

Another option is to use two separate callbacks: `onFailure` and `onSuccess`.

```

RCT_EXPORT_METHOD(createCalendarEventCallback:(NSString *)title
                  location:(NSString *)location
                  errorCallback: (RCTResponseSenderBlock)errorCallback
                  successCallback: (RCTResponseSenderBlock)successCallback)
{
  @try {
    NSNumber *eventId = [NSNumber numberWithInt:123];
    successCallback(@[eventId]);
  }

  @catch ( NSException *e ) {
    errorCallback(@[e]);
  }
}

```

Then in JavaScript you can add a separate callback for error and success responses:

```

const onPress = () => {
  CalendarModule.createCalendarEventCallback(
    'testName',
    'testLocation',
    error => {
      console.error(`Error found! ${error}`);
    },
    eventId => {
      console.log(`event id ${eventId} returned`);
    },
  );
};

```

If you want to pass error-like objects to JavaScript, use `RCTMakeError` from `RCTUtils.h`. Right now this only passes an Error-shaped dictionary to JavaScript, but React Native aims to automatically generate real JavaScript Error objects in the future. You can also provide a `RCTResponseErrorBlock` argument, which is used for error callbacks and accepts an `NSError` * object. Please note that this argument type will not be supported with TurboModules.

Promises

Native modules can also fulfill a promise, which can simplify your JavaScript, especially when using ES2016's `async/await` syntax. When the last parameter of a native module method is a `RCTPromiseResolveBlock` and `RCTPromiseRejectBlock`, its corresponding JS method will return a JS Promise object.

Refactoring the above code to use a promise instead of callbacks looks like this:

```
RCT_EXPORT_METHOD(createCalendarEvent:(NSString *)title
                  location:(NSString *)location
                  resolver:(RCTPromiseResolveBlock)resolve
                  rejecter:(RCTPromiseRejectBlock)reject)
{
  NSInteger eventId = createCalendarEvent();
  if (eventId) {
    resolve(@(eventId));
  } else {
    reject(@"event_failure", @"no event id returned", nil);
  }
}
```

The JavaScript counterpart of this method returns a Promise. This means you can use the `await` keyword within an `async` function to call it and wait for its result:

```
const onSubmit = async () => {
  try {
    const eventId = await CalendarModule.createCalendarEvent(
      'Party',
      'my house',
    );
    console.log(`Created a new event with id ${eventId}`);
  } catch (e) {
    console.error(e);
  }
};
```

Sending Events to JavaScript

Native modules can signal events to JavaScript without being invoked directly. For example, you might want to signal to JavaScript a reminder that a calendar event from the native iOS calendar app will occur soon. The preferred way to do this is to subclass `RCTEventEmitter`, implement `supportedEvents` and call `self sendEventWithName:`

Update your header class to import `RCTEventEmitter` and subclass `RCTEventEmitter`:

```
// CalendarModule.h

#import <React/RCTBridgeModule.h>
#import <React/RCTEventEmitter.h>

@interface CalendarModule : RCTEventEmitter <RCTBridgeModule>
@end
```

JavaScript code can subscribe to these events by creating a new `NativeEventEmitter` instance around your module.

You will receive a warning if you expend resources unnecessarily by emitting an event while there are no listeners. To avoid this, and to optimize your module's workload (e.g. by unsubscribing from upstream notifications or pausing background tasks), you can override `startObserving` and `stopObserving` in your `RCTEventEmitter` subclass.

```
@implementation CalendarManager
{
    bool hasListeners;
}

// Will be called when this module's first listener is added.
-(void)startObserving {
    hasListeners = YES;
    // Set up any upstream listeners or background tasks as necessary
}

// Will be called when this module's last listener is removed, or on dealloc.
-(void)stopObserving {
    hasListeners = NO;
    // Remove upstream listeners, stop unnecessary background tasks
}
```



```
- (void)calendarEventReminderReceived:(NSNotification *)notification
{
    NSString *eventName = notification.userInfo[@"name"];
    if (hasListeners) { // Only send events if anyone is listening
        [self sendEventWithName:@"EventReminder" body:@{@"name": eventName}];
    }
}
```

Threading

Unless the native module provides its own method queue, it shouldn't make any assumptions about what thread it's being called on. Currently, if a native module doesn't provide a method queue, React Native will create a separate GCD queue for it and invoke its methods there. Please note that this is an implementation detail and might change. If you want to explicitly provide a method queue for a native module, override the `(dispatch_queue_t) methodQueue` method in the native module. For example, if it needs to use a main-thread-only iOS API, it should specify this via:

```
- (dispatch_queue_t)methodQueue
{
    return dispatch_get_main_queue();
}
```

Similarly, if an operation may take a long time to complete, the native module can specify its own queue to run operations on. Again, currently React Native will provide a separate method queue for your native module, but this is an implementation detail you should not rely on. If you don't provide your own method queue, in the future, your native module's long running operations may end up blocking async calls being executed on other unrelated native modules. The `RCTAsyncLocalStorage` module here, for example, creates its own queue so the React queue isn't blocked waiting on potentially slow disk access.

```
- (dispatch_queue_t)methodQueue
{
    return dispatch_queue_create("com.facebook.React.AsyncLocalStorageQueue",
```

```
DISPATCH_QUEUE_SERIAL);
}
```

The specified `methodQueue` will be shared by all of the methods in your module. If only one of your methods is long-running (or needs to be run on a different queue than the others for some reason), you can use `dispatch_async` inside the method to perform that particular method's code on another queue, without affecting the others:

```
RCT_EXPORT_METHOD(doSomethingExpensive:(NSString *)param callback:
(RCTResponseSenderBlock)callback)
{
  dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    // Call long-running code on background thread
    ...
    // You can invoke callback from any thread/queue
    callback(@[...]);
  });
}
```

Sharing dispatch queues between modules

The `methodQueue` method will be called once when the module is initialized, and then retained by React Native, so there is no need to keep a reference to the queue yourself, unless you wish to make use of it within your module. However, if you wish to share the same queue between multiple modules then you will need to ensure that you retain and return the same queue instance for each of them.

Dependency Injection

React Native will create and initialize any registered native modules automatically. However, you may wish to create and initialize your own module instances to, for example, inject dependencies.

You can do this by creating a class that implements the `RCTBridgeDelegate` Protocol, initializing an `RCTBridge` with the delegate as an argument and initialising a `RCTRootView` with the initialized bridge.

```
id<RCTBridgeDelegate> moduleInitialiser = [[classThatImplementsRCTBridgeDelegate alloc]
init];

RCTBridge *bridge = [[RCTBridge alloc] initWithDelegate:moduleInitialiser
launchOptions:nil];

RCTRootView *rootView = [[RCTRootView alloc]
initWithBridge:bridge
moduleName:kModuleName
initialProperties:nil];
```

Exporting Swift

Swift doesn't have support for macros, so exposing native modules and their methods to JavaScript inside React Native requires a bit more setup. However, it works relatively the same. Let's say you have the same `CalendarModule` but as a Swift class:

```
// CalendarManager.swift

@objc(CalendarManager)
class CalendarManager: NSObject {

    @objc(addEvent:location:date:)
    func addEvent(_ name: String, location: String, date: NSNumber) -> Void {
        // Date is ready to use!
    }

    @objc
    func constantsToExport() -> [String: Any]! {
        return ["someKey": "someValue"]
    }
}
```

It is important to use the `@objc` modifiers to ensure the class and functions are exported properly to the Objective-C runtime.

Then create a private implementation file that will register the required information with React Native:

```
// CalendarManagerBridge.m
#import <React/RCTBridgeModule.h>

@interface RCT_EXTERN_MODULE(CalendarManager, NSObject)

RCT_EXTERN_METHOD(addEvent:(NSString *)name location:(NSString *)location date:(nonnull
NSNumber *)date)

@end
```

For those of you new to Swift and Objective-C, whenever you mix the two languages in an iOS project, you will also need an additional bridging file, known as a bridging header, to expose the Objective-C files to Swift. Xcode will offer to create this header file for you if you add your Swift file to your app through the Xcode **File**>**New File** menu option. You will need to import `RCTBridgeModule.h` in this header file.

```
// CalendarManager-Bridging-Header.h
#import <React/RCTBridgeModule.h>
```

You can also use `RCT_EXTERN_REMAP_MODULE` and `_RCT_EXTERN_REMAP_METHOD` to alter the JavaScript name of the module or methods you are exporting. For more information see [RCTBridgeModule](#).

Important when making third party modules: Static libraries with Swift are only supported in Xcode 9 and later. In order for the Xcode project to build when you use Swift in the iOS static library you include in the module, your main app project must contain Swift code and a bridging header itself. If your app project does not contain any Swift code, a workaround can be a single empty `.swift` file and an empty bridging header.


Reserved Method Names

invalidate()

Native modules can conform to the [RCTInvalidating](#) protocol on iOS by implementing the `invalidate()` method. This method can be invoked when the native bridge is invalidated

(ie: on devmode reload). Please use this mechanism as necessary to do the required cleanup for your native module.

Is this page useful?  

 Edit this page

*Last updated on **Aug 21, 2023***