

Integration with Existing Apps

React Native is great when you are starting a new mobile app from scratch. However, it also works well for adding a single view or user flow to existing native applications. With a few steps, you can add new React Native based features, screens, views, etc.

The specific steps are different depending on what platform you're targeting.

Android (Kotlin)

Android (Java)

iOS (Objective-C)

iOS (Swift)

Key Concepts

The keys to integrating React Native components into your Android application are to:

1. Set up React Native dependencies and directory structure.
2. Develop your React Native components in JavaScript.
3. Add a `ReactRootView` to your Android app. This view will serve as the container for your React Native component.
4. Start the React Native server and run your native application.
5. Verify that the React Native aspect of your application works as expected.

Prerequisites

Follow the React Native CLI Quickstart in the [environment setup guide](#) to configure your development environment for building React Native apps for Android.

1. Set up directory structure

To ensure a smooth experience, create a new folder for your integrated React Native project, then copy your existing Android project to an `/android` subfolder.

2. Install JavaScript dependencies

Go to the root directory for your project and create a new `package.json` file with the following contents:

```
{
  "name": "MyReactNativeApp",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "yarn react-native start"
  }
}
```

Next, install the `react` and `react-native` packages. Open a terminal or command prompt, then navigate to the directory with your `package.json` file and run:

npm **Yarn**

```
yarn add react-native
```

This will print a message similar to the following (scroll up in the yarn output to see it):

```
warning "react-native@0.70.5" has unmet peer dependency "react@18.1.0"
```

This is OK, it means we also need to install React:

npm **Yarn**

```
yarn add react@version_printed_above
```

Yarn has created a new `/node_modules` folder. This folder stores all the JavaScript dependencies required to build your project.

Add `node_modules/` to your `.gitignore` file.

Adding React Native to your app

Configuring Gradle

React Native uses the React Native Gradle Plugin to configure your dependencies and project setup.

First, let's edit your `settings.gradle` file by adding this line:

```
includeBuild('../node_modules/@react-native/gradle-plugin')
```

Then you need to open your top level `build.gradle` and include this line:

```
buildscript {  
    repositories {  
        google()  
        mavenCentral()  
    }  
    dependencies {  
        classpath("com.android.tools.build:gradle:7.3.1")  
+       classpath("com.facebook.react:react-native-gradle-plugin")  
    }  
}
```

This makes sure the React Native Gradle Plugin is available inside your project. Finally, add those lines inside your app's `build.gradle` file (it's a different `build.gradle` file inside your app folder):

```
apply plugin: "com.android.application"  
+apply plugin: "com.facebook.react"  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    // Other dependencies here  
+    implementation "com.facebook.react:react-android"  
+    implementation "com.facebook.react:hermes-android"  
}
```

Those dependencies are available on `mavenCentral()` so make sure you have it defined in your `repositories{}` block.

! INFO

We intentionally don't specify the version for those `implementation` dependencies as the React Native Gradle Plugin will take care of it. If you don't use the React Native Gradle Plugin, you'll have to specify version manually.

Enable native modules autolinking

To use the power of autolinking, we have to apply it a few places. First add the following entry to `settings.gradle`:

```
apply from: file("../node_modules/@react-native-community/cli-platform-android/native_modules.gradle"); applyNativeModulesSettingsGradle(settings)
```

Next add the following entry at the very bottom of the `app/build.gradle`:

```
apply from: file("../..node_modules/@react-native-community/cli-platform-android/native_modules.gradle"); applyNativeModulesAppBuildGradle(project)
```

Configuring permissions

Next, make sure you have the Internet permission in your `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET" />
```

If you need to access to the `DevSettingsActivity` add to your `AndroidManifest.xml`:

```
<activity android:name="com.facebook.react.devsupport.DevSettingsActivity" />
```

This is only used in dev mode when reloading JavaScript from the development server, so you can strip this in release builds if you need to.

Cleartext Traffic (API level 28+)

Starting with Android 9 (API level 28), cleartext traffic is disabled by default; this prevents your application from connecting to the [Metro bundler](#). The changes below allow cleartext traffic in debug builds.

1. Apply the `usesCleartextTraffic` option to your `Debug AndroidManifest.xml`

```
<!-- ... -->
<application
  android:usesCleartextTraffic="true" tools:targetApi="28" >
  <!-- ... -->
</application>
<!-- ... -->
```

This is not required for Release builds.

To learn more about Network Security Config and the cleartext traffic policy [see this link](#).

Code integration

Now we will actually modify the native Android application to integrate React Native.

The React Native component

The first bit of code we will write is the actual React Native code for the new "High Score" screen that will be integrated into our application.

1. Create a `index.js` file

First, create an empty `index.js` file in the root of your React Native project.

`index.js` is the starting point for React Native applications, and it is always required. It can be a small file that requires other files that are part of your React Native component or application, or it can contain all the code that is needed for it. In our case, we will put everything in `index.js`.

2. Add your React Native code

In your `index.js`, create your component. In our sample here, we will add a `<Text>` component within a styled `<View>`:

```

import React from 'react';
import {AppRegistry, StyleSheet, Text, View} from 'react-native';

const HelloWorld = () => {
  return (
    <View style={styles.container}>
      <Text style={styles.hello}>Hello, World</Text>
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
  },
  hello: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
});

AppRegistry.registerComponent(
  'MyReactNativeApp',
  () => HelloWorld,
);

```

3. Configure permissions for development error overlay

If your app is targeting the Android API level 23 or greater, make sure you have the permission `android.permission.SYSTEM_ALERT_WINDOW` enabled for the development build. You can check this with `Settings.canDrawOverlays(this)`. This is required in dev builds because React Native development errors must be displayed above all the other windows. Due to the new permissions system introduced in the API level 23 (Android M), the user needs to approve it. This can be achieved by adding the following code to your Activity's `onCreate()` method.

```

companion object {
  const val OVERLAY_PERMISSION_REQ_CODE = 1 // Choose any value
}

...

```

```

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    if(!Settings.canDrawOverlays(this)) {
        val intent = Intent(Settings.ACTION_MANAGE_OVERLAY_PERMISSION,
                           Uri.parse("package: $packageName"))
        startActivityForResult(intent, OVERLAY_PERMISSION_REQ_CODE);
    }
}

```

Finally, the `onActivityResult()` method (as shown in the code below) has to be overridden to handle the permission Accepted or Denied cases for consistent UX. Also, for integrating Native Modules which use `startActivityForResult`, we need to pass the result to the `onActivityResult` method of our `ReactInstanceManager` instance.

```

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    if (requestCode == OVERLAY_PERMISSION_REQ_CODE) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
            if (!Settings.canDrawOverlays(this)) {
                // SYSTEM_ALERT_WINDOW permission not granted
            }
        }
    }
    reactInstanceManager?.onActivityResult(this, requestCode, resultCode, data)
}

```

The Magic: ReactRootView

Let's add some native code in order to start the React Native runtime and tell it to render our JS component. To do this, we're going to create an `Activity` that creates a `ReactRootView`, starts a React application inside it and sets it as the main content view.

If you are targeting Android version <5, use the `AppCompatActivity` class from the `com.android.support:appcompat` package instead of `Activity`.

```

class MyReactActivity : Activity(), DefaultHardwareBackBtnHandler {
    private lateinit var reactRootView: ReactRootView
    private lateinit var reactInstanceManager: ReactInstanceManager
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }
}

```

```

SoLoader.init(this, false)
reactRootView = ReactRootView(this)
val packages: List<ReactPackage> = PackageList(application).packages
// Packages that cannot be autolinked yet can be added manually here, for
example:
// packages.add(MyReactNativePackage())
// Remember to include them in `settings.gradle` and `app/build.gradle` too.
reactInstanceManager = ReactInstanceManager.builder()
    .setApplication(application)
    .setCurrentActivity(this)
    .setBundleAssetName("index.android.bundle")
    .setJSMainModulePath("index")
    .addPackages(packages)
    .setUseDeveloperSupport(BuildConfig.DEBUG)
    .setInitialLifecycleState(LifecycleState.RESUMED)
    .build()
// The string here (e.g. "MyReactNativeApp") has to match
// the string in AppRegistry.registerComponent() in index.js
reactRootView?.startReactApplication(reactInstanceManager, "MyReactNativeApp",
null)
    setContentView(reactRootView)
}

override fun invokeDefaultOnBackPressed() {
    super.onBackPressed()
}
}

```

If you are using a starter kit for React Native, replace the "HelloWorld" string with the one in your index.js file (it's the first argument to the `AppRegistry.registerComponent()` method).

Perform a “Sync Project files with Gradle” operation.

If you are using Android Studio, use `Alt + Enter` to add all missing imports in your `MyReactActivity` class. Be careful to use your package's `BuildConfig` and not the one from the facebook package.

We need set the theme of `MyReactActivity` to `Theme.AppCompat.Light.NoActionBar` because some React Native UI components rely on this theme.


```
<activity
  android:name=".MyReactActivity"
  android:label="@string/app_name"
  android:theme="@style/Theme.AppCompat.Light.NoActionBar">
</activity>
```

A `ReactInstanceManager` can be shared by multiple activities and/or fragments. You will want to make your own `ReactFragment` or `ReactActivity` and have a singleton *holder* that holds a `ReactInstanceManager`. When you need the `ReactInstanceManager` (e.g., to hook up the `ReactInstanceManager` to the lifecycle of those Activities or Fragments) use the one provided by the singleton.

Next, we need to pass some activity lifecycle callbacks to the `ReactInstanceManager` and `ReactRootView`:

```
override fun onPause() {
    super.onPause()
    reactInstanceManager.onHostPause(this)
}

override fun onResume() {
    super.onResume()
    reactInstanceManager.onHostResume(this, this)
}

override fun onDestroy() {
    super.onDestroy()
    reactInstanceManager.onHostDestroy(this)
    reactRootView.unmountReactApplication()
}
```

We also need to pass back button events to React Native:

```
override fun onBackPressed() {
    reactInstanceManager.onBackPressed()
    super.onBackPressed()
}
```

This allows JavaScript to control what happens when the user presses the hardware back button (e.g. to implement navigation). When JavaScript doesn't handle the back button press, your `invokeDefaultOnBackPressed` method will be called. By default this finishes your Activity.

Finally, we need to hook up the dev menu. By default, this is activated by (rage) shaking the device, but this is not very useful in emulators. So we make it show when you press the hardware menu button (use `Ctrl` + `M` if you're using Android Studio emulator):

```
override fun onKeyDown(keyCode: Int, event: KeyEvent?): Boolean {  
    if (keyCode == KeyEvent.KEYCODE_MENU && reactInstanceManager != null) {  
        reactInstanceManager.showDevOptionsDialog()  
        return true  
    }  
    return super.onKeyDown(keyCode, event)  
}
```

Now your activity is ready to run some JavaScript code.

Test your integration

You have now done all the basic steps to integrate React Native with your current application. Now we will start the Metro bundler to build the `index.bundle` package and the server running on localhost to serve it.

1. Run the packager

To run your app, you need to first start the development server. To do this, run the following command in the root directory of your React Native project:

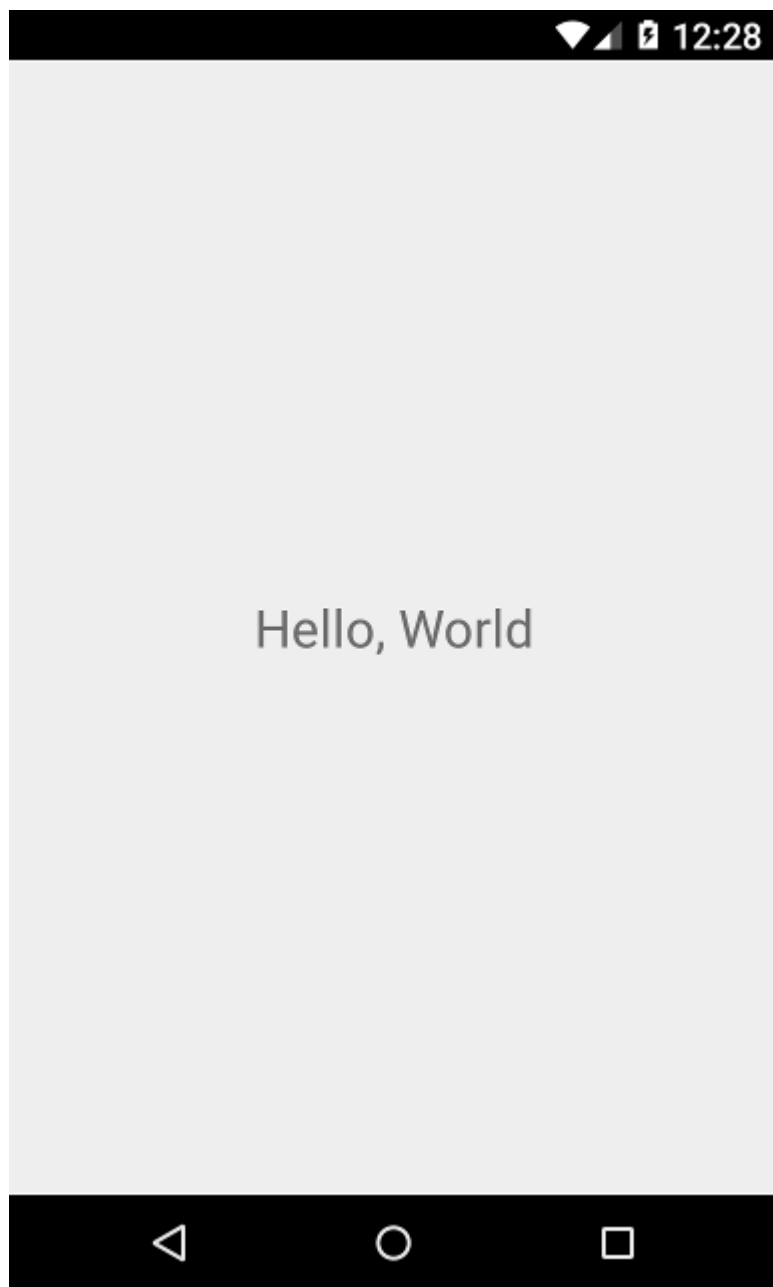
`npm` **`Yarn`**

`yarn start`

2. Run the app

Now build and run your Android app as normal.

Once you reach your React-powered activity inside the app, it should load the JavaScript code from the development server and display:



Creating a release build in Android Studio

You can use Android Studio to create your release builds too! It's as quick as creating release builds of your previously-existing native Android app.

If you use the React Native Gradle Plugin as described above, everything should work when running app from Android Studio.

If you're not using the React Native Gradle Plugin, there's one additional step which you'll have to do before every release build. You need to execute the following to create a React Native bundle, which will be included with your native Android app:

```
$ npx react-native bundle --platform android --dev false --entry-file index.js --bundle-output android/com/your-company-name/app-package-name/src/main/assets/index.android.bundle --assets-dest android/com/your-company-name/app-package-name/src/main/res/
```


Don't forget to replace the paths with correct ones and create the assets folder if it doesn't exist.

Now, create a release build of your native app from within Android Studio as usual and you should be good to go!

Now what?

At this point you can continue developing your app as usual. Refer to our [debugging](#) and [deployment](#) docs to learn more about working with React Native.

Is this page useful?  

 Edit this page

Last updated on **Jun 21, 2023**