Android 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 Android. 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 Android's calendar APIs from JavaScript. By the end, you will be able to call `CalendarModule.createCalendarEvent('Dinner Party', 'My House');` from JavaScript, invoking a Java/Kotlin method that creates a calendar event.

Setup

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

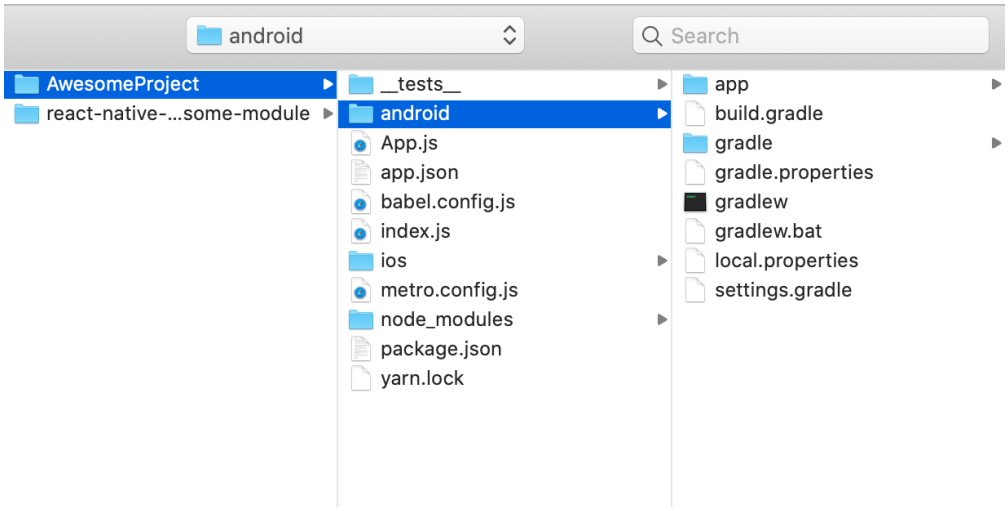

Image of where you can find your Android project

We recommend using Android Studio to write your native code. Android studio is an IDE built for Android development and using it will help you resolve minor issues like code syntax errors quickly.

We also recommend enabling Gradle Daemon to speed up builds as you iterate on Java/Kotlin code.

Create A Custom Native Module File

The first step is to create the (`CalendarModule.java` or `CalendarModule.kt`) Java/Kotlin file inside `android/app/src/main/java/com/your-app-name/` folder (the folder is the same for both Kotlin and Java). This Java/Kotlin file will contain your native module Java/Kotlin class.

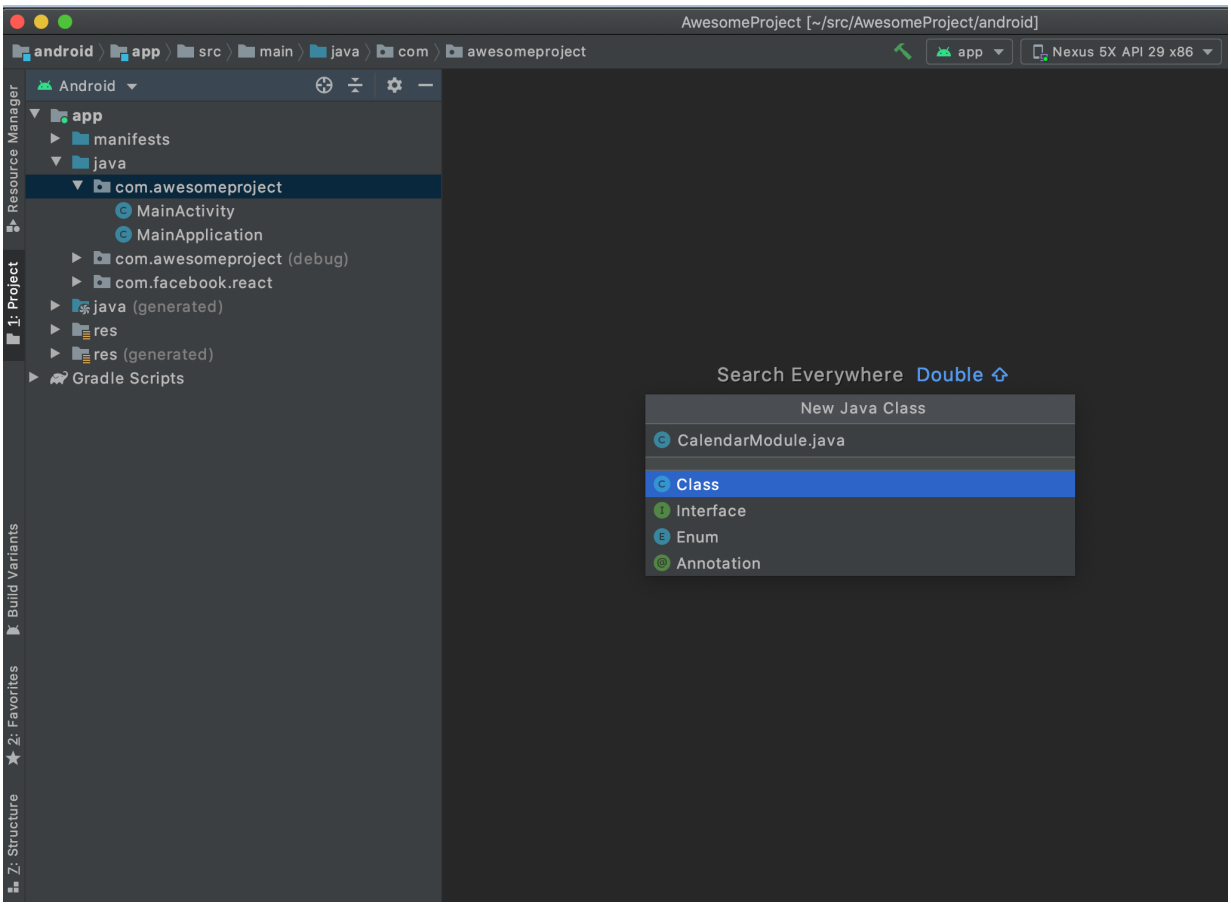

Image of how to add the CalendarModuleClass

Then add the following content:

Java Kotlin

```
package com.your-apps-package-name; // replace your-apps-package-name with your app's
package name
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;
import java.util.Map;
import java.util.HashMap;

public class CalendarModule extends ReactContextBaseJavaModule {
   CalendarModule(ReactApplicationContext context) {
       super(context);
```

```
    }
}
```

As you can see, your `CalendarModule` class extends the `ReactContextBaseJavaModule` class. For Android, Java/Kotlin native modules are written as classes that extend `ReactContextBaseJavaModule` and implement the functionality required by JavaScript.

> It is worth noting that technically Java/Kotlin classes only need to extend the `BaseJavaModule` class or implement the `NativeModule` interface to be considered a Native Module by React Native.

> However we recommend that you use `ReactContextBaseJavaModule`, as shown above. `ReactContextBaseJavaModule` gives access to the `ReactApplicationContext` (RAC), which is useful for Native Modules that need to hook into activity lifecycle methods. Using `ReactContextBaseJavaModule` will also make it easier to make your native module type-safe in the future. For native module type-safety, which is coming in future releases, React Native looks at each native module's JavaScript spec and generates an abstract base class that extends `ReactContextBaseJavaModule`.

Module Name

All Java/Kotlin native modules in Android need to implement the `getName()` method. This method returns a string, which represents the name of the native module. The native module can then be accessed in JavaScript using its name. For example, in the below code snippet, `getName()` returns `"CalendarModule"`.

Java Kotlin

```java
// add to CalendarModule.java
@Override
public String getName() {
   return "CalendarModule";
}
```

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

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

Export a Native Method to JavaScript

Next you will need to add a method to your native module that will create calendar events and can be invoked in JavaScript. All native module methods meant to be invoked from JavaScript must be annotated with `@ReactMethod`.

Set up a method `createCalendarEvent()` for `CalendarModule` that can be invoked in JS through `CalendarModule.createCalendarEvent()`. For now, the method will take in a name and location as strings. Argument type options will be covered shortly.

Java Kotlin

```
@ReactMethod
public void createCalendarEvent(String name, String location) {
}
```

Add a debug log in the method to confirm it has been invoked when you call it from your application. Below is an example of how you can import and use the Log class from the Android util package:

Java Kotlin

```
import android.util.Log;

@ReactMethod
public void createCalendarEvent(String name, String location) {
   Log.d("CalendarModule", "Create event called with name: " + name
   + " and location: " + location);
}
```

Once you finish implementing the native module and hook it up in JavaScript, you can follow these steps to view the logs from your app.

Synchronous Methods

You can pass `isBlockingSynchronousMethod = true` to a native method to mark it as a synchronous method.

Java Kotlin

```
@ReactMethod(isBlockingSynchronousMethod = true)
```

At the moment, we do not recommend this, 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 enable `isBlockingSynchronousMethod`, 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.

Register the Module (Android Specific)

Once a native module is written, it needs to be registered with React Native. In order to do so, you need to add your native module to a `ReactPackage` and register the `ReactPackage` with React Native. During initialization, React Native will loop over all packages, and for each `ReactPackage`, register each native module within.

React Native invokes the method `createNativeModules()` on a `ReactPackage` in order to get the list of native modules to register. For Android, if a module is not instantiated and returned in createNativeModules it will not be available from JavaScript.

To add your Native Module to `ReactPackage`, first create a new Java/Kotlin Class named (`MyAppPackage.java` or `MyAppPackage.kt`) that implements `ReactPackage` inside the `android/app/src/main/java/com/your-app-name/` folder:

Then add the following content:

Java Kotlin

```
package com.your-app-name; // replace your-app-name with your app's name
import com.facebook.react.ReactPackage;
import com.facebook.react.bridge.NativeModule;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.uimanager.ViewManager;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MyAppPackage implements ReactPackage {

   @Override
   public List<ViewManager> createViewManagers(ReactApplicationContext reactContext) {
       return Collections.emptyList();
   }

   @Override
   public List<NativeModule> createNativeModules(
           ReactApplicationContext reactContext) {
       List<NativeModule> modules = new ArrayList<>();

       modules.add(new CalendarModule(reactContext));

       return modules;
   }

}
```

This file imports the native module you created, `CalendarModule`. It then instantiates `CalendarModule` within the `createNativeModules()` function and returns it as a list of `NativeModules` to register. If you add more native modules down the line, you can also instantiate them and add them to the list returned here.

> It is worth noting that this way of registering native modules eagerly initializes all native modules when the application starts, which adds to the startup time of an application. You can use TurboReactPackage as an alternative. Instead of `createNativeModules`,

> which return a list of instantiated native module objects, TurboReactPackage implements a `getModule(String name, ReactApplicationContext rac)` method that creates the native module object, when required. TurboReactPackage is a bit more complicated to implement at the moment. In addition to implementing a `getModule()` method, you have to implement a `getReactModuleInfoProvider()` method, which returns a list of all the native modules the package can instantiate along with a function that instantiates them, example here. Again, using TurboReactPackage will allow your application to have a faster startup time, but it is currently a bit cumbersome to write. So proceed with caution if you choose to use TurboReactPackages.

To register the `CalendarModule` package, you must add `MyAppPackage` to the list of packages returned in ReactNativeHost's `getPackages()` method. Open up your `MainApplication.java` or `MainApplication.kt` file, which can be found in the following path: `android/app/src/main/java/com/your-app-name/`.

Locate ReactNativeHost's `getPackages()` method and add your package to the packages list `getPackages()` returns:

Java Kotlin

```
@Override
  protected List<ReactPackage> getPackages() {
    @SuppressWarnings("UnnecessaryLocalVariable")
    List<ReactPackage> packages = new PackageList(this).getPackages();
    // below MyAppPackage is added to the list of packages returned
    packages.add(new MyAppPackage());
    return packages;
  }
```

You have now successfully registered your native module for Android!

Test What You Have Built

At this point, you have set up the basic scaffolding for your native module in Android. Test that out by accessing the native module and invoking its 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 android
```

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 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 the app. In our example this occurs by pressing the `NewModuleButton`. You can confirm this by viewing the log you set up in your `createCalendarEvent()` method. You can follow these steps to view ADB logs in your app. You should then be able to search for your `Log.d` message (in our example "Create event called with name: testName and location: testLocation") and see your message logged each time you invoke your native module method.

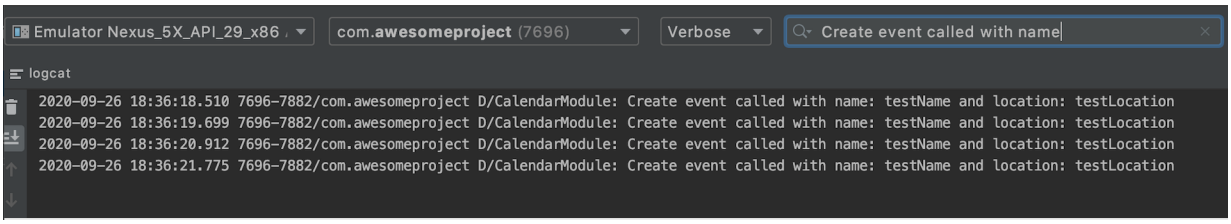

Image of ADB logs in Android Studio

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

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 `CalendarModule.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, all your JS code will be type safe. Doing so 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 CalendarModule:

```
/**
 * 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 CalendarModule from './CalendarModule';
CalendarModule.createCalendarEvent('foo', 'bar');
```

> 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 Java/Kotlin object analogues. So for example, if your Java Native Module method accepts a double, 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.

JAVA	KOTLIN	JAVASCRIPT
Boolean	Boolean	?boolean
boolean		boolean
Double	Double	?number
double		number
String	String	string
Callback	Callback	Function
Promise	Promise	Promise
ReadableMap	ReadableMap	Object
ReadableArray	ReadableArray	Array

> The following types are currently supported but will not be supported in TurboModules. Please avoid using them:
>
> - Integer Java/Kotlin -> ?number
> - Float Java/Kotlin -> ?number
> - int Java -> number
> - float Java -> number

For argument types not listed above, you will need to handle the conversion yourself. For example, in Android, `Date` conversion is not supported out of the box. You can handle the conversion to the `Date` type within the native method yourself like so:

Java Kotlin

```
String dateFormat = "yyyy-MM-dd";
SimpleDateFormat sdf = new SimpleDateFormat(dateFormat);
Calendar eStartDate = Calendar.getInstance();
try {
    eStartDate.setTime(sdf.parse(startDate));
```

```
    }
```

Exporting Constants

A native module can export constants by implementing the native method `getConstants()`, which is available in JS. Below you will implement `getConstants()` and return a Map that contains a `DEFAULT_EVENT_NAME` constant you can access in JavaScript:

Java Kotlin

```java
@Override
public Map<String, Object> getConstants() {
   final Map<String, Object> constants = new HashMap<>();
   constants.put("DEFAULT_EVENT_NAME", "New Event");
   return constants;
}
```

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

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

Technically it is possible to access constants exported in `getConstants()` directly off the native module 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.

> That currently constants are exported only at initialization time, so if you change getConstants values at runtime it won't affect the JavaScript environment. This will change with Turbomodules. With Turbomodules, `getConstants()` will become a regular native module method, and each invocation will hit the native side.

Callbacks

Native modules also support a unique kind of argument: a callback. Callbacks are used to pass data from Java/Kotlin to JavaScript for asynchronous methods. They can also be used to asynchronously execute JavaScript from the native side.

In order to create a native module method with a callback, first import the `Callback` interface, and then add a new parameter to your native module method of type `Callback`. There are a couple of nuances with callback arguments that will soon be lifted with TurboModules. First off, you can only have two callbacks in your function arguments- a successCallback and a failureCallback. In addition, the last argument to a native module method call, if it's a function, is treated as the successCallback, and the second to last argument to a native module method call, if it's a function, is treated as the failure callback.

Java Kotlin

```
import com.facebook.react.bridge.Callback;

@ReactMethod
public void createCalendarEvent(String name, String location, Callback callBack) {
}
```

You can invoke the callback in your Java/Kotlin method, providing whatever data you want to pass to JavaScript. Please note that you can only pass serializable data from native code to JavaScript. If you need to pass back a native object you can use `WriteableMaps`, if you need to use a collection use `WritableArrays`. It is also important to highlight that the callback is not invoked immediately after the native function completes. Below the ID of an event created in an earlier call is passed to the callback.

Java Kotlin

```
@ReactMethod
 public void createCalendarEvent(String name, String location, Callback callBack) {
     Integer eventId = ...
     callBack.invoke(eventId);
 }
```

This method could then be accessed in JavaScript using:

```
const onPress = () => {
  CalendarModule.createCalendarEvent(
    'Party',
    'My House',
    eventId => {
      console.log(`Created a new event with id ${eventId}`);
    },
  );
};
```

Another important detail to note is that a native module method can only invoke one callback, one time. This means that you can either call a success callback or a failure callback, but not both, and each callback can only be invoked at most one time. A native module can, however, store the callback and invoke it later.

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 as an error object.

Java Kotlin

```
@ReactMethod
 public void createCalendarEvent(String name, String location, Callback callBack) {
     Integer eventId = ...
     callBack.invoke(null, eventId);
 }
```

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

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

```
      console.log(`event id ${eventId} returned`);
    },
  );
};
```

Another option is to use an onSuccess and onFailure callback:

Java Kotlin

```
@ReactMethod
public void createCalendarEvent(String name, String location, Callback myFailureCallback,
Callback mySuccessCallback) {
}
```

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

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

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 Java/Kotlin method is a Promise, its corresponding JS method will return a JS Promise object.

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

Java Kotlin

```java
import com.facebook.react.bridge.Promise;

@ReactMethod
public void createCalendarEvent(String name, String location, Promise promise) {
    try {
        Integer eventId = ...
        promise.resolve(eventId);
    } catch(Exception e) {
        promise.reject("Create Event Error", e);
    }
}
```

> Similar to callbacks, a native module method can either reject or resolve a promise (but not both) and can do so at most once. This means that you can either call a success callback or a failure callback, but not both, and each callback can only be invoked at most one time. A native module can, however, store the callback and invoke it later.

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:

```javascript
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);
  }
};
```

The reject method takes different combinations of the following arguments:

Java Kotlin

```
String code, String message, WritableMap userInfo, Throwable throwable
```

For more detail, you can find the `Promise.java` interface here. If `userInfo` is not provided, ReactNative will set it to null. For the rest of the parameters React Native will use a default value. The `message` argument provides the error `message` shown at the top of an error call stack. Below is an example of the error message shown in JavaScript from the following reject call in Java/Kotlin.

Java/Kotlin reject call:

Java Kotlin

```
promise.reject("Create Event error", "Error parsing date", e);
```

Error message in React Native App when promise is rejected:

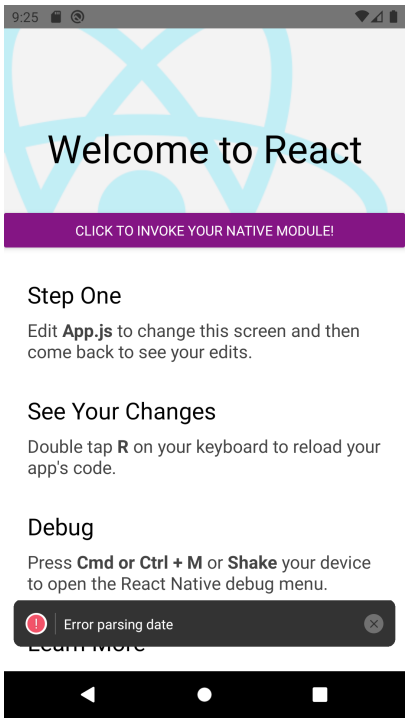

Image of error message

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 Android calendar app will occur soon. The easiest way to do this is to use the `RCTDeviceEventEmitter` which can be obtained from the `ReactContext` as in the code snippet below.

Java Kotlin

```
...
import com.facebook.react.modules.core.DeviceEventManagerModule;
import com.facebook.react.bridge.WritableMap;
import com.facebook.react.bridge.Arguments;
...
private void sendEvent(ReactContext reactContext,
                       String eventName,
                       @Nullable WritableMap params) {
 reactContext
     .getJSModule(DeviceEventManagerModule.RCTDeviceEventEmitter.class)
     .emit(eventName, params);
}

private int listenerCount = 0;

@ReactMethod
public void addListener(String eventName) {
  if (listenerCount == 0) {
    // Set up any upstream listeners or background tasks as necessary
  }

  listenerCount += 1;
}

@ReactMethod
public void removeListeners(Integer count) {
  listenerCount -= count;
  if (listenerCount == 0) {
    // Remove upstream listeners, stop unnecessary background tasks
  }
}
...
WritableMap params = Arguments.createMap();
params.putString("eventProperty", "someValue");
```

```
...
sendEvent(reactContext, "EventReminder", params);
```

JavaScript modules can then register to receive events by `addListener` on the NativeEventEmitter class.

```
import {NativeEventEmitter, NativeModules} from 'react-native';
...
useEffect(() => {
    const eventEmitter = new NativeEventEmitter(NativeModules.ToastExample);
    let eventListener = eventEmitter.addListener('EventReminder', event => {
      console.log(event.eventProperty) // "someValue"
    });

    // Removes the listener once unmounted
    return () => {
      eventListener.remove();
    };
  }, []);
```

Getting Activity Result from startActivityForResult

You'll need to listen to `onActivityResult` if you want to get results from an activity you started with `startActivityForResult`. To do this, you must extend `BaseActivityEventListener` or implement `ActivityEventListener`. The former is preferred as it is more resilient to API changes. Then, you need to register the listener in the module's constructor like so:

Java Kotlin

```
reactContext.addActivityEventListener(mActivityResultListener);
```

Now you can listen to `onActivityResult` by implementing the following method:

Java Kotlin

```
@Override
public void onActivityResult(
 final Activity activity,
 final int requestCode,
 final int resultCode,
 final Intent intent) {
 // Your logic here
}
```

Let's implement a basic image picker to demonstrate this. The image picker will expose the method `pickImage` to JavaScript, which will return the path of the image when called.

Java Kotlin

```
public class ImagePickerModule extends ReactContextBaseJavaModule {

  private static final int IMAGE_PICKER_REQUEST = 1;
  private static final String E_ACTIVITY_DOES_NOT_EXIST = "E_ACTIVITY_DOES_NOT_EXIST";
  private static final String E_PICKER_CANCELLED = "E_PICKER_CANCELLED";
  private static final String E_FAILED_TO_SHOW_PICKER = "E_FAILED_TO_SHOW_PICKER";
  private static final String E_NO_IMAGE_DATA_FOUND = "E_NO_IMAGE_DATA_FOUND";

  private Promise mPickerPromise;

  private final ActivityEventListener mActivityEventListener = new
BaseActivityEventListener() {

    @Override
    public void onActivityResult(Activity activity, int requestCode, int resultCode,
Intent intent) {
      if (requestCode == IMAGE_PICKER_REQUEST) {
        if (mPickerPromise != null) {
          if (resultCode == Activity.RESULT_CANCELED) {
            mPickerPromise.reject(E_PICKER_CANCELLED, "Image picker was cancelled");
          } else if (resultCode == Activity.RESULT_OK) {
            Uri uri = intent.getData();

            if (uri == null) {
              mPickerPromise.reject(E_NO_IMAGE_DATA_FOUND, "No image data found");
            } else {
              mPickerPromise.resolve(uri.toString());
            }
          }
```

```
            mPickerPromise = null;
          }
        }
      }
    };

    ImagePickerModule(ReactApplicationContext reactContext) {
      super(reactContext);

      // Add the listener for `onActivityResult`
      reactContext.addActivityEventListener(mActivityEventListener);
    }

    @Override
    public String getName() {
      return "ImagePickerModule";
    }

    @ReactMethod
    public void pickImage(final Promise promise) {
      Activity currentActivity = getCurrentActivity();

      if (currentActivity == null) {
        promise.reject(E_ACTIVITY_DOES_NOT_EXIST, "Activity doesn't exist");
        return;
      }

      // Store the promise to resolve/reject when picker returns data
      mPickerPromise = promise;

      try {
        final Intent galleryIntent = new Intent(Intent.ACTION_PICK);

        galleryIntent.setType("image/*");

        final Intent chooserIntent = Intent.createChooser(galleryIntent, "Pick an image");

        currentActivity.startActivityForResult(chooserIntent, IMAGE_PICKER_REQUEST);
      } catch (Exception e) {
        mPickerPromise.reject(E_FAILED_TO_SHOW_PICKER, e);
        mPickerPromise = null;
      }
    }
  }
```

Listening to Lifecycle Events

Listening to the activity's LifeCycle events such as `onResume`, `onPause` etc. is very similar to how `ActivityEventListener` was implemented. The module must implement `LifecycleEventListener`. Then, you need to register a listener in the module's constructor like so:

Java Kotlin

```
reactContext.addLifecycleEventListener(this);
```

Now you can listen to the activity's LifeCycle events by implementing the following methods:

Java Kotlin

```
@Override
public void onHostResume() {
   // Activity `onResume`
}
@Override
public void onHostPause() {
   // Activity `onPause`
}
@Override
public void onHostDestroy() {
   // Activity `onDestroy`
}
```

Threading

To date, on Android, all native module async methods execute on one thread. Native modules should not have any assumptions about what thread they are being called on, as the current assignment is subject to change in the future. If a blocking call is required, the heavy work should be dispatched to an internally managed worker thread, and any callbacks distributed from there.

Is this page useful? 

Edit this page

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