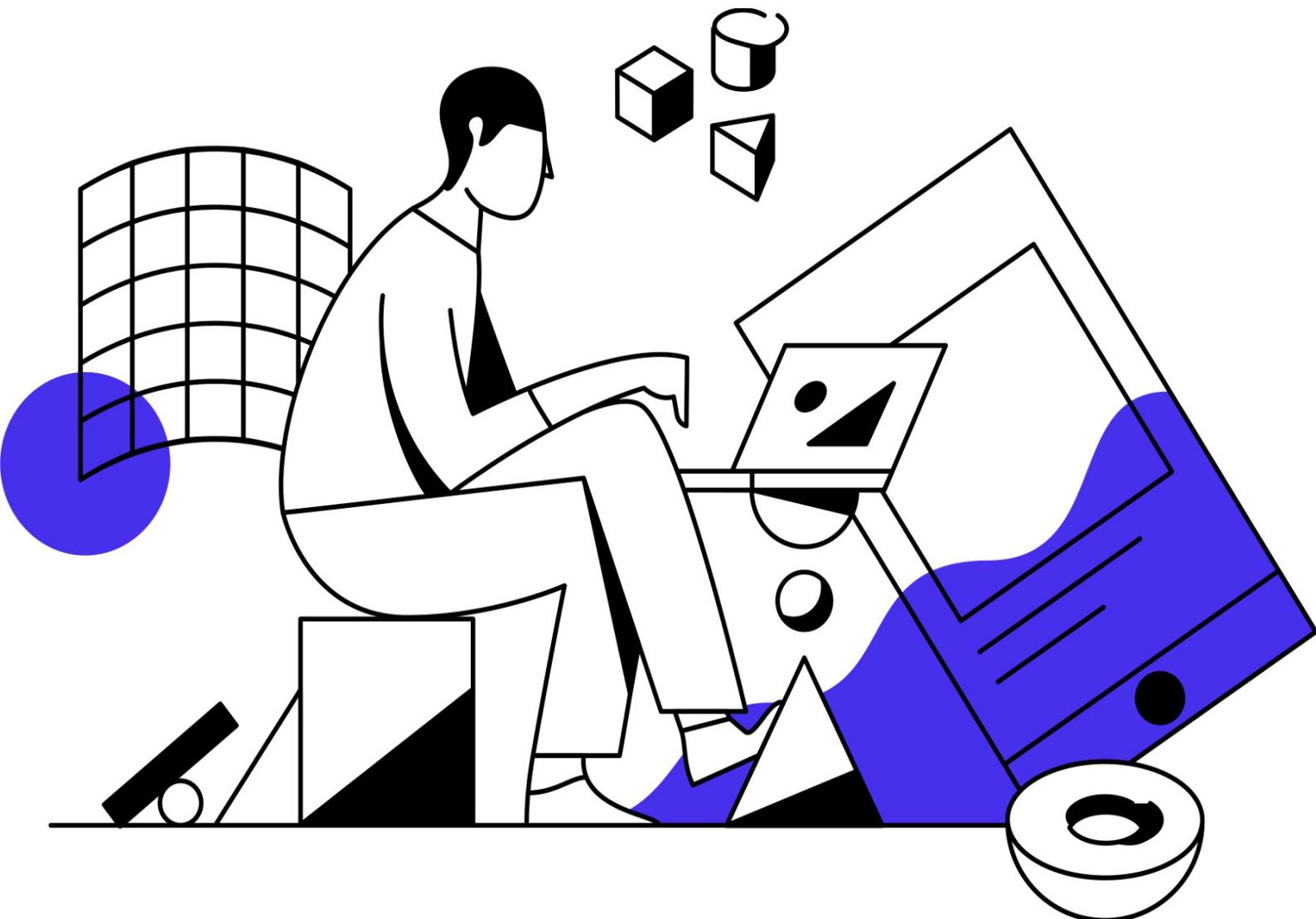


Expo SDK Documentation



v32.0.0

Table of Contents

Introduction	1.1
Get Started	1.2
Introduction	1.2.1
Getting to know Expo	1.2.1.1
Installation	1.2.1.2
Managed and Bare Workflows	1.2.1.3
Project Lifecycle	1.2.1.4
Community	1.2.1.5
Additional Resources	1.2.1.6
Troubleshooting Proxies	1.2.1.7
Frequently Asked Questions	1.2.1.8
Already used React Native?	1.2.1.9
Why not Expo?	1.2.1.10
Managed Workflow	1.3
Fundamentals	1.3.1
Up and Running	1.3.1.1
Expo CLI	1.3.1.2
Viewing Logs	1.3.1.3
Debugging	1.3.1.4
Development Mode	1.3.1.5
Android Studio Emulator	1.3.1.6
Configuration with app.json	1.3.1.7
Publishing	1.3.1.8
Upgrading Expo	1.3.1.9
Upgrading Expo SDK Walkthrough	1.3.1.10
Linking	1.3.1.11
How Expo Works	1.3.1.12
Glossary of terms	1.3.1.13
Expo & "Create React Native App"	1.3.1.14
Guides	1.3.2
App Icons	1.3.2.1
Assets	1.3.2.2
Error Handling	1.3.2.3
Preloading & Caching Assets	1.3.2.4
Icons	1.3.2.5

Using Custom Fonts	1.3.2.6
Routing & Navigation	1.3.2.7
Configuring StatusBar	1.3.2.8
Offline Support	1.3.2.9
Configuring OTA Updates	1.3.2.10
Push Notifications	1.3.2.11
Using FCM for Push Notifications	1.3.2.12
Notification Channels	1.3.2.13
Using ClojureScript	1.3.2.14
Using GraphQL	1.3.2.15
Using Sentry	1.3.2.16
Create a Splash Screen	1.3.2.17
Setting up Continuous Integration	1.3.2.18
Testing on physical devices	1.3.2.19
Using Firebase	1.3.2.20
Distributing Your App	1.3.3
Overview	1.3.3.1
Building Standalone Apps	1.3.3.2
App signing	1.3.3.3
Deploying to App Stores	1.3.3.4
Release Channels	1.3.3.5
Advanced Release Channels	1.3.3.6
Hosting An App on Your Servers	1.3.3.7
Building Standalone Apps on Your CI	1.3.3.8
Uploading Apps to the Apple App Store and Google Play	1.3.3.9
App Transfers	1.3.3.10
ExpoKit	1.3.4
Overview	1.3.4.1
Ejecting to ExpoKit	1.3.4.2
Developing With ExpoKit	1.3.4.3
Advanced ExpoKit Topics	1.3.4.4
Universal Modules and ExpoKit	1.3.4.5
Bare Workflow	1.4
Essentials	1.4.1
Hello World	1.4.1.1
API Reference	1.5
Expo SDK	1.5.1
Overview	1.5.1.1
Overview	1.5.1.2

ImageManipulator	1.5.1.3
AppAuth	1.5.1.4
AppLoading	1.5.1.5
AR	1.5.1.6
ART	1.5.1.7
Asset	1.5.1.8
Audio	1.5.1.9
AuthSession	1.5.1.10
AV	1.5.1.11
BackgroundFetch	1.5.1.12
BarCodeScanner	1.5.1.13
Barometer	1.5.1.14
BlurView	1.5.1.15
Branch	1.5.1.16
Brightness	1.5.1.17
Calendar	1.5.1.18
Camera	1.5.1.19
Constants	1.5.1.20
Contacts	1.5.1.21
DeviceMotion	1.5.1.22
DocumentPicker	1.5.1.23
ErrorRecovery	1.5.1.24
Facebook	1.5.1.25
FacebookAds	1.5.1.26
FaceDetector	1.5.1.27
FileSystem	1.5.1.28
Font	1.5.1.29
GestureHandler	1.5.1.30
GLView	1.5.1.31
Google	1.5.1.32
GoogleSignIn	1.5.1.33
Gyroscope	1.5.1.34
Haptic	1.5.1.35
Haptics	1.5.1.36
Admob	1.5.1.37
ImagePicker	1.5.1.38
IntentLauncherAndroid	1.5.1.39
Accelerometer	1.5.1.40

KeepAwake	1.5.1.41
LinearGradient	1.5.1.42
Linking	1.5.1.43
LocalAuthentication	1.5.1.44
Localization	1.5.1.45
Location	1.5.1.46
Lottie	1.5.1.47
Magnetometer	1.5.1.48
MailComposer	1.5.1.49
MapView	1.5.1.50
MediaLibrary	1.5.1.51
Notifications	1.5.1.52
Amplitude	1.5.1.53
Payments	1.5.1.54
Pedometer	1.5.1.55
Permissions	1.5.1.56
Print	1.5.1.57
registerRootComponent	1.5.1.58
ScreenOrientation	1.5.1.59
SecureStore	1.5.1.60
Segment	1.5.1.61
Sensors	1.5.1.62
SMS	1.5.1.63
Speech	1.5.1.64
SplashScreen	1.5.1.65
SQLite	1.5.1.66
StoreReview	1.5.1.67
Svg	1.5.1.68
takeSnapshotAsync	1.5.1.69
TaskManager	1.5.1.70
Updates	1.5.1.71
Video	1.5.1.72
WebBrowser	1.5.1.73
React Native	1.5.2
Learn the Basics	1.5.2.1
Props	1.5.2.2
State	1.5.2.3
Style	1.5.2.4
Height and Width	1.5.2.5

Layout with Flexbox	1.5.2.6
Handling Text Input	1.5.2.7
Handling Touches	1.5.2.8
Using a ScrollView	1.5.2.9
Using List Views	1.5.2.10
Networking	1.5.2.11
Platform Specific Code	1.5.2.12
Navigating Between Screens	1.5.2.13
Images	1.5.2.14
Animations	1.5.2.15
Accessibility	1.5.2.16
Timers	1.5.2.17
Performance	1.5.2.18
Gesture Responder System	1.5.2.19
JavaScript Environment	1.5.2.20
Direct Manipulation	1.5.2.21
Color Reference	1.5.2.22
ActivityIndicator	1.5.2.23
Button	1.5.2.24
DatePickerIOS	1.5.2.25
DrawerLayoutAndroid	1.5.2.26
FlatList	1.5.2.27
Image	1.5.2.28
InputAccessoryView	1.5.2.29
KeyboardAvoidingView	1.5.2.30
ListView	1.5.2.31
MaskedViewIOS	1.5.2.32
Modal	1.5.2.33
NavigatorIOS	1.5.2.34
Picker	1.5.2.35
PickerIOS	1.5.2.36
ProgressBarAndroid	1.5.2.37
ProgressViewIOS	1.5.2.38
RefreshControl	1.5.2.39
SafeAreaView	1.5.2.40
ScrollView	1.5.2.41
SectionList	1.5.2.42
SegmentedControlIOS	1.5.2.43

Slider	1.5.2.44
SnapshotViewIOS	1.5.2.45
StatusBar	1.5.2.46
Switch	1.5.2.47
TabBarIOS.Item	1.5.2.48
TabBarIOS	1.5.2.49
Text	1.5.2.50
TextInput	1.5.2.51
ToolbarAndroid	1.5.2.52
TouchableHighlight	1.5.2.53
TouchableNativeFeedback	1.5.2.54
TouchableOpacity	1.5.2.55
TouchableWithoutFeedback	1.5.2.56
View	1.5.2.57
ViewPagerAndroid	1.5.2.58
VirtualizedList	1.5.2.59
WebView	1.5.2.60
AccessibilityInfo	1.5.2.61
ActionSheetIOS	1.5.2.62
Alert	1.5.2.63
AlertIOS	1.5.2.64
Animated	1.5.2.65
AppState	1.5.2.66
AsyncStorage	1.5.2.67
BackAndroid	1.5.2.68
BackHandler	1.5.2.69
Clipboard	1.5.2.70
DatePickerAndroid	1.5.2.71
Dimensions	1.5.2.72
Easing	1.5.2.73
Image Style Props	1.5.2.74
ImageStore	1.5.2.75
InteractionManager	1.5.2.76
Keyboard	1.5.2.77
Layout Props	1.5.2.78
LayoutAnimation	1.5.2.79
ListViewDataSource	1.5.2.80
NetInfo	1.5.2.81
PanResponder	1.5.2.82

PixelRatio	1.5.2.83
Settings	1.5.2.84
Shadow Props	1.5.2.85
Share	1.5.2.86
StatusBarIOS	1.5.2.87
StyleSheet	1.5.2.88
Systrace	1.5.2.89
Text Style Props	1.5.2.90
TimePickerAndroid	1.5.2.91
ToastAndroid	1.5.2.92
Transforms	1.5.2.93
Vibration	1.5.2.94
VibrationIOS	1.5.2.95
View Style Props	1.5.2.96
ImageBackground	1.5.2.97
Native Modules Setup	1.5.2.98
Out-of-Tree Platforms	1.5.2.99

Official Documentation

- [HTML on the web](#)
- [Markdown on GitHub](#)

Unofficial Ebooks

- [conversion scripts](#)
 - tools:
 - [GitBook](#)
 - [calibre](#)
 - [GitHub Downloader](#)
 - [node](#)
 - [perl](#)
- [releases: pdf, epub, mobi](#)

Getting to know Expo

Introduction

This is the documentation for [Expo](#). Expo is a set of tools, libraries and services you can use to build native iOS and Android apps faster than ever before.

There are two ways to build a project with Expo, we call these workflows: you can use the "managed" workflow or the "bare" workflow. With the "managed" workflow, you only write JavaScript and lean on the [Expo SDK](#) to give you access to your device capabilities and the Expo services to handle the heavy lifting of building your app binary and uploading it to the store, all without you touching Xcode or Android Studio. With the "bare" workflow, we also speed up your development with the [Expo SDK](#) and React Native, and you have full control over your iOS and Android projects.

More about the Expo SDK

The Expo SDK is a set of libraries written natively for each platform which provides access to the device's system functionality (things like the camera, push notifications, contacts, local storage, and other hardware and operating system APIs) from JavaScript. The SDK is designed to smooth out differences in platforms as much as possible, which makes your project very portable because it can run in any native environment containing the Expo SDK.

Expo also provides UI components to handle a variety of use-cases that almost all apps will cover but are not built into React Native core, e.g. icons, blur views, and more.

Considering using Expo?

- If you'd like an overview of what Expo offers, you might want to familiarize yourself with the [lifecycle of an Expo project](#), which describes how you go from square one to a production iOS and Android app.
- For further explanation, it's also good to check out the [Frequently Asked Questions](#).

Ready to get started?

- Head over to [Installation](#) to grab our tools and have a look around.
- Make your first project by following the [Up and Running](#) guide.
- If you're not already familiar with React and React Native, you can bootstrap your knowledge with [React Native Express](#).
- For hands-on React Native projects from beginner to advanced, check out [Fullstack React Native](#), a (paid) book by the author of React Native Express.
- Join our [Community](#) and let us know what you're working on!

Installation

There are two tools that you need to develop apps with Expo: a local development tool and a mobile client to open your app.

Local Development Tool: Expo CLI

Expo CLI is a tool for developing apps with Expo. In addition the command-line interface (CLI) it also has a graphical UI, Expo Developer Tools, that pops up in your web browser. With Expo Dev Tools you can quickly set up your test devices, view logs and more.

You'll need to have Node.js (version 10 or newer) installed on your computer. [Download the latest version of Node.js](#). Additionally, you'll need Git to create new projects with Expo CLI. [You can download Git from here](#).

You can install Expo CLI by running:

```
npm install -g expo-cli
```

Mobile Client: Expo for iOS and Android

Expo Client helps view your projects while you're developing them. When you serve your project with Expo CLI, it generates a development URL that you can open in Expo Client to preview your app. On Android, Expo Client can also be used to view others' projects on [expo.io](#). Expo Client works on devices, simulators, and emulators.

On your device

[Download for Android from the Play Store](#) or for iOS from the [App Store](#)

Required Android and iOS versions: The minimum Android version Expo supports is Android 5 and the minimum iOS version is iOS 10.0.

You don't need to manually install the Expo client on your emulator/simulator, because Expo CLI will do that automatically. See the next sections of this guide.

iOS simulator

Install [Xcode through the Apple App Store](#). It'll take a while, go have a nap. Next, open up Xcode, go to preferences and click the Components tab, install a simulator from the list.

Once the simulator is open and you have a project open in Expo Dev Tools, you can press *Run on iOS simulator* in Expo Dev Tools and it will install the Expo Client to the simulator and open up your app inside of it.

Not working? Occasionally Expo CLI will have trouble installing the Expo Client automatically, usually due to annoying small differences in your environment or Xcode toolchain. If you need to install the Expo Client on your simulator manually, you can follow these steps:

- Download the [latest simulator build](#).
- Extract the contents of the archive: `mkdir Exponent-X.XX.X.app && tar xvf Exponent-X.XX.X.tar.gz -C Exponent-X.XX.X.app`. You should get a directory like `Exponent-X.XX.X.app`.

- Make sure Simulator is running.
- At a terminal, run `xcrun simctl install booted [path to Exponent-X.XX.X.app]`.

Android emulator

Follow our [Android Studio emulator guide](#) to set up Android tools and create a virtual device. Start up the virtual device when it's ready.

Once the emulator is open and you have a project open in Expo Dev Tools, you can press *Run on Android device/emulator* in Expo Dev Tools and it will install the Expo client to the emulator and open up your app inside of it.

Watchman

Some macOS users encounter issues if they do not have this installed on their machine, so we recommend that you install Watchman. Watchman watches files and records when they change, then triggers actions in response to this, and it's used internally by React Native. [Download and install Watchman](#).

Managed and Bare Workflows

The two ways to use Expo tools are called the "managed" and "bare" workflows.

Managed workflow

Apps are built with the managed workflow using the [expo-cli](#), the Expo client on your mobile device, and our various services: [push notifications](#), the [build service](#), and [over-the-air \(OTA\) updates](#). Expo tries to manage as much of the complexity of building apps for you as we can, so we call it the managed workflow. A developer using the managed workflow doesn't use Xcode or Android Studio, they just write JavaScript code and managed configuration through [app.json](#). There are tradeoffs that you should consider when building your app this way, check out [Why not Expo?](#) to learn more.

Bare workflow

Bare apps give the developer complete control, along with the complexity that comes with it. You can use most APIs in the Expo SDK, you will just need to install and configure them manually rather than having them ready for you out-of-the-box. Most of the Expo documentation will not apply to building your app if you use this workflow, instead you can refer to tutorials and guides that are oriented towards native iOS and Android apps and React Native.

Project Lifecycle

Expo makes it easy to get started writing apps and to take them all the way to production using the managed workflow. Here's an overview of the tools and services you might use along the way.

This guide is meant to give a high-level explanation of what Expo offers. For the curious, the technical implementation of these topics is covered in much more detail [here](#).



Creating an Expo project

You can create a new Expo project with only our desktop tool and a text editor. See [Up and Running](#) for a fast guide to creating a project, running it on a device, and making changes.

Expo apps are React Native apps with the Expo SDK built-in. The fastest way to get started is using the [Up and Running](#) guide, but you can also [convert an existing React Native app](#) or adopt only bits and pieces of Expo into your app.

Developing locally

When you work on an Expo project, we serve an instance of your project from your local computer. If you close the project or turn off your computer, your development project stops being served.

During this time, you test your project using a pre-built iOS/Android app called [Expo Client](#). It asks your computer for a local copy of your project (via localhost, LAN, or a tunnel), downloads it, and runs it. You can take advantage of various development tools such as [debugging](#), [streaming device logs](#), inspecting elements, hot module reloading, and more.

Publishing your project

If you click the **Publish** button in Expo Dev Tools, we upload a minified copy of your app to our CDN, and give you a shareable url of the form `expo.io/@your-username/your-app-slug`.

You can share this link immediately with anybody who has the Expo Client app for Android. [Read more about Publishing here](#).

On iOS, you'll need to use Apple TestFlight to share your app with others.

Updating your app

You can continue making changes locally without disrupting your users. Any time you **Publish** changes to your app, your new version becomes available immediately to anybody with the link.

We frequently release updates to the [Expo SDK](#). Each update includes instructions for how to upgrade your project. If you decide to update to a newer version of our SDK, copies of the older version will continue to work fine. Users will download the newest copy that their client supports.

Deploying to the Apple App Store and Google Play

When you're ready to list your app officially on the Apple App Store and Google Play Store, Expo can generate deployment-ready `.ipa` and `.apk` archives which are ready to submit to Apple and Google. We generate them on our servers, so you still don't need any Apple or Google software. See the documentation about [Distributing Apps](#).

Changing native code

You can take your app all the way to the App Store and Play Store while writing only JS using the managed workflow. However, if you run into needs which aren't met by the Expo SDK (see "[Why not Expo?](#)" to help anticipate whether you will encounter this), we provide the ability to eject, which gives you the native Xcode and Android Studio representation of your project so you can change anything that you need to.

Note: If you choose to eject, some Expo services are no longer available. For example, `expo-cli` won't work, we can't generate standalone builds for you anymore, and you won't be able to publish updates using `expo publish`. Your project becomes a normal React Native project with most of the Expo SDK APIs included.

Community

Want to chat about Expo? The best way to get in touch with our team and other people developing with Expo is to join our Slack chat. We always like to hear about projects and components that people are building on Expo and there's usually someone available to answer questions.

- [Join our forums](#)
- [Request an invitation to our Slack.](#)
- [Follow us on Twitter @expo](#)
- [Become a contributor on Github](#)
- [Follow Expo Community on Hashnode](#)

Additional Resources

The following resources are useful for learning Expo and some of the projects that it depends on.

Expo Blog

- [Exposition](#) - our official blog, where we post release notes every month and other Expo related content at random intervals.

Courses using Expo

- [Harvard CS50 Mobile](#) (free)
- [Stanford CS 47SI Cross-Platform Mobile Development](#) (free)
- [repl.it - React Native - Build your first app in the next 5 minutes](#) (free)
- [React Europe - Introduction to React Native Workshop videos on YouTube](#) (free)
- [Udemy - React Native: Advanced Concepts by Stephen Grider](#) (paid)
- [Udacity - React Nanodegree](#) (paid)

React Native

- [React Native Express](#) - The best way to get started with React Native! It's a walkthrough the building blocks of React and React Native.
- [Fullstack React Native](#) (book) - From the author of React Native Express: build 7 complete apps, covering React Native fundamentals and advanced topics in-depth.
- [Official React Native Documentation](#)
- [React Native Fundamentals](#) (video course on Egghead.io)
- [Animating React Native UI Elements](#) (video course on Egghead.io)
- [Learning React Native, 2nd Edition](#) (book)

Talks

- [React Europe - Expo Snack](#)
- [React Conf - Create React Native App](#)
- [Reactive - From React web to native mobile](#)
- [React Europe - Building li.st for Android with Expo and React Native](#)
- [CS50 at Harvard University - Easily Build Apps with React Native](#)
- [Apollo Day - GraphQL at Expo](#)

React

- [React Express](#)
- [React lessons on Egghead.io](#) - several playlists that cover React and Redux fundamentals.
- [Official React Documentation](#)

JavaScript

- [ES6 Katas](#) - exercises to familiarize yourself with new JavaScript features used in React Native

Troubleshooting Proxies

Mac OS Proxy Configuration (Sierra)

If anything goes wrong, you can revert back to the "Automatic Proxy settings" in System Network Preferences using Automatic Proxy Configuration `your-corporate-proxy-uri:port-number/proxy.pac`

Overview

In order to run this in the local iOS Simulator while on your corporate wi-fi network, a local proxy manager is required. This local proxy application, named [Charles](#), is what our iOS Dev Team uses. Meet Charles, get to know him. He is your friend.

Open Mac OS Network Preferences

1. Open `System Preferences` for your Mac (Apple Menu > System Preferences).
2. Go to Network.
3. Be sure your `Location` is set to your proxy network, and not "Automatic".
4. With Wi-Fi selected on the left and/or ethernet connection, click `Advanced...` on the bottom right side of the window.

Configure Proxy Address

1. Disable/uncheck "Automatic Proxy Configuration" if it is set.
2. Check "Web Proxy (HTTP)" and set "Web Proxy Server" to 127.0.0.1 : 8888
3. Check "Secure Web Proxy (HTTPS)" and set "Secure Web Proxy Server" to 127.0.0.1 : 8888

Configure Charles

1. Open Charles
2. If it asks, don't allow it to manage your Mac OS Network Configuration, the previous steps do that. (If you change Charles port, update the previous step to the correct port instead of default 8888)
3. In the menu of Charles go to `Proxy > External Proxy Settings`, check `Use external proxy servers`
4. Check `Web Proxy (HTTP)`, and enter `your-corporate-proxy-uri:port-number`
5. Check `Proxy server requires a password`
6. Domain: YOUR DOMAIN, Username: YOUR USERNAME Password: YOUR PASSWORD
7. Same for Secure Web Proxy (HTTPS). *Be sure to fill in the same proxy, username, and password address fields.*
8. In the text area for `Bypass external proxies for the following hosts:` enter

```
localhost  
*.local
```

You may need to include your mail server, or other corporate network addresses.

1. Check "Always bypass external proxies for localhost"

iOS Simulator Configuration

If you have an existing iOS Simulator custom setup going that is not working, "Simulator > Reset Content and Settings" from the menu.

If you have the Simulator open still, quit it.

Now, in Charles under the "Help" menu > Install Charles Root Certificate, and then again for Install Charles Root Certificate in iOS Simulators

Technical note: This whole process is required because the iOS Simulator is served a bum proxy certificate instead of the actual certificate, and doesn't allow it, for <https://exp.host/> which is required to run Expo.

Also note: Configure applications that need internet access, such as Spotify, to use <http://localhost:8888> as your proxy. Some apps, such as Chrome and Firefox, you can configure in the settings to use your "System Network Preferences" which will use Charles : 8888, or no proxy, depending on how you have your "Location" set in the Apple menu / network preferences. If you are set to "Automatic" no proxy is used, if it is set to "your proxy network" the proxy is used and Charles will need to be running.

Command line application proxy configuration

npm, git, Brew, Curl, and any other command line applications need proxy access too.

For npm

Open `~/.npmrc` and set:

```
http_proxy=http://localhost:8888  
https_proxy=http://localhost:8888
```

For git

Open `~/.gitconfig` and set

```
[http]  
proxy = http://localhost:8888  
[https]  
proxy = http://localhost:8888
```

For Command line applications

Depending on your shell, and config, Open `~/.bashrc`, `~/.bash_profile`, or `~/.zshrc` or wherever you set your shell variables, and set:

```
export HTTP_PROXY="http://localhost:8888"  
export http_proxy="http://localhost:8888"  
export ALL_PROXY="http://localhost:8888"  
export all_proxy="http://localhost:8888"  
export HTTPS_PROXY="http://localhost:8888"  
export https_proxy="http://localhost:8888"
```

Note: if you switch your network location back to "Automatic" in order to use npm or git, you will need to comment these lines out using a `#` before the line you wish to disable. You could alternatively use a command-line proxy manager if you prefer.

Frequently Asked Questions

In addition to the questions below, see the [Expo Forum](#) or [Expo AMA on Hashnode](#) for more common questions and answers.

How much does Expo cost?

Expo is free.

Our plan is to keep it this way indefinitely.

Expo is also open source, so you don't have to trust us to stick to that plan.

We might eventually charge money for services built on top of Expo or for some kind of premium level of support and consulting.

How do you make money if Expo is free?

Right now, we don't make any money. We have a small team that just wants to work on this, and we keep our expenses low and are mostly self-funded. We can keep working on this project like this for a while as long as there are people who want to use it.

We think if we can make Expo good enough, we can eventually help developers make money, and we could take a cut of that. This could be through helping them collect money from people using their software, or by helping them place ads in their apps, or other things. We don't really have a clear plan for this yet; our first priority is just to make Expo work really well and be as popular as possible.

What is the difference between Expo and React Native?

Expo is kind of like Rails for React Native. Lots of things are set up for you, so it's quicker to get started and on the right path.

With Expo, you don't need Xcode or Android Studio. You just write JavaScript using whatever text editor you are comfortable with (Atom, vim, emacs, Sublime, VS Code, whatever you like). You can run Expo CLI (our command line tool and web UI) on Mac, Windows, and Linux.

Here are some of the things Expo gives you out of the box that work right away:

- **Support for iOS and Android**

You can use apps written in Expo on both iOS and Android right out of the box. You don't need to go through a separate build process for each one. Just open any Expo app in the Expo Client app from the App Store on either iOS or Android (or in a simulator or emulator on your computer).

- **Push Notifications**

Push notifications work right out of the box across both iOS and Android, using a single, unified API. You don't have to set up APNS and GCM/FCM or configure ZeroPush or anything like that. We think we've made this as easy as it can be right now.

- **Facebook Login**

This can take a long time to get set up properly yourself, but you should be able to get it working in 10 minutes or less on Expo.

- **Instant Updating**

All Expo apps can be updated in seconds by just clicking Publish in Expo Dev Tools. You don't have to set anything up; it just works this way. If you aren't using Expo, you'd either use Microsoft Code Push or roll your own solution for this problem.

- **Asset Management**

Images, videos, fonts, etc. are all distributed dynamically over the Internet with Expo. This means they work with instant updating and can be changed on the fly. The asset management system built-in to Expo takes care of uploading all the assets in your repo to a CDN so they'll load quickly for anyone.

Without Expo, the normal thing to do is to bundle your assets into your app which means you can't change them. Or you'd have to manage putting your assets on a CDN or similar yourself.

- **Easier Updating To New React Native Releases**

We do new releases of Expo every few weeks. You can stay on an old version of React Native if you like, or upgrade to a new one, without worrying about rebuilding your app binary. You can worry about upgrading the JavaScript on your own time.

But no native modules...

The most limiting thing about Expo is that you can't add in your own native modules without `detach`ing and using ExpoKit. Continue reading the next question for a full explanation.

How do I add custom native code to my Expo project?

TL;DR you can do it, but most people never need to.

Standard Expo projects don't support custom native code, including third-party libraries which require custom native components. In an Expo project, you only write pure JS. Expo is designed this way on purpose and we think it's better this way.

In [our SDK](#), we give you a large set of commonly desired, high-quality native modules. We recommend doing as much in JS as possible, since it can immediately deploy to all your users and work across both platforms, and will always continue to benefit from Expo SDK updates. Especially in the case of UI components, there is pretty much always a better option written in JS.

However, if you need something very custom--like on-the-fly video processing or low level control over the Bluetooth radio to do a firmware update--we do have early/alpha support for [using Expo in native Xcode and Android Studio projects](#).

Is Expo similar to React for web development?

Expo and React Native are similar to React. You'll have to learn a new set of components (`View` instead of `div`, for example) and writing mobile apps is very different from websites; you think more in terms of screens and different navigators instead of separate web pages, but much more of your knowledge carries over than if you were writing a traditional Android or iOS app.

How do I share my Expo project? Can I submit it to the app stores?

The fastest way to share your Expo project is to publish it. You can do this by clicking 'Publish' in Expo Dev Tools or running `expo publish` in your project. This gives your app a URL; you can share this URL with anybody who has the Expo Client for Android and they can open your app immediately. [Read more about publishing on Expo](#). To share with iOS users, you can use Apple TestFlight.

When you're ready, you can create a standalone app (`.ipa` and `.apk`) for submission to Apple and Google's app stores. Expo will build the binary for you when you run one command; see [Building Standalone Apps](#). Apple charges \$99/year to publish your app in the App Store and Google charges a \$25 one-time fee for the Play Store.

Why does Expo use a fork of React Native?

Each Expo SDK Version corresponds to a React Native release. For example, SDK 19 corresponds to React Native 0.46.1. Often there is no difference between the fork for a given a SDK version and its corresponding React Native version, but occasionally we will find issues or want to include some code that hasn't yet been merged into the release and we will put it in our fork. Using a fork also makes it easier for people to verify that they are using the correct version of React Native for the Expo SDK version -- you know that if your SDK Version is set to 19.0.0 then you should use <https://github.com/expo/react-native/archive/sdk-19.0.0.tar.gz>.

How do I get my existing React Native project running with Expo?

Right now, the easiest way to do this is to use `expo init` (with Expo CLI) to make a new project, and then copy over all your JavaScript source code from your existing project, and then `yarn add` ing the library dependencies you have.

If you have similar native module dependencies to what is exposed through the Expo SDK, this process shouldn't take more than a few minutes (not including `npm install` time). Please feel free to ask us questions if you run into any issues.

If you are using native libraries that aren't supported by Expo, you will either have to rewrite some parts of your application to use similar APIs that are part of Expo, or you just might not be able to get all parts of your app to work. Many things do though!

N.B. We used to maintain a tool `exp convert` but it is not currently working or maintained so the above method is the best way to get an existing React Native project working on Expo

How do I remove an Expo project that I published?

The default [privacy setting](#) for Expo apps is `unlisted` so nobody can find your app unless you share the link with them.

If you really want your published app to be 'unpublished', check out our guide on [Advanced Release Channels](#), which explains how to roll back.

What is Exponent and how is it different from Expo?

Exponent is the original name of the Expo project. You might occasionally run across some old references to it in blog posts or code or documentation. They are the same thing; we just shortened the name.

What version of Android and iOS are supported by Expo apps?

Expo supports Android 5+ and iOS 10+.

Can I use Node.js packages with Expo?

If the package depends on [Node standard library APIs](#), you will not be able to use it with Expo. The Node standard library is a set of functionality implemented largely in C++ that exposes functions to JavaScript that aren't part of the JavaScript language specification, such as the ability to read and write to your filesystem. React Native, and by extension Expo, do not include the Node standard library, just like Chrome and Firefox do not include it. JavaScript is a language that is used in many contexts, from mobile apps (in our case), to servers, and of course on websites. These contexts all include their own runtime environments that expose different APIs to JavaScript, depending on what makes sense in the context.

As a side note, some Node standard library APIs do not depend on C++ extensions but instead can be implemented directly in JavaScript, such as `url` and `assert`. If a package you wish to use only depends on these Node APIs, you can install them from npm and the package will work.

Can I use Expo with Relay?

You can! Update your `.babelrc` you get on a new Expo project to the following:

```
{
  "presets": [
    "babel-preset-expo",
    {"plugins": ["./pathToYourBabelRelayPlugin/babelRelayPlugin"]}
  ],
  "env": {
    "development": {
      "plugins": ["transform-react-jsx-source"]
    }
  }
};
```

Substitute `./pathToYourBabelRelayPlugin` with the path to your Relay plugin.

How do I handle expired push notification credentials?

When your push notification credentials have expired, simply run `expo build:ios -c --no-publish` to clear your expired credentials and generate new ones. The new credentials will take effect within a few minutes of being generated. You do not have to submit a new build!

Already used React Native?

This guide is intended to give developers who have already used React Native a quick outline on some of the key concepts, resources, and differences they will encounter when using Expo.

What is Expo?

Expo provides a *shared native runtime* so you don't write native code, you focus on writing your React app in JavaScript. You don't have to worry about iOS or Android specific settings, or even opening up Xcode. Expo has its own workflow including Expo CLI (a command line interface) and Expo Dev Tools (a web UI) to make developing and deploying easy.

- If you've ever upgraded React Native or a native module you'll appreciate Expo's ability to seamlessly do this for you by only changing the version number.

Expo extends the React Native platform by offering additional, battle-tested modules that are maintained by the team. This means you're spending less time configuring and more time building.

- If you've ever had to go through the trouble of upgrading a module or installing something like `react-native-maps`, you'll appreciate when things *just work*.

Expo also offers OTA (Over The Air) updates and a push notification service.

- If you've ever been in a situation where you find a spelling mistake in your app and have to wait for Apple to approve a change, you'll appreciate OTA updates - these changes will appear as soon as you run `expo publish`! You aren't limited to text either, this applies to assets like images and configuration updates too!

There's no need re-build or redeploy your app to the App and Play store. It's like [Code Push](#) if you've used that before. There are a few limitations, however. [Read about those here](#).

Expo offers a shared configuration file we call a manifest. Typically you'd update your Xcode plist or Android Studio xml files to handle changes. For example, if you want to lock screen orientation, change your icon, customize your splash screen or add/remove permissions you'd do this within `app.json` once and it would apply to both.

- Configuration that you would typically do inside of your Xcode / plist files or Android studio / xml files is handled through `app.json`. For example, if you want to lock the orientation, change your icon, customize your splash screen, add/remove permissions and entitlements (in standalone apps), configure keys for Google Maps and other services, you set this in `app.json`. [See the guide here](#).

With Expo, you can share your app with anyone, anywhere in the world while you're working through the Expo client ([available on the App / Play Store](#)). Scan a QR code, or enter in a phone number and we'll send you a link that will instantly load your app on your device.

- Instead of having to sign up several external testers through iTunes connect, you can easily have them download the Expo client app and immediately have a working version on their phone.

We talk about permissions we set within `app.json`, but there's also the [Permissions API](#). Permissions inside `app.json` are meant to be used by Android standalone apps for things like camera access, geolocation, fingerprint, etc. The Permissions API on the other hand, is used to request and verify access at runtime. It offers an easy API for asking your users for push notifications, location, camera, audio recording and contacts.

How does Expo work?

Since you write your code in Javascript, we bundle it up and serve it from S3. Every time you publish your app, we update those assets and then push them to your app so you've always got an up-to-date version.

Developing in Expo

Apps are served from Expo CLI through a tunnel service by default (we currently use [ngrok](#) for this) -- this means that you don't have to have your device connected to your computer, or to even be in the same room or country (or planet? I guess it should work from space) as the development machine and you can still live reload, use hot module reloading, enable remote JS debugging, and all of those things you can do normally with React Native. One caveat to this is that using a tunnel is a bit slower than using your LAN address or localhost, so if you can, you should use LAN or localhost. [See how to configure this in Expo CLI](#).

- Expo streams your device logs to Expo CLI and Expo Dev Tools so you don't need to run `adb logcat` or the iOS equivalent -- the `console.log / warn /error` messages from any device that is connected to your app will show up automatically in your terminal and Expo Dev Tools.

What Expo can't do

- See [Why not Expo?](#) for more information.

Deploying to the App / Play Store

When you're ready, you can run `expo build:ios` or `expo build:android` and Expo will build your app and output a link to the binary required for you to submit. Then you can use something like Application Loader for iOS, or directly upload an APK for Android.

If you prefer to build your app on your own machine, you can [follow these steps](#).

Helpful Tools & Resources

- [snack.expo.io](#)
 - The best way to test and share examples and small projects directly from your browser. Point your phone at the QR code and you have a sandbox environment you can build in the browser and test directly on your device.
- [docs.expo.io](#)
 - If there's something you don't understand or wish to learn more about, this is a great place to start.
- [forums.expo.io](#)
 - The fastest way to get help from the Expo team or community
- [github.com/expo](#)
 - The Expo Client and SDK are all open source. If there's something you'd like to fix, or figure out how we implement our native modules, you're welcome to look through the code yourself!
- [Slack](#)
 - Chat with other folks who use Expo.

Useful Commands

When developing in Expo, you have the option to use command line tools instead. Here are some of our friends' favorite commands and workflows:

- `expo start -c --localhost --ios`
 - start expo server, clear cache, load only on localhost and open on iOS simulator
- `expo start --tunnel`
 - start expo server (don't clear cache) and run expo on a tunnel so you can share it with anyone!
- `expo send -s 2245551234`
 - send a link to a friend's phone number so they can view on their phone exactly what I'm working on

Why not Expo?

The [bare](#) workflow has limited upsides (we can't handle as much of the complexity for you), but otherwise you're just building a React Native app and there is no need to discuss Expo-specific tradeoffs in this context.

So, this question could be better phrased as: why not the [managed](#) workflow?

The managed workflow isn't ready to be used for *all apps* yet. There are plenty of cases where its current constraints may not be appropriate for your project. The intention of this document is to outline some of those cases, so that you don't end up building an app with Expo and getting frustrated when you encounter an obstacle that you can't overcome without switching to the bare workflow. We are either planning on or actively working on building solutions to all of the features listed below, and if you think anything is missing, please bring it to our attention by posting to our [feature requests board](#).

- **The SDK doesn't support all types of background code execution** (running code when the app is not foregrounded or the device is sleeping). We support background geolocation (including geofencing) and background fetch, but we do not yet support background audio with the operating-system playback controls and you cannot handle push notifications in the background. This is a work in progress.
- **If you need to keep your app size extremely lean, Expo may not be the best choice.** The size for an Expo app on iOS is approximately 20mb (download), and Android is about 15mb. This is because Expo includes a bunch of APIs regardless of whether or not you are using them -- this lets you push over the air updates to use new APIs, but comes at the cost of binary size. We will make this customizable in the future, so you can trim down the size of your binaries.
- **If you know that you want to use a particular push notification service** (such as OneSignal) instead of Expo's [Push Notification service/API](#), you will need to use the bare workflow.
- **The minimum supported OS versions are Android 5+ and iOS 10+.** If you need to support older versions, you will not be able to use the managed workflow.
- Many device APIs are supported (check out the "SDK API Reference" in the sidebar), but **not all iOS and Android APIs are available yet**: need Bluetooth? Sorry, we haven't built support for it yet. WebRTC? Not quite. One of the most frequent requests we get is for In-App Purchases and Apple and Google Pay integration. We haven't built this yet, but it's on the roadmap. We are constantly adding new APIs, so if we don't have something you need now, you can either use ExpoKit or follow [our blog](#) to see the release notes for our SDK updates. Feature prioritization isn't strictly based off of popular vote, but it certainly helps us to gauge what is important to users.
- We typically avoid adding native modules to the SDK if they are tied to externally controlled services - we can't add something to the SDK just because a few users need it for their app, we have to think of the broader userbase.

Are we missing something here? Let us know [on Slack](#) or on our [feature requests board](#).

Up and Running

The aim of this first guide is to get an Expo application up and running as quickly as possible.

At this point we should have Expo CLI installed on our development machine and the Expo client on an iOS or Android physical device or emulator. If not, go back to the [Installation](#) guide before proceeding.

Alright, let's get started.

Creating the project

Run `expo init` to create a project. You'll be asked to name your project. The project will be created in a new directory with that name in the current working directory. I'll call mine `first-project`, and press Enter.

Next you can choose which project template to use. Choose the `tabs` option since that will give us a good starting point.

Expo CLI is now initializing a new project: it copies a basic template and installs `react`, `react-native` and `expo`.

When the project is initialized and ready to go, the command will exit.

Start the development server

Navigate to your project folder and type `npm start` to start the local development server of Expo CLI.

Expo CLI starts Metro Bundler, which is an HTTP server that compiles the JavaScript code of our app using [Babel](#) and serves it to the Expo app. It also pops up Expo Dev Tools, a control panel for developing your app, in your default web browser.

Note: If you are on MacOS and Expo CLI gets stuck on `Starting project at <path>`, you may need to [install Watchman on your machine](#). The easiest way to do this is with [Homebrew](#), `brew install watchman`.

Open the app on your phone or simulator

The fastest way to see your app on your device is to log in to Expo CLI with an Expo account (you can sign up by pressing `s` in the terminal window with the development server running, or by running `expo register`) and then use the same account to log in to Expo Client mobile app. Once you log in, a link to your current project will automatically appear inside Expo Client on your phone.

Alternatively, press `e` in the terminal or `Send link with email/SMS...` in Dev Tools to send a message with a link you can tap on your phone to open the app. You can share this link with anybody else who has the Expo app installed, but it will only be available as long as you have the project running with Expo CLI.

To open the app in the iOS simulator you can press the `i` in the terminal or `Run on iOS simulator` in Dev Tools.

To open the app in the Android emulator, first boot it up and then press `a` in the terminal or `Run on Android device/emulator` in Dev Tools.

Lastly, you will also see a QR code in terminal and Dev Tools. One fast way to open your project is to simply scan the QR code with the Expo Client app on Android or using the built-in QR code scanner of the Camera app on iOS.

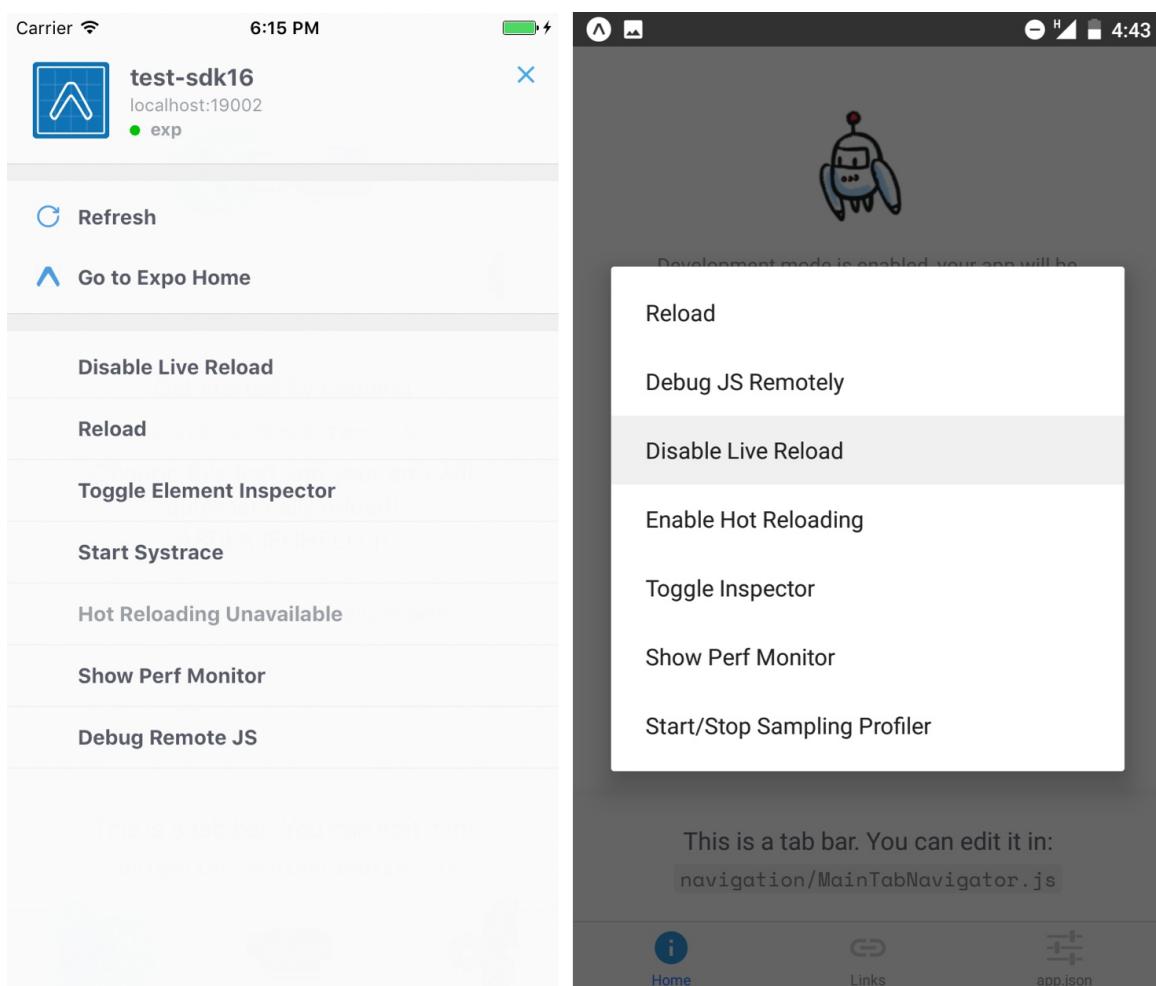
Making your first change

Open up `screens/HomeScreen.js` in your new project and change any of the text in the `render()` function. You should see your app reload with your changes.

Can't see your changes?

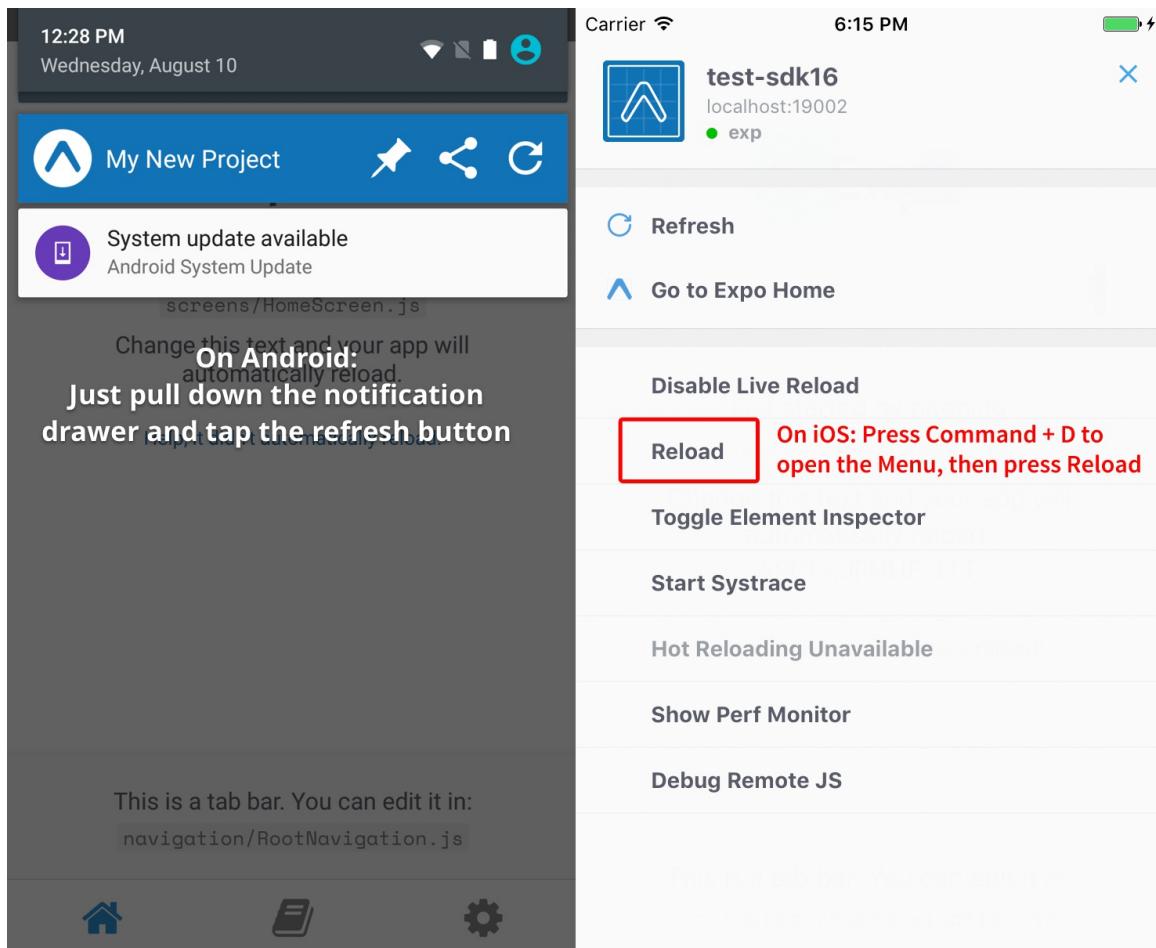
Live reload is enabled by default, but let's just make sure we go over the steps to enable it in case somehow things just aren't working.

- First, make sure you have [development mode enabled in Expo CLI](#).
- Next, close the app and reopen it.
- Once the app is open again, shake your device to reveal the developer menu. If you are using an emulator, press `⌘+d` for iOS or `ctrl+m` for Android.
- If you see `Enable Live Reload`, press it and your app will reload. If you see `Disable Live Reload` then exit the developer menu and try making another change.



Manually reloading the app

- If you've followed the above steps and live reload **still** doesn't work, [post to Expo Forums](#) to send us a support request. Until we resolve the issue for you, you can either shake the device and press `Reload`, or use one of the following tools which work both with and without development mode.



Congratulations

You have created a new Expo project, made a change, and seen it update.

Next Steps

- The [Additional Resources](#) has a bunch of useful resources for learning.
- Read about the [Expo SDK](#) to learn about some useful APIs we provide out of the box.
- Read some of our other guides, such as how to implement [Push Notifications](#), how we can take care of [Assets](#) for you, or how to build [Standalone Apps](#) you can submit to Apple or Google.
- Join us on Slack to get your questions answered.

Expo CLI

Expo CLI is a command line interface for developing Expo apps. It also includes a web-based interface (Expo Dev Tools) for using some of the most often used features also from a quick to use graphical interface.

Installation

See [Installation](#).

Commands

View the list of commands using `expo --help`:

```
Usage: expo [options] [command]

Options:
  -V, --version                                output the version number
  -o, --output [format]                          Output format. pretty (default), raw
  --non-interactive                            Fail, if an interactive prompt would be required to continue. Enabled by default if stdin is not a TTY.
  -h, --help                                     output usage information

Commands:
  android [options] [project-dir]               Opens your app in Expo on a connected Android device
  build:ios|bi [options] [project-dir]           Build a standalone IPA for your project, signed and ready for submission to the Apple App Store.
  build:android|ba [options] [project-dir]        Build a standalone APK for your project, signed and ready for submission to the Google Play Store.
  build:status|bs [options] [project-dir]         Gets the status of a current (or most recently finished) build for your project.
  bundle-assets [options] [project-dir]          Bundles assets for a detached app. This command should be executed from xcode or gradle.
  diagnostics [options] [project-dir]            Prints environment info to console.
  doctor [options] [project-dir]                 Diagnoses issues with your Expo project.
  eject [options] [project-dir]                  Creates Xcode and Android Studio projects for your app. Use this if you need to add custom native functionality.
  export [options] [project-dir]                 Exports the static files of the app for hosting it on a web server.
  fetch:ios:certs [options] [project-dir]        Fetch this project's iOS certificates/keys and provisioning profile. Writes files to the PROJECT_DIR and prints passwords to stdout.
  fetch:android:keystore [options] [project-dir] Fetch this project's Android keystore. Writes keystore to PROJECT_DIR/PROJECT_NAME.jks and prints passwords to stdout.
  fetch:android:hashes [options] [project-dir]   Fetch this project's Android key hashes needed to set up Google /Facebook authentication. Note: if you are using Google Play signing, this app will be signed with a different key after publishing to the store, and you'll need to use the hashes displayed in the Google Play console.
  fetch:android:upload-cert [options] [project-dir] Fetch this project's upload certificate needed after opting into App Signing by Google Play or after resetting a previous upload certificate.
  generate-module [options] [new-module-project] Generate a universal module for Expo from a template in [new-module-project] directory.
  init|i [options] [project-dir]                 Initializes a directory with an example project. Run it without any options and you will be prompted for the name and type.
  install:ios [options]                         Install the latest version of Expo Client for iOS on the simulator
  install:android [options]                     Install the latest version of Expo Client for Android on a connected device or emulator
```

<code>ios [options] [project-dir]</code>	Opens your app in Expo in an iOS simulator on your computer
<code>login signin [options]</code>	Login with your Expo account
<code>logout [options]</code>	Logout from your Expo account
<code>opt-in-google-play-signing [options] [project-dir]</code>	Switch from the old method of signing APKs to the new App Signing by Google Play. The APK will be signed with an upload key and after uploading to it to the store, app will be re-signed with the key from the original keystore.
<code>prepare-detached-build [options] [project-dir]</code>	Prepares a detached project for building
<code>publish:history ph [options] [project-dir]</code>	View a log of your published releases.
<code>publish:details pd [options] [project-dir]</code>	View the details of a published release.
<code>publish:set ps [options] [project-dir]</code>	Set a published release to be served from a specified channel.
<code>publish:rollback pr [options] [project-dir]</code>	Rollback an update to a channel.
<code>publish p [options] [project-dir]</code>	Publishes your project to exp.host
<code>push:android:upload [options] [project-dir]</code>	Uploads a Firebase Cloud Messaging key for Android push notifications.
<code>push:android:show [options] [project-dir]</code>	Print the value currently in use for FCM notifications for this project.
<code>push:android:clear [options] [project-dir]</code>	Deletes a previously uploaded FCM credential.
<code>register [options]</code>	Sign up for a new Expo account
<code>send [options] [project-dir]</code>	Sends a link to your project to a phone number or e-mail address
<code>s</code>	
<code>start r [options] [project-dir]</code>	Starts or restarts a local server for your app and gives you a URL to it
<code>upload:android ua [options] [projectDir] only).</code>	Uploads a standalone Android app to Google Play (works on macOS only). Uploads the latest build by default.
<code>upload:ios ui [options] [projectDir]</code>	Uploads a standalone app to Apple TestFlight (works on macOS only). Uploads the latest build by default.
<code>url u [options] [project-dir]</code>	Displays the URL you can use to view your project in Expo
<code>url:ipa [options] [project-dir]</code>	Displays the standalone iOS binary URL you can use to download your app binary
<code>url:apk [options] [project-dir]</code>	Displays the standalone Android binary URL you can use to download your app binary
<code>webhooks:set [options] [project-dir]</code>	Set a webhook for the project.
<code>webhooks:show [options] [project-dir]</code>	Show webhooks for the project.
<code>webhooks:clear [options] [project-dir]</code>	Clear a webhook associated with this project.
<code>whoami w [options]</code>	Checks with the server and then says who are logged in as

Viewing Logs

Writing to the logs in an Expo app works just like in the browser: use `console.log`, `console.warn` and `console.error`. Note: we don't currently support `console.table` outside of remote debugging mode.

Recommended: View logs with Expo tools

When you open an app that is being served from Expo CLI, the app will send logs over to the server and make them conveniently available to you. This means that you don't need to even have your device connected to your computer to see the logs -- in fact, if someone opens the app from the other side of the world you can still see your app's logs from their device.

Viewing logs with Expo CLI

If you use our command line tool Expo CLI, bundler logs and app logs will both automatically stream as long as your project is running. To stop your project (and end the logs stream), terminate the process with `ctrl+C`.

Expo Dev Tools logs

When you start a project with Expo CLI, it also opens Expo Dev Tools in your browser. Expo Dev Tools allows you to display many log windows side by side and to choose which logs to view from bundler logs and app logs from each connected device.

Optional: Manually access device logs

While it's usually not necessary, if you want to see logs for everything happening on your device, even the logs from other apps and the OS itself, you can use one of the following approaches.

View logs for an iOS simulator

Option 1: Use GUI log

- In simulator, press `⌘ + /`, or go to `Debug -> Open System Log` -- both of these open a log window that displays all of the logs from your device, including the logs from your Expo app.

Option 2: Open it in terminal

- Run `instruments -s devices`
- Find the device / OS version that the simulator you are using, eg: `iPhone 6s (9.2) [5083E2F9-29B4-421C-BDB5-893952F2B780]`
- The part in the brackets at the end is the device code, so you can now do this: `tail -f ~/Library/Logs/CoreSimulator/DEVICE_CODE/system.log`, eg: `tail -f ~/Library/Logs/CoreSimulator/5083E2F9-29B4-421C-BDB5-893952F2B780/system.log`

View logs for your iPhone

- `brew install libimobiledevice`
- Plug your phone in
- `idevicepair pair`
- Press accept on your device
- Run `idevicesyslog`

View logs from Android device or emulator

- Ensure Android SDK is installed
- Ensure that [USB debugging is enabled on your device](#) (not necessary for emulator).
- Run `adb logcat`

Debugging

Using a Simulator / Emulator

There is no substitute to testing the performance and feel of your app on an actual device, but when it comes to debugging you might have an easier time using an emulator/simulator.

Apple refers to their emulator as a "Simulator" and Google refers to theirs as an "Emulator".

iOS

Make sure you have the latest Xcode (e.g. from the [Mac App Store](#)). This includes the iOS Simulator, among several other tools.

Android

Follow our [Android Studio emulator guide](#) to set up the Android tools and create a virtual device to use for testing.

Developer Menu

This menu gives you access to several functions which are useful for debugging. It is also known as the Debug Menu. Invoking it depends on the device where you are running your application.

On an iOS Device

Shake the device a little bit.

On iOS Simulator

Hit `Ctrl-Cmd-Z` on a Mac in the emulator to simulate the shake gesture, or press `Cmd+D`.

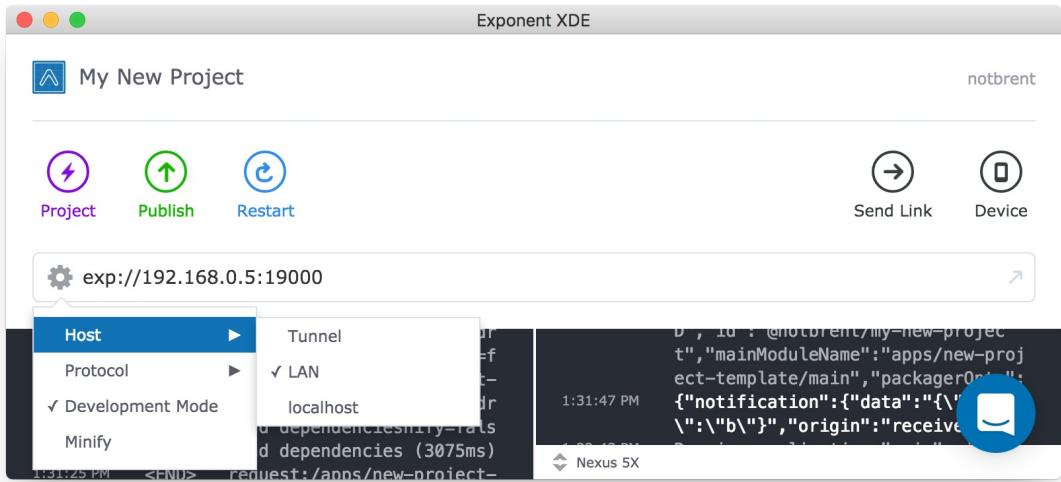
On Android Virtual Device

Either hit `Cmd+M`, or run `adb shell input keyevent 82` in your terminal window.

Debugging Javascript

You can debug Expo apps using the Chrome debugger tools. Rather than running your app's JavaScript on your phone, it will instead run it inside of a webworker in Chrome. You can then set breakpoints, inspect variables, execute code, etc, as you would when debugging a web app.

- To ensure the best debugging experience, first change your host type in Expo Dev Tools to `LAN` or `localhost`. If you use `Tunnel` with debugging enabled, you are likely to experience so much latency that your app is unusable. While here, also ensure that `Development Mode` is checked.



- If you are using `LAN`, make sure your device is on the same wifi network as your development machine. This may not work on some public networks. `localhost` will not work for iOS unless you are in the simulator, and it only works on Android if your device is connected to your machine via usb.
- Open the app on your device, reveal the developer menu then tap on `Debug JS Remotely`. This should open up a Chrome tab with the URL `http://localhost:19001/debugger-ui`. From there, you can set breakpoints and interact through the JavaScript console. Shake the device and stop Chrome debugging when you're done.
- Line numbers for `console.log` statements don't work by default when using Chrome debugging. To get correct line numbers open up the Chrome Dev Tools settings, go to the "Blackboxing" tab, make sure that "Blackbox content scripts" is checked, and add `expo/src/Logs.js` as a pattern with "Blackbox" selected.

Troubleshooting localhost debugging

When you start a project with Expo CLI and when you press `Run on Android device/emulator` in Expo Dev Tools (or `a` in the terminal), Expo CLI will automatically tell your device to forward `localhost:19000` and `19001` to your development machine, as long as your device is plugged in or emulator is running. If you are using `localhost` for debugging and it isn't working, close the app and open it up again using `Open on Android`. Alternatively, you can manually forward the ports using the following command if you have the Android developer tools installed: `adb reverse tcp:19000 tcp:19000 - adb reverse tcp:19001 tcp:19001`

Source maps and async functions

Source maps and async functions aren't 100% reliable. React Native doesn't play well with Chrome's source mapping in every case, so if you want to make sure you're breakpointing in the correct place, you should use the `debugger` call directly from your code.

Debugging HTTP

To debug your app's HTTP requests you should use a proxy. The following options will all work:

- [Charles Proxy](#) (\$50 USD, our preferred tool)
- [mitmproxy](#)

- Fiddler

On Android, the [Proxy Settings](#) app is helpful for switch between debug and non-debug mode. Unfortunately it doesn't work with Android M yet.

There is [future work](#) to get network requests showing up in Chrome DevTools.

Hot Reloading and Live Reloading

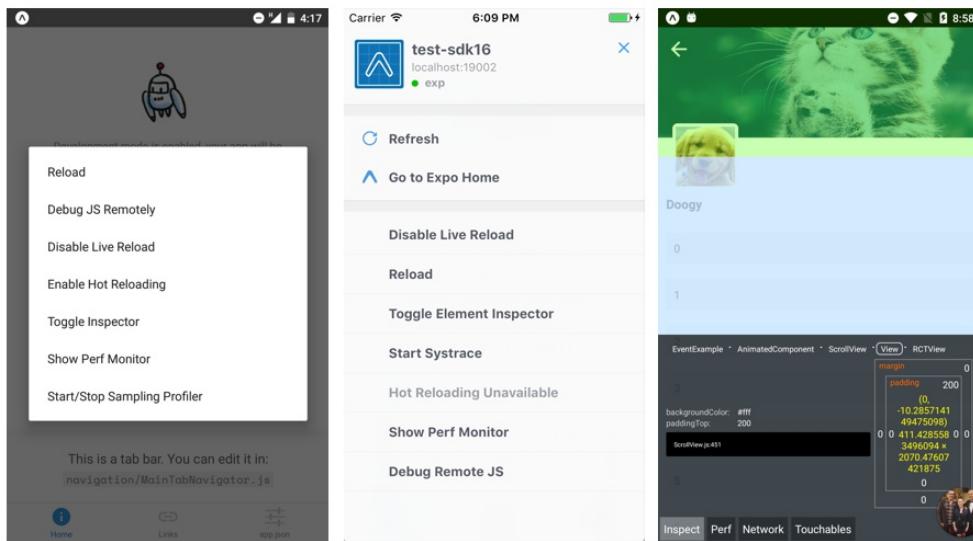
[Hot Module Reloading](#) is a quick way to reload changes without losing your state in the screen or navigation stack. To enable, invoke the developer menu and tap the "Enable Hot Reloading" item. Whereas Live Reload will reload the entire JS context, Hot Module Reloading will make your debug cycles even faster. However, make sure you don't have both options turned on, as that is unsupported behavior.

Other Debugging Tips

Dotan Nahum outlined in his "[Debugging React Native Applications](#)" Medium post other useful tools such as spying on bridge messages and JSEventLoopWatchdog.

Development Mode

React Native includes some very useful tools for development: remote JavaScript debugging in Chrome, live reload, hot reloading, and an element inspector similar to the beloved inspector that you use in Chrome. It also performs bunch of validations while your app is running to give you warnings if you're using a deprecated property or if you forgot to pass a required property into a component, for example.

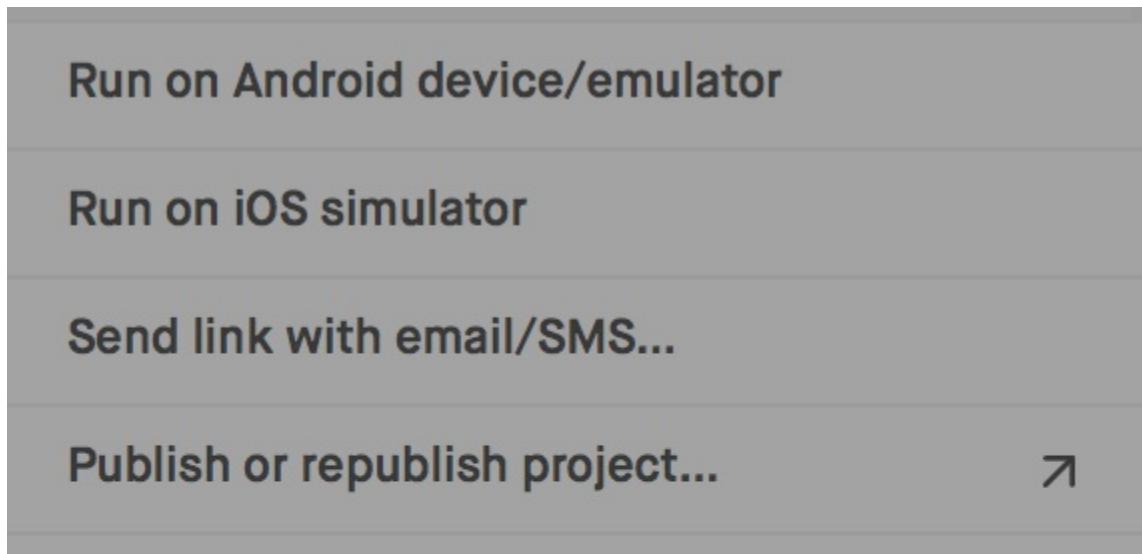


Development mode on Android, iOS, and the inspector in action.

This comes at a cost: your app runs slower in development mode. You can toggle it on and off from Expo Dev Tools and Expo CLI. When you switch it, just close and re-open your app for the change to take effect. **Any time you are testing the performance of your app, be sure to disable development mode.**

Toggling Development Mode in Expo Dev Tools

To enable development mode, make sure the "Production mode" switch is turned off:



Toggling Development Mode in Expo CLI

In the terminal with your project running in Expo CLI, press `p` to toggle the production mode.

Showing the Developer Menu

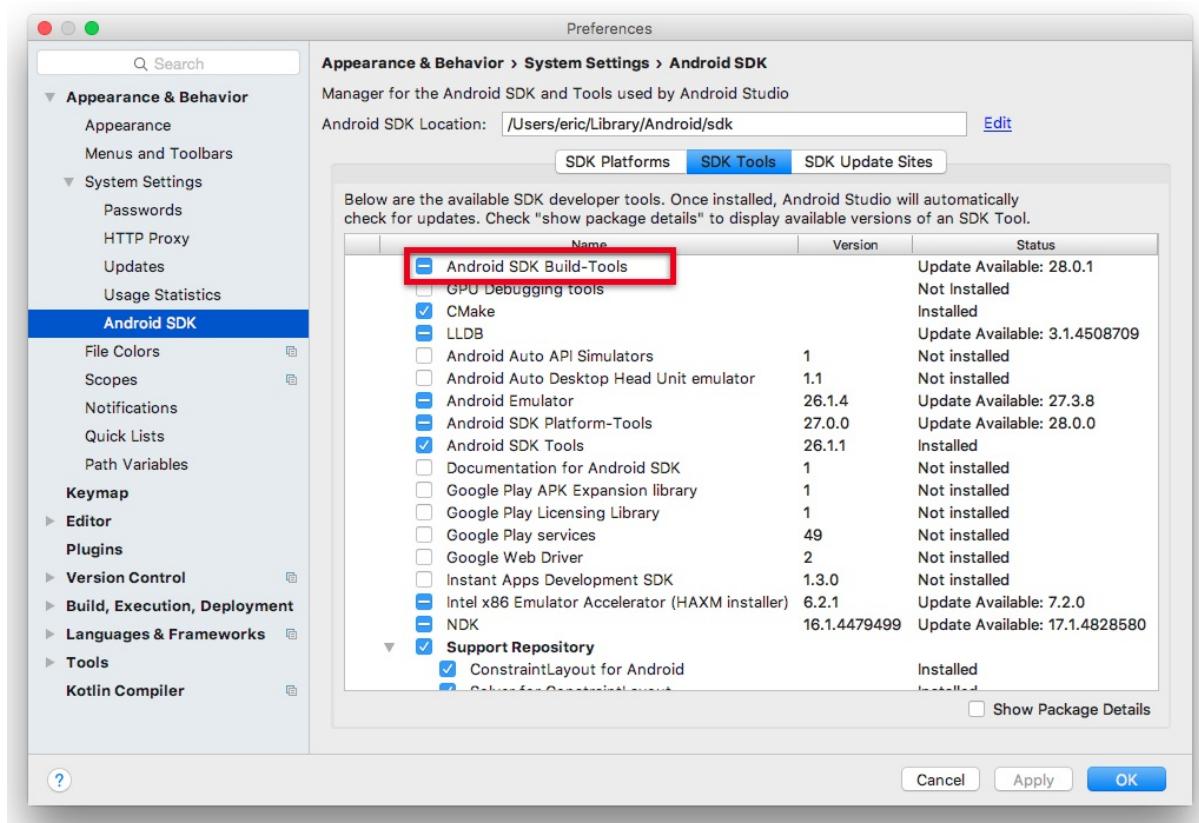
When in Development Mode, depending on your settings of the Expo Client, you will either shake your device or use a two-finger force touch to reveal the Developer Menu. When using an emulator, use the key command `⌘+D` for iOS and `Ctrl+M` for Android.

Android Studio Emulator

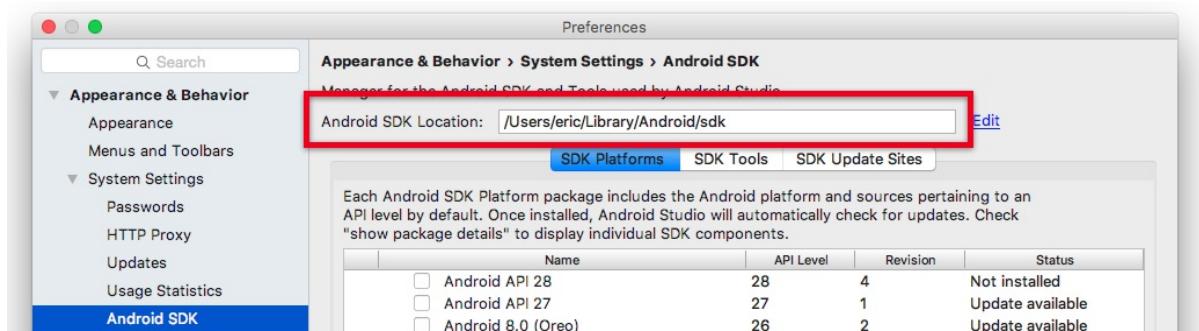
If you don't have an Android device available to test with, we recommend using the default emulator that comes with Android Studio. If you run into any problems setting it up, follow the steps in this guide.

Step 1: Set up Android Studio's tools

- Install Android Studio 3.0+.
- Go to Preferences -> Appearance & Behavior -> System Settings -> Android SDK. Click on the "SDK Tools" tab and make sure you have at least one version of the "Android SDK Build-Tools" installed.



- Copy or remember the path listed in the box that says "Android SDK Location."

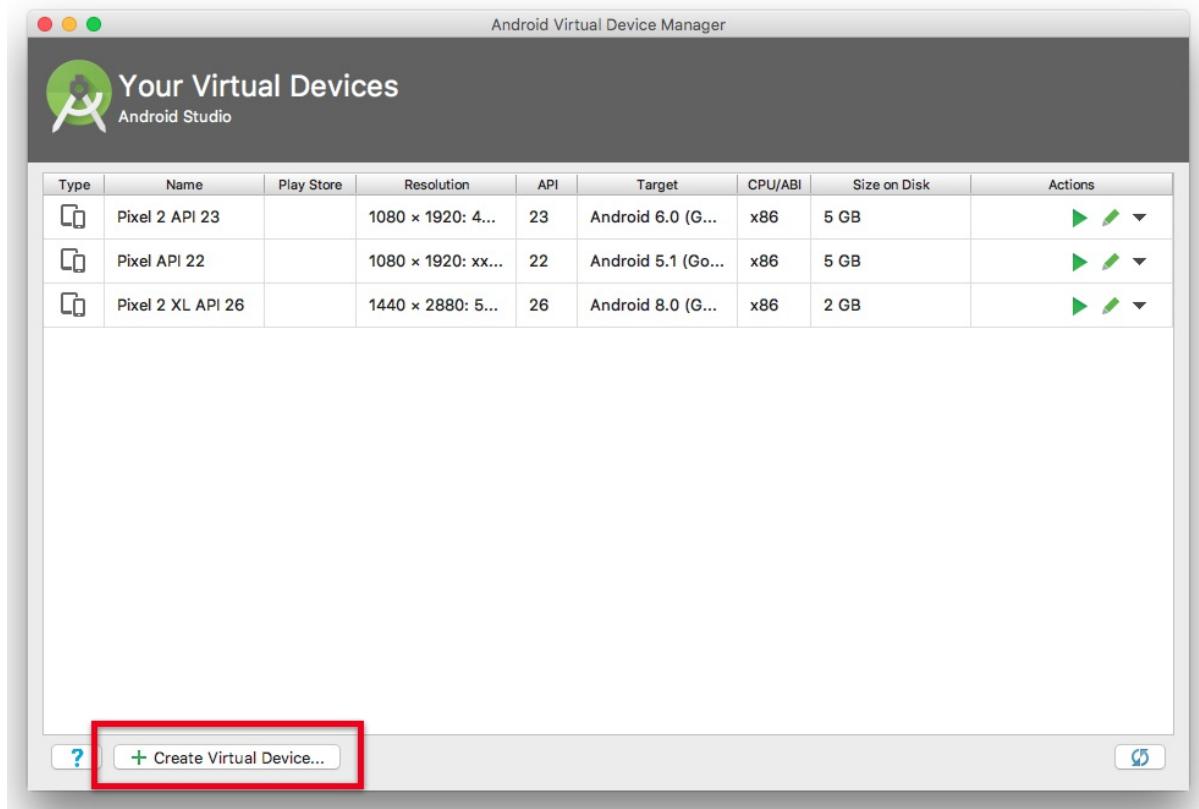


- If you are on macOS or Linux, add the Android SDK location to your PATH using `~/.bash_profile` or `~/.bashrc`. You can do this by adding a line like `export ANDROID_SDK=/Users/myuser/Library/Android/sdk`.

- On macOS, you will also need to add `platform-tools` to your `~/.bash_profile` OR `~/.bash_rc.` , by adding a line like `export PATH=/Users/myuser/Library/Android/sdk/platform-tools:$PATH`
- Make sure that you can run `adb` from your terminal.

Step 2: Set up a virtual device

- In Android Studio, go to Tools -> Android -> AVD Manager.
- Press the "+ Create Virtual Device" button.



- Choose the type of hardware you'd like to emulate. We recommend testing against a variety of devices, but if you're unsure where to start, the newest device in the Pixel line could be a good choice.
- Select an OS version to load on the emulator (probably one of the system images in the "Recommended" tab), and download the image.
- Change any other settings you'd like, and press "Finish" to create the virtual device. You can now run this device anytime by pressing the Play button in the AVD Manager window.

Multiple adb versions

Having multiple `adb` versions on your system can result in the error `adb server version (xx) doesn't match this client (yy); killing...`

This is because the `adb` version on your system is different from the `adb` version on the android sdk `platform-tools`.

- Open the terminal and check the `adb` version on the system:

```
$adb version
```

- And from the Android SDK platform-tool directory:

```
$cd ~/Android/Sdk/platform-tools
```

```
$./adb version
```

- Copy `adb` from Android SDK directory to `usr/bin` directory:

```
$sudo cp ~/Android/Sdk/platform-tools/adb /usr/bin
```

Configuration with app.json

`app.json` is your go-to place for configuring parts of your app that don't belong in code. It is located at the root of your project next to your `package.json`. It looks something like this:

```
{  
  "expo": {  
    "name": "My app",  
    "slug": "my-app",  
    "sdkVersion": "UNVERSIONED",  
    "privacy": "public"  
  }  
}
```

`app.json` was previously referred to as `exp.json`, but for consistency with [Create React Native App](#) it has been consolidated under one file. If you are converting your app from using `exp.json` to `app.json`, all you need to do is add an `"expo"` key at the root of `app.json`, as the parent of all other keys.

Most configuration from `app.json` is accessible at runtime from your JavaScript code via `Constants.manifest`. Sensitive information such as secret keys are removed. See the `"extra"` key below for information about how to pass arbitrary configuration data to your app.

ExpoKit

While some of the properties defined in `app.json` can be applied at runtime, others require modifying native build configuration files. For ExpoKit projects, we only apply these settings once, at the time the native projects are generated (i.e. when you run `expo eject`).

This means that for existing ExpoKit projects, **changing certain properties in `app.json` will not have the desired effect**. Instead, you must modify the corresponding native configuration files. In most cases, we've provided here a brief description of the files or settings that need to be changed, but you can also refer to the Apple and Android documentation for more information.

Properties

The following is a list of properties that are available for you under the `"expo"` key in `app.json`:

"name"

Required. The name of your app as it appears both within Expo and on your home screen as a standalone app.

ExpoKit: To change the name of your app, edit the "Display Name" field in Xcode and the `app_name` string in `android/app/src/main/res/values/strings.xml`.

"description"

A short description of what your app is and why it is great.

"slug"

Required. The friendly url name for publishing. eg: `my-app-name` will refer to the `expo.io/@your-username/my-app-name` project.

"privacy"

Either `public` or `unlisted`. If not provided, defaults to `unlisted`. In the future `private` will be supported. `unlisted` hides the experience from search results. Valid values: `public`, `unlisted`

"sdkVersion"

Required. The Expo sdkVersion to run the project on. This should line up with the version specified in your package.json.

"version"

Your app version; use whatever versioning scheme that you like.

ExpoKit: To change your app version, edit the "Version" field in Xcode and the `versionName` string in `android/app/build.gradle`.

"platforms"

Platforms that your project explicitly supports. If not specified, it defaults to `["ios", "android"]`.

"githubUrl"

If you would like to share the source code of your app on Github, enter the URL for the repository here and it will be linked to from your Expo project page.

"orientation"

Lock your app to a specific orientation with `portrait` or `landscape`. Defaults to no lock. Valid values: 'default', 'portrait', 'landscape'

"primaryColor"

On Android, this will determine the color of your app in the multitasker. Currently this is not used on iOS, but it may be used for other purposes in the future.

6 character long hex color string, eg: "#000000"

"icon"

Local path or remote url to an image to use for your app's icon. We recommend that you use a 1024x1024 png file. This icon will appear on the home screen and within the Expo app.

ExpoKit: To change your app's icon, edit or replace the files in `ios/<PROJECT-NAME>/Assets.xcassets/AppIcon.appiconset` (we recommend using Xcode), and `android/app/src/main/res/mipmap-<RESOLUTION>`. Be sure to follow the guidelines for each platform ([iOS](#), [Android 7.1 and below](#), and [Android](#))

8+) and to provide your new icon in each existing size.

"appKey"

By default, Expo looks for the application registered with the AppRegistry as `main`. If you would like to change this, you can specify the name in this property.

"androidShowExponentNotificationInShellApp"

Adds a notification to your standalone app with refresh button and debug info.

"scheme"

Standalone Apps Only. URL scheme to link into your app. For example, if we set this to `'demo'`, then `demo://` URLs would open your app when tapped. String beginning with a letter followed by any combination of letters, digits, "+", "." or "-".

ExpoKit: To change your app's scheme, replace all occurrences of the old scheme in `Info.plist`, `AndroidManifest.xml`, and `android/app/src/main/java/host/exp/exponent/generated/AppConstants.java`.

"entryPoint"

The relative path to your main JavaScript file.

"extra"

Any extra fields you want to pass to your experience. Values are accessible via `Constants.manifest.extra` ([read more](#))

"rnCliPath"

"packagerOpts"

"ignoreNodeModulesValidation"

"nodeModulesPath"

"facebookAppId"

Used for all Facebook libraries. Set up your Facebook App ID at <https://developers.facebook.com>.

ExpoKit: To change this field, edit `Info.plist`.

"facebookDisplayName"

Used for native Facebook login.

ExpoKit: To change this field, edit `Info.plist`.

"facebookScheme"

Used for Facebook native login. Starts with 'fb' and followed by a string of digits, like 'fb1234567890'. You can find your scheme at <https://developers.facebook.com/docs/facebook-login/ios> in the 'Configuring Your info.plist' section.

ExpoKit: To change this field, edit `Info.plist`.

"locales"

Provide overrides by locale for System Dialog prompts like Permissions alerts

ExpoKit: To add or change language and localization information in your iOS app, you need to use Xcode.

"assetBundlePatterns"

An array of file glob strings which point to assets that will be bundled within your standalone app binary. Read more in the [Offline Support guide](#)

"androidStatusBar"

Configuration for android statusbar.

```
{  
  "androidStatusBar": {  
    /*  
     * Configure the statusbar icons to have light or dark color.  
     * Valid values: "light-content", "dark-content".  
    */  
    "barStyle": STRING,  
  
    /*  
     * Configuration for android statusbar.  
     * 6 character long hex color string, eg: "#000000"  
    */  
    "backgroundColor": STRING  
  }  
}
```

"splash"

Configuration for loading and splash screen for standalone apps.

```
{  
  "splash": {  
    /*  
     * Color to fill the loading screen background  
     * 6 character long hex color string, eg: "#000000"  
    */  
    "backgroundColor": STRING,  
  
    /*  
     * Determines how the "image" will be displayed in the splash loading screen.  
     * Must be one of "cover" or "contain", defaults to `contain`.  
     * Valid values: "cover", "contain"  
    */  
    "resizeMode": STRING,  
  
    /*  
     * Local path or remote url to an image.  
    */  
  }  
}
```

```

        Will fill the background of the loading/splash screen.
        Image size and aspect ratio are up to you. Must be a .png.
    */
    "image": STRING
}
}

```

ExpoKit: To change your iOS app's splash screen, use Xcode to edit `LaunchScreen.xib`. For Android, edit or replace the files in `android/app/src/main/res/drawable-<RESOLUTION>`; to change the background color, edit `android/app/src/main/res/values/colors.xml`; and to change the resizeMode, set `SHOW_LOADING_VIEW_IN_SHELL_APP` in `android/app/src/main/java/host/exp/exponent/generated/AppConstants.java` (`true` for "contain", `false` for "cover").

"notification"

Configuration for remote (push) notifications.

```

{
  "notification": {
    /*
      Local path or remote url to an image to use as the icon for push notifications.
      96x96 png grayscale with transparency.
    */
    "icon": STRING,
    /*
      Tint color for the push notification image when it appears in the notification tray.
      6 character long hex color string eg: "#000000"
    */
    "color": STRING,
    /*
      Show each push notification individually "default" or collapse into one "collapse".
      Valid values: "default", "collapse"
    */
    "androidMode": STRING,
    /*
      If "androidMode" is set to "collapse", this title is used for the collapsed notification message.
      eg: "{unread_notifications} new interactions"
    */
    "androidCollapsedTitle": STRING
  }
}

```

ExpoKit: To change the notification icon, edit or replace the `shell_notification_icon.png` files in `android/app/src/main/res/mipmap-<RESOLUTION>`. On iOS, notification icons are the same as the app icon. All other properties are set at runtime.

"hooks"

Configuration for scripts to run to hook into the publish process

```

{
  "hooks": {
    "postPublish": STRING
  }
}

```

"updates"

Configuration for how and when the app should request OTA JavaScript updates

```
{  
  "updates": {  
    /*  
     * If set to false, your standalone app will never download any code.  
     * And will only use code bundled locally on the device.  
     * In that case, all updates to your app must be submitted through Apple review.  
     * Defaults to true.  
  
     * Note that this will not work out of the box with ExpoKit projects.  
    */  
    "enabled": BOOLEAN,  
  
    /*  
     * By default, Expo will check for updates every time the app is loaded.  
     * Set this to `ON_ERROR_RECOVERY` to disable automatic checking unless recovering from an error.  
  
     * Must be one of `ON_LOAD` or `ON_ERROR_RECOVERY`.  
    */  
    "checkAutomatically": STRING,  
  
    /*  
     * How long (in ms) to allow for fetching OTA updates before falling back to a cached version of the app.  
  
     * Defaults to 30000 (30 sec). Must be between 0 and 300000 (5 minutes).  
    */  
    "fallbackToCacheTimeout": NUMBER  
  }  
}
```

ExpoKit: To change the value of `enabled`, edit `ios/<PROJECT-NAME>/Supporting/EXShell.plist` and `android/app/src/main/java/host/exp/exponent/generated/AppConstants.java`. All other properties are set at runtime.

"ios"

Standalone Apps Only. iOS standalone app specific configuration

```
{  
  "ios": {  
    /*  
     * The bundle identifier for your iOS standalone app.  
     * You make it up, but it needs to be unique on the App Store.  
  
     * stackoverflow.com/questions/11347470/what-does-bundle-identifier-mean-in-the-ios-project.  
  
     * iOS bundle identifier notation unique name for your app.  
     * For example, host.exp.exponent, where exp.host is our domain  
     * and Expo is our app.  
  
     * ExpoKit: use Xcode to set this.  
    */  
    "bundleIdentifier": STRING,  
  
    /*  
     * Build number for your iOS standalone app. Must be a string  
     * that matches Apple's format for CFBundleVersion.  
  
     * developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CoreFoundationKeys.html#//apple_ref/plist/bundle-version  
    */  
  }  
}
```

```

onKeys.html#/apple_ref/doc/uid/20001431-102364.

    ExpoKit: use Xcode to set this.
*/
"buildNumber": STRING,

/*
  Local path or remote URL to an image to use for your app's
  icon on iOS. If specified, this overrides the top-level "icon" key.

  Use a 1024x1024 icon which follows Apple's interface guidelines for icons, including color profile and transpar
  ency.

  Expo will generate the other required sizes.
  This icon will appear on the home screen and within the Expo app.
*/
"icon": STRING,

/*
  URL to your app on the Apple App Store, if you have deployed it there.
  This is used to link to your store page from your Expo project page if your app is public.
*/
"appStoreUrl": STRING,

/*
  Whether your standalone iOS app supports tablet screen sizes.
  Defaults to `false`.

  ExpoKit: use Xcode to set this.
*/
"supportsTablet": BOOLEAN,

/*
  If true, indicates that your standalone iOS app does not support handsets.
  Your app will only support tablets.

  ExpoKit: use Xcode to set this.
*/
"isTabletOnly": BOOLEAN,

/*
  Dictionary of arbitrary configuration to add to your standalone app's native Info.plist. Applied prior to all o
  ther Expo-specific configuration.

  No other validation is performed, so use this at your own risk of rejection from the App Store.
*/
"infoPlist": OBJECT,

/*
  An array that contains Associated Domains for the standalone app. See apple's docs for config: https://develope
  r.apple.com/documentation/uikit/core_app/allowing_apps_and_websites_to_link_to_your_content/enabling_universal_links
  Entries must be prefixed with "www."

  ExpoKit: use Xcode to set this.
*/
"associatedDomains": ARRAY,

/*
  A boolean indicating if the app uses iCloud Storage for DocumentPicker.
  See DocumentPicker docs for details.

  ExpoKit: use Xcode to set this.
*/
"usesIcloudStorage": BOOLEAN,
*/

```

```

Extra module configuration to be added to your app's native Info.plist.

For ExpoKit apps, just add these to the Info.plist file directly.

*/
"config": {
  /*
    Branch (https://branch.io/) key to hook up Branch linking services.
  */
  "branch": {
    /*
      Your Branch API key
    */
    "apiKey": STRING
  },
  /*
    Sets `ITSSAppUsesNonExemptEncryption` in the standalone ipa's Info.plist to the given boolean value.
  */
  "usesNonExemptEncryption": BOOLEAN,
  /*
    Google Maps iOS SDK key for your standalone app.

    developers.google.com/maps/documentation/ios-sdk/start
  */
  "googleMapsApiKey": STRING,
  /*
    Google Sign-In iOS SDK keys for your standalone app.

    developers.google.com/identity/sign-in/ios/start-integrating
  */
  "googleSignIn": {
    /*
      The reserved client ID URL scheme.
      Can be found in GoogeService-Info.plist.
    */
    "reservedClientId": STRING
  },
  "splash": {
    /*
      Color to fill the loading screen background 6 character long hex color string, eg: "#000000"
    */
    "backgroundColor": STRING,
    /*
      Determines how the "image" will be displayed in the splash loading screen.
      Must be one of "cover" or "contain", defaults to "contain".
      Valid values: "cover", "contain"
    */
    "resizeMode": STRING,
    /*
      Local path or remote url to an image to fill the background of the loading screen.
      Image size and aspect ratio are up to you.
      Must be a .png.
    */
    "image": STRING,
    /*
      Local path or remote url to an image to fill the background of the loading screen.
      Image size and aspect ratio are up to you.
      Must be a .png.
    */
  }
}

```

```
        "tabletImage": STRING
    }
}
```

"android"

Standalone Apps Only. Android standalone app specific configuration

```
{
  "android": {
    /*
      The package name for your Android standalone app.
      You make it up, but it needs to be unique on the Play Store.

      stackoverflow.com/questions/6273892/android-package-name-convention

      Reverse DNS notation unique name for your app.
      For example, host.exp.exponent, where exp.host is our domain and Expo is our app.
      The name may only contain lowercase and uppercase letters (a-z, A-Z),
      numbers (0-9) and underscores (_). Each component of the name should start
      with a lowercase letter.

      ExpoKit: this is set in `android/app/build.gradle` as well as your
      AndroidManifest.xml file (multiple places).
    */
    "package": STRING,

    /*
      Version number required by Google Play.
      Increment by one for each release.
      Must be an integer.
      developer.android.com/studio/publish/versioning.html

      ExpoKit: this is set in `android/app/build.gradle`.
    */
    "versionCode": NUMBER,

    /*
      Local path or remote url to an image to use for your app's icon on Android.
      If specified, this overrides the top-level "icon" key.

      We recommend that you use a 1024x1024 png file.
      Transparency is recommended for the Google Play Store.
      This icon will appear on the home screen and within the Expo app.
    */
    "icon": STRING,

    /*
      Settings for an Adaptive Launcher Icon on Android.
      https://developer.android.com/guide/practices/ui_guidelines/icon_design_adaptive

      ExpoKit: icons are saved in `android/app/src/main/res/mipmap-<RESOLUTION>-v26`
      and the "backgroundColor" is set in `android/app/src/main/res/values/colors.xml`.

    */
    "adaptiveIcon": {
      /*
        Local path or remote url to an image to use for
        the foreground of your app's icon on Android.

        We recommend that you use a 1024x1024 png file,
        leaving at least the outer 1/6 transparent on each side.
        If specified, this overrides the top-level "icon" and the "android.icon" keys.
        This icon will appear on the home screen.
      */
    }
  }
}
```

```

/*
"foregroundImage": STRING,
/*
Color to use as the background for your app's Adaptive Icon on Android.
Defaults to white (#FFFFFF).

Has no effect if "foregroundImage" is not specified.
*/
"backgroundColor": STRING,
/*
Local path or remote url to a background image for
the background of your app's icon on Android.

If specified, this overrides the "backgroundColor" key.
Must have the same dimensions as "foregroundImage", and has no effect if
"foregroundImage" is not specified.
*/
"backgroundImage": STRING
},

/*
URL to your app on the Google Play Store, if you have deployed it there.
This is used to link to your store page from your Expo project page if your app is public.
*/
"playStoreUrl": STRING,
/*
List of additional permissions the standalone app will request upon installation,
along with the minimum necessary for an Expo app to function.

To use ALL permissions supported by Expo, do not specify the "permissions" key.

To use ONLY the following minimum necessary permissions and none of the extras supported
by Expo, set "permissions" to []. The minimum necessary permissions do not require a
Privacy Policy when uploading to Google Play Store and are:



- receive data from Internet
- view network connections
- full network access
- change your audio settings
- draw over other apps
- prevent device from sleeping



To use the minimum necessary permissions ALONG with certain additional permissions,
specify those extras in "permissions", e.g.

["CAMERA", "RECORD_AUDIO"]

ExpoKit: to change the permissions your app requests, you'll need to edit
AndroidManifest.xml manually. To prevent your app from requesting one of the
permissions listed below, you'll need to explicitly add it to `AndroidManifest.xml`  

along with a `tools:node="remove"` tag.
*/
"permissions": [
  "ACCESS_COARSE_LOCATION",
  "ACCESS_FINE_LOCATION",
  "CAMERA",
  "MANAGE_DOCUMENTS",
  "READ_CONTACTS",
  "READ_CALENDAR",
  "WRITE_CALENDAR",
  "READ_EXTERNAL_STORAGE",
  "READ_PHONE_STATE",
  "RECORD_AUDIO",
]

```

```

    "USE_FINGERPRINT",
    "VIBRATE",
    "WAKE_LOCK",
    "WRITE_EXTERNAL_STORAGE",
    "com.anddoes.launcher.permission.UPDATE_COUNT",
    "com.android.launcher.permission.INSTALL_SHORTCUT",
    "com.google.android.c2dm.permission.RECEIVE",
    "com.google.android.gms.permission.ACTIVITY_RECOGNITION",
    "com.google.android.providers.gsf.permission.READ_GSERVICES",
    "com.htc.launcher.permission.READ_SETTINGS",
    "com.htc.launcher.permission.UPDATE_SHORTCUT",
    "com.majeur.launcher.permission.UPDATE_BADGE",
    "com.sec.android.provider.badge.permission.READ",
    "com.sec.android.provider.badge.permission.WRITE",
    "com.sonyericsson.home.permission.BROADCAST_BADGE"
],
/*
Location of the google-services.json file for configuring Firebase. Including this
key automatically enables FCM in your standalone app.

For ExpoKit apps, add or edit the file directly at `android/app/google-services.json`.
To enable FCM, edit the value of `FCM_ENABLED` in
`android/app/src/main/java/host/exp/exponent/generated/AppConstants.java` .
*/
"googleServicesFile": STRING,
/*
Extra module configuration to be added to your app's native AndroidManifest.xml.

For ExpoKit apps, just add these to the AndroidManifest.xml file directly.
*/
"config": {
/*
Branch (https://branch.io/) key to hook up Branch linking services.
*/
"branch": {
/*
Your Branch API key
*/
"apiKey": STRING
},
/*
Google Developers Fabric keys to hook up Crashlytics and other services.
get.fabric.io/
*/
"fabric": {
/*
Your Fabric API key
*/
"apiKey": STRING,
/*
Your Fabric build secret
*/
"buildSecret": STRING
},
/*
Google Maps Android SDK key for your standalone app.
developers.google.com/maps/documentation/android-api/signup
*/
"googleMaps": {
/*
Your Google Maps Android SDK API key
*/
}
}

```

```

        /*
      "apiKey": STRING
    }

/*
  Google Sign-In Android SDK keys for your standalone app.
  developers.google.com/identity/sign-in/android/start-integrating
*/
"googleSignIn": {
  /*
    The Android API key.
    Can be found in the credentials section of the developer console
    or in "google-services.json"
  */
  "apiKey": STRING,
}

/*
  The SHA-1 hash of the signing certificate used to build the apk without any separator `;`.
  Can be found in "google-services.json".
  developers.google.com/android/guides/client-auth
*/
"certificateHash": STRING
},
),

/*
  Configuration for loading and splash screen for standalone Android apps.
*/
"splash": {
  /*
    Color to fill the loading screen background
    6 character long hex color string, eg: "#000000"
  */
  "backgroundColor": STRING,

  /*
    Determines how the "image" will be displayed in the splash loading screen.
    Must be one of "cover" or "contain", defaults to "contain".
    Valid values: "cover", "contain"
  */
  "resizeMode": STRING,
}

/*
  Local path or remote url to an image to fill the background of the loading screen in 'cover' mode.
  Image size and aspect ratio are up to you.
  Pay extra attention to the size of each image.
  See here: https://docs.expo.io/versions/latest/guides/splash-screens.html#differences-between-environments-android
  Must be a .png
  For more information see https://developer.android.com/training/multiscreen/screendensities
*/
"mdpi": STRING, // natural sized image (baseline)
"hdpi": STRING, // scale 1.5x
"xdpi": STRING, // scale 2x
"xxhdpi": STRING, // scale 3x
"xxxhdpi": STRING // scale 4x
},
)

/*
  Configuration for setting custom intent filters in Android manifest.
  The following example demonstrates how to set up deep links. When
  the user taps a link matching *.myapp.io, they will be shown a
  dialog asking whether the link should be handled by your app or by
  the web browser.

```

The data attribute may either be an object or an array of objects.

The object may have the following keys to specify attributes of URLs matched by the filter:

- scheme (string): the scheme of the URL, e.g. "https"
- host (string): the host, e.g. "myapp.io"
- port (string): the port, e.g. "3000"
- path (string): an exact path for URLs that should be matched by the filter, e.g. "/records"
- pathPattern (string): a regex for paths that should be matched by the filter, e.g. ".+"
- pathPrefix (string): a prefix for paths that should be matched by the filter, e.g. "/records/" will match "/records/123"
- mimeType (string): a mime type for URLs that should be matched by the filter

See Android's documentation for more details on intent filter matching:

developer.android.com/guide/components/intents-filters

You may also use an intent filter to set your app as the default handler for links (without showing the user a dialog with options). To do so, you must set "autoVerify": true on the filter object below, and then configure your server to serve a JSON file verifying that you own the domain. See Android's documentation for details:

developer.android.com/training/app-links

To add or edit intent filters in an ExpoKit project, edit `AndroidManifest.xml` directly.

```
/*
"intentFilters": [
  {
    "action": "VIEW",
    "data": {
      "scheme": "https",
      "host": "*.myapp.io"
    },
    "category": [
      "BROWSABLE",
      "DEFAULT"
    ]
  }
]
```

Publishing

While you're developing your project, you're writing code on your computer, and when you use Expo CLI, a server and the React Native packager run on your machine and bundle up all your source code and make it available from a URL. Your URL for a project you're working on probably looks something like this: `exp://i3-kvb.ccheever.an-example.exp.direct:80`

`exp.direct` is a domain we use for tunneling, so that even if you're behind a VPN or firewall, any device on the internet that has your URL should be able to access your project. This makes it much easier to open your project on your phone or send it someone else you're collaborating with who isn't on the same LAN.

But since the packager and server are running on your computer, if you turn off your laptop or stop Expo CLI, you won't be able to load your project from that URL. "Publish" is the term we use for deploying your project. It makes your project available at a persistent URL, for example <https://expo.io/@community/native-component-list>, which can be opened with the Expo Client app. It also uploads all of your app images, fonts, and videos to a CDN ([read more here](#)).

How to Publish

To publish a project, click the Publish button in Expo Dev Tools. (It's in the left side bar.) If you're using command line, run `expo publish`. No setup is required, go ahead and create a new project and publish it without any changes and you will see that it works.

When you do this, the packager will minify all your code and generate two versions of your code (one for iOS, one for Android) and then upload those to a CDN. You'll get a link like <https://exp.host/@ccheever/an-example> that anyone can load your project from.

Any time you want to deploy an update, hit publish again and a new version will be available immediately to your users the next time they open it.

Deploying to the App Store and Play Store

When you're ready to distribute your app to end-users, you can create a standalone app binary (an ipa or apk file) and put it in the iOS App Store and the Google Play Store. See [Distributing Your App](#).

The standalone app knows to look for updates at your app's published url, and if you publish an update then the next time a user opens your app they will automatically download the new version. These are commonly referred to as "Over the Air" (OTA) updates, the functionality is similar to [CodePush](#), but it is built into Expo so you don't need to install anything.

To configure the way your app handles JS updates, see [Offline Support](#).

Limitations

Some native configuration can't be updated by publishing

If you make any of the following changes in `app.json`, you will need to re-build the binaries for your app for the change to take effect:

- Increment the Expo SDK Version
- Change anything under the `ios` or `android` keys
- Change your app `splash`
- Change your app `icon`
- Change your app `name`
- Change your app `scheme`
- Change your `facebookScheme`
- Change your bundled assets under `assetBundlePatterns`

On iOS, you can't share your published link

When you publish, any Android user can open your app inside Expo Client immediately.

Due to restrictions imposed by Apple, the best way to share your published app is to build a native binary with Expo's build service. You can use Apple TestFlight to share the app with your testers, and you can submit it to the iTunes Store to share more widely.

Privacy

You can set the privacy of your project in your `app.json` configuration file by setting the key "privacy" to either "`public`" or "`unlisted`".

These options work similarly to the way they do on YouTube. Unlisted project URLs will be secret unless you tell people about them or share them. Public projects might be surfaced to other developers.

Upgrading Expo

It isn't strictly necessary to update your app when a new version of Expo is released. New versions of the Expo client are backwards compatible with apps published for previous versions. This means that you can download a new version of the Expo client and open apps that were published for previous versions and they will work perfectly.

That said, each version is better than the last, so you might want to stay up to date to take advantage of new features and performance improvements.

Upgrade guides vary depending on the `sdkVersion`, so follow the guide in the release notes

We post our release notes to [Exposition](#), for example the [v31.0.0 release notes](#). If you're upgrading by more than one major version, we recommend following the upgrade guide for each major version between your current version and your target.

If you are running ExpoKit inside a native project, upgrading will require extra steps. ExpoKit is currently an alpha feature and upgrading difficulty will vary between versions, but there is some information [here](#).

Past Release Notes

- [32.0.0](#)
- [31.0.0](#)
- [30.0.0](#)
- [29.0.0](#)
- [28.0.0](#)
- [27.0.0](#)
- [26.0.0](#)
- [25.0.0](#)
- [24.0.0](#)
- [23.0.0](#)
- [22.0.0](#)
- [21.0.0](#)
- [20.0.0](#)
- [19.0.0](#)
- [18.0.0](#)
- [17.0.0](#)
- [16.0.0](#)
- [15.0.0](#)

Upgrading Expo SDK Walkthrough

If you are a couple of versions behind, upgrading your projects Expo SDK version can be difficult because of the amount of breaking changes and deprecations in each upgrade. Don't worry, here are all the breaking changes in each SDK version upgrade.

Expo only provides support for the last 6 versions since the latest version.

SDK 32

Upgrade from SDK 31

- `app.json`, change `sdkVersion` to `"32.0.0"`,
- In `package.json`, change these dependencies:

```
{  
  "react-native": "https://github.com/expo/react-native/archive/sdk-32.0.0.tar.gz",  
  "expo": "^32.0.0",  
  "react": "16.5.0"  
}
```

- Delete your project's `node_modules` directory and run `npm install` again

Notes

- There are several small breaking API changes with this release. See the [changelog](#) for the full list.

SDK 31

Upgrade from SDK 30

- `app.json`, change `sdkVersion` to `"31.0.0"`,
- In `package.json`, change these dependencies:

```
{  
  "react-native": "https://github.com/expo/react-native/archive/sdk-31.0.0.tar.gz",  
  "expo": "^31.0.0",  
  "react": "16.5.0"  
}
```

- Delete your project's `node_modules` directory and run `npm install` again

Notes

- There are several small breaking API changes with this release. See the [changelog](#) for the full list.

SDK 30

Upgrade from SDK 29

- `app.json`, change `sdkVersion` to `"30.0.0"`,
- In `package.json`, change these dependencies:

```
{  
  "react-native": "https://github.com/expo/react-native/archive/sdk-30.0.0.tar.gz",  
  "expo": "^30.0.0",  
  "react": "16.3.1"  
}
```

- Delete your project's `node_modules` directory and run `npm install` again

Notes

- `Fingerprint` has been renamed to `LocalAuthentication`

SDK 29

Upgrade from SDK 28

- `app.json`, change `sdkVersion` to `"29.0.0"`,
- In `package.json`, change these dependencies:

```
{  
  "react-native": "https://github.com/expo/react-native/archive/sdk-29.0.0.tar.gz",  
  "expo": "^29.0.0",  
  "react": "16.3.1"  
}
```

- Delete your project's `node_modules` directory and run `npm install` again

Notes

- Some field names in `Contacts` were changed. See the [documentation](#) for more information.

SDK 28

Upgrade from SDK 27

- `app.json`, change `sdkVersion` to `"28.0.0"`,
- In `package.json`, change these dependencies:

```
{  
  "react-native": "https://github.com/expo/react-native/archive/sdk-28.0.0.tar.gz",  
  "expo": "^28.0.0",  
  "react": "16.3.1"  
}
```

- Delete your project's `node_modules` directory and run `npm install` again

Notes

- Android apps on all SDK versions now require notification channels for push notifications. This may impact you even if you don't yet use SDK 28. [Read this blog post](#) for all of the necessary information.
- Android app icons are now coerced into adaptive icons. Be sure to test your app icon and supply an adaptive icon if needed. [Read this blog post](#) for all of the necessary information.
- Print has been moved out of DangerZone; update your imports accordingly.

SDK 27

Upgrade from SDK 26

- In app.json, change sdkVersion to "27.0.0"
- In package.json, change these dependencies:

```
{  
  "react-native": "https://github.com/expo/react-native/archive/sdk-27.0.0.tar.gz",  
  "expo": "^27.0.0",  
  "react": "16.3.1"  
}
```

- Delete your project's node_modules directory and run npm install again

Notes

- `View.propTypes` has been removed from React Native, so if your code (or any of your dependent libraries) uses it, that will break. Use `viewPropTypes` instead. We strongly recommend running your app with the dev flag disabled to test whether it's affected by this change.
- We changed the format of `Constants linkingUri` (see Linking changes above), so if your code makes assumptions about this, you should double check that.
- [Camera roll permissions](#) are now required to use `ImagePicker.launchCameraAsync()` and `ImagePicker.launchImageLibraryAsync()`. You can ask for them by calling
`Permissions.askAsync(Permissions.CAMERA_ROLL)`.

SDK 26

Upgrade from SDK 25

- In app.json, change sdkVersion to "26.0.0"
- In package.json, change these dependencies:

```
{  
  "react-native": "https://github.com/expo/react-native/archive/sdk-26.0.0.tar.gz",  
  "expo": "^26.0.0",  
  "react": "16.3.0-alpha.1"  
}
```

- Delete your project's node_modules directory and run npm install again

Notes

- `Expo.Util` is deprecated, functionality has been moved out to `Expo.DangerZone.Localization` and `Expo.Updates`.
- `ios.loadJSInBackgroundExperimental` is now deprecated, use the new `Updates` API instead. The equivalent of this configuration is `updates.fallbackToCacheTimeout: 0`.
- `isRemoteJSEnabled` is also deprecated, use `updates.enabled` instead.
- React Native 0.54 depends on React 16.3.0-alpha.1. React 16.3 deprecates the usage of `componentWillMount`, `componentWillReceiveProps`, and `componentWillUpdate`. These have been replaced with static lifecycle methods: `getDerivedStateFromProps` and `getSnapshotBeforeUpdate`, but only `getDerivedStateFromProps` is available in 16.3.0-alpha.1.
- On iOS, `WebBrowser.dismissBrowser()` promise now resolves with `{type:'dismiss'}` rather than `{type:'dismissed'}` to match Android
- AdMob method name changes. `requestAd` to `requestAdAsync`, `showAd` to `showAdAsync`, `isReady` to `getIsReadyAsync`.
- On iOS, `Contacts urls` was renamed to `urlAddresses` to match Android. [Related commit](#).
- On iOS, calling `Notifications.getExpoPushToken()` will throw an error if you don't have permission to send notifications. We recommend call `Permissions.getAsync(Permissions.NOTIFICATIONS)` and, if needed and you haven't asked before, `Permissions.askAsync(Permissions.NOTIFICATIONS)` before getting push token.
- React native 0.53.0 removed the `TextInput` `autoGrow` prop. [Commit](#).

SDK 25

Upgrade from SDK 24

- In `app.json`, change `sdkVersion` to `"25.0.0"`
- In `package.json`, change these dependencies:

```
{
  "react-native": "https://github.com/expo/react-native/archive/sdk-25.0.0.tar.gz",
  "expo": "^25.0.0",
  "react": "16.2.0"
}
```

- Delete your project's `node_modules` directory and run `npm install` again

Notes

- If you have any scripts in your project that depend on `metro-bundler`, you will need to change those to `metro` ([related commit](#)). A likely place for this is in `rn-cli.config.js`.
- Although not technically part of the SDK, React Navigation is commonly used by Expo users, and it's worth mentioning that on Android React Navigation now properly accounts for the translucent status bar. This may require you to remove code that you have to workaround that (maybe a `paddingTop` somewhere to avoid the content from rendering underneath the status bar). [Read the React Navigation release notes for more information](#). Only applies to `react-navigation@1.0.0-beta.26` and higher.

SDK 24

Upgrade from SDK 23

- In `app.json`, change `sdkVersion` to `"24.0.0"`

- In package.json, change these dependencies:

```
{
  "react-native": "https://github.com/expo/react-native/archive/sdk-24.0.0.tar.gz",
  "expo": "^24.0.0",
  "react": "16.0.0"
}
```

- Delete your project's node_modules directory and run npm install again

Notes

The following APIs have been removed after being deprecated for a minimum of 2 releases:

- `Expo.LegacyAsyncStorage`
- `Expo.Font.style`
- Passing an object into `Expo.SQLite.openDatabase()` instead of separate arguments is no longer supported.

SDK 23

Upgrade from SDK 22

- In app.json, change sdkVersion to `"23.0.0"`
- In package.json, change these dependencies:

```
{
  "react-native": "https://github.com/expo/react-native/archive/sdk-23.0.0.tar.gz",
  "expo": "^23.0.0",
  "react": "16.0.0"
}
```

- Delete your project's node_modules directory and run npm install again

Notes

- React Native no longer supports nesting components inside of `<Image>`—some developers used this to use an image as a background behind other views. To fix this in your app, replace the `Image` component anywhere where you are nesting views inside of it with the `ImageBackground` component. [See a Snack example here.](#)
- React Native now defaults `enableBabelRCLookup` (recursive) to false in Metro bundler (the packager used by React Native / Expo). This is unlikely to cause any problems for your application—in our case, this lets us remove a script to delete nested `.babelrc` files from `node_modules` in our postinstall. If you run into transform errors when updating your app, [read this commit message for more information](#) and to see how to opt-in to the old behavior.

SDK 22

Upgrade from SDK 21

- In app.json, change sdkVersion to `"22.0.0"`

- In package.json, change these dependencies:

```
{
  "react-native": "https://github.com/expo/react-native/archive/sdk-22.0.1.tar.gz",
  "expo": "^22.0.0",
  "react": "16.0.0-beta.5"
}
```

- Delete your project's node_modules directory and run npm install again

Notes

Metro Bundler (the React Native packager) now errors (instead of silently ignoring) dynamic requires. In particular this breaks an older version of moment.js if you were using that (or indirectly depending on it).

- This is a known issue with Metro [which is being tracked on the project's Github issues](#).
- If you use moment.js in your app, [you may have success with this fix](#).

Several deprecated APIs have been removed. All of these APIs printed warning messages in previous releases:

- `Expo.Notifications.getExponentPushToken` is now `Expo.Notifications.getExpoPushToken`.
- `Expo.AdMob.AdMobInterstitials.tryShowNewInterstitial` has been removed in favor of `requestAd` and `showAd`.
- `Expo.Segment.initializeAndroid` and `initializeiOS` have been removed in favor of `Expo.Segment.initialize`.
- The `tintEffect` prop of `Expo.BlurView` has been removed in favor of the `tint` prop.
- `Expo.SecureStore.setValueWithKeyAsync`, `getValueWithKeyAsync`, and `deleteValueWithKeyAsync` are now `setItemAsync`, `getItemAsync`, and `deleteItemAsync`. The order of arguments to `setValueWithKeyAsync` changed from `(value, key)` to `(key, value)`.
- The `callback` prop of `Expo.Video` and `Expo.Audio` is now `onPlaybackStatusUpdate`. This is not a breaking change yet but we plan to remove `Legacy AsyncStorage` in SDK 24. If you, or any libraries that you use, use `View.propTypes.style` you will need to change that to `ViewPropTypes.style`.

If you have not yet updated your imports of `PropTypes` as warned in deprecation warnings in previous releases, you will need to do this now. Install the `prop-types` package and `import PropTypes from 'prop-types'`; instead of `import { PropTypes } from React;`! Similarly, if you depend on `React.createClass`, you will need to install the `create-react-class` package and `import createReactClass from 'create-react-class'`; [as described in the React documentation](#).

SDK 21

Upgrade from SDK 20

- In app.json, change sdkVersion to `"21.0.0"`
- In package.json, change these dependencies:

```
{
  "react-native": "https://github.com/expo/react-native/archive/sdk-21.0.2.tar.gz",
  "expo": "^21.0.0",
  "react": "16.0.0-alpha.12"
}
```

- Delete your project's node_modules directory and run npm install again

Notes

Camera

- The `takePicture` function is now called `takePictureAsync` and now returns an object with many keys, instead of just returning the URI. The URI is available under the `uri` key of the returned object.
- Previously this function would return a value like: `"file://path/to/your/file.jpg"`
- And will now return an object like: `{ "uri": "file://path/to/your/file.jpg" }`

Secure Store

- `setValueWithKeyAsync` → `setItemAsync` : The order of the arguments has been reversed to match typical key-value store APIs. This function used to expect `(value, key)` and now expects `(key, value)`.
- `getValueWithKeyAsync` → `getItemAsync` : Trying to retrieve an entry that doesn't exist returns `null` instead of throwing an error.
- `deleteValueWithKeyAsync` → `deleteItemAsync`

Payments

- We'd previously announced Stripe support on iOS as part of our experimental DangerZone APIs. The Payments API was using the Stripe SDK on iOS. We learned that Apple sometimes rejects apps which contain the Stripe SDK but don't offer anything for sale. To help your App Review process go more smoothly, we've decided to remove the Stripe SDK and experimental Payments API from apps built with the Expo standalone builder. We're still excited to give developers a way to let users pay for goods when they need to and we'll announce more ways to do so shortly.

Linking

Introduction

Every good website is prefixed with `https://`, and `https` is what is known as a *URL scheme*. Insecure websites are prefixed with `http://`, and `http` is the URL scheme. Let's call it scheme for short.

To navigate from one website to another, you can use an anchor tag (`<a>`) on the web. You can also use JavaScript APIs like `window.history` and `window.location`.

In addition to `https`, you're likely also familiar with the `mailto` scheme. When you open a link with the `mailto` scheme, your operating system will open an installed mail application. If you have more than one mail application installed then your operating system may prompt you to pick one. Similarly, there are schemes for making phone calls and sending SMS'. Read more about [built-in URL schemes](#) below.

`https` and `http` are handled by your browser, but it's possible to link to other applications by using different url schemes. For example, when you get a "Magic Link" email from Slack, the "Launch Slack" button is an anchor tag with an href that looks something like: `slack://secret/magic-login/other-secret`. Like with Slack, you can tell the operating system that you want to handle a custom scheme. Read more about [configuring a scheme](#). When the Slack app opens, it receives the URL that was used to open it and can then act on the data that is made available through the url -- in this case, a secret string that will log the user in to a particular server. This is often referred to as **deep linking**. Read more about [handling deep links into your app](#).

Deep linking with scheme isn't the only linking tool available to you. It is often desirable for regular HTTPS links to open your application on mobile. For example, if you're sending a notification email about a change to a record, you don't want to use a custom URL scheme in links in the email, because then the links would be broken on desktop. Instead, you want to use a regular HTTPS link such as `https://www.myapp.io/records/123`, and on mobile you want that link to open your app. iOS terms this concept "universal links" and Android calls it "deep links"; Expo supports these links on both platforms (with some [configuration](#)). Expo also supports deferred deep links with [Branch](#).

Linking from your app to other apps

Built-in URL Schemes

As mentioned in the introduction, there are some URL schemes for core functionality that exist on every platform. The following is a non-exhaustive list, but covers the most commonly used schemes.

Scheme	Description	iOS	Android
<code>mailto</code>	Open mail app, eg: <code>mailto: support@expo.io</code>		
<code>tel</code>	Open phone app, eg: <code>tel:+123456789</code>		
<code>sms</code>	Open SMS app, eg: <code>sms:+123456789</code>		
<code>https / http</code>	Open web browser app, eg: <code>https://expo.io</code>		

Opening links from your app

There is no anchor tag in React Native, so we can't write ``, instead we have to use `Linking.openURL`.

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

Linking.openURL('https://expo.io');
```

Usually you don't open a URL without it being requested by the user -- here's an example of a simple `Anchor` component that will open a URL when it is pressed.

```
import { Linking, Text } from 'react-native';

export default class Anchor extends React.Component {
  _handlePress = () => {
    Linking.openURL(this.props.href);
    this.props.onPress && this.props.onPress();
  };

  render() {
    return (
      <Text {...this.props} onPress={this._handlePress}>
        {this.props.children}
      </Text>
    );
  }
}

// <Anchor href="https://google.com">Go to Google</Anchor>
// <Anchor href="mailto://support@expo.io">Email support</Anchor>
```

Using `Expo.WebBrowser` instead of `Linking` for opening web links

The following example illustrates the difference between opening a web link with `Expo.WebBrowser.openBrowserAsync` and React Native's `Linking.openURL`. Often `WebBrowser` is a better option because it's a modal within your app and users can easily close out of it and return to your app.

Opening links to other apps

If you know the custom scheme for another app you can link to it. Some services provide documentation for deep linking, for example the [Lyft deep linking documentation](#) describes how to link directly to a specific pickup location and destination:

```
lyft://ridetype?id=lyft&pickup[latitude]=37.764728&pickup[longitude]=-122.422999&destination[latitude]=37.7763592&des-
tination[longitude]=-122.4242038
```

It's possible that the user doesn't have the Lyft app installed, in which case you may want to open the App / Play Store, or let them know that they need to install it first. We recommend using the library [react-native-app-link](#) for these cases.

On iOS, `Linking.canOpenURL` requires additional configuration to query other apps' linking schemes. You can use the `ios.infoPlist` key in your `app.json` to specify a list of schemes your app needs to query. For example:

```
"infoPlist": {
  "LSApplicationQueriesSchemes": ["lyft"]
}
```

If you don't specify this list, `Linking.canOpenURL` may return `false` regardless of whether the device has the app installed. Note that this configuration can only be tested in standalone apps, because it requires native changes that will not be applied when testing in Expo Client.

Linking to your app

In the Expo client

Before continuing it's worth taking a moment to learn how to link to your app within the Expo client. The Expo client uses the `exp://` scheme, but if we link to `exp://` without any address afterwards, it will open the app to the main screen.

In development, your app will live at a url like `exp://wg-qka.community.app.exp.direct:80`. When it's deployed, it will be at a URL like `exp://exp.host/@community/with-webbrowser-redirect`. If you create a website with a link like `Open my project`, then open that site on your device and click the link, it will open your app within the Expo client. You can link to it from another app by using `Linking.openURL` too.

In a standalone app

To link to your standalone app, you need to specify a scheme for your app. You can register for a scheme in your `app.json` by adding a string under the `scheme` key:

```
{
  "expo": {
    "scheme": "myapp"
  }
}
```

Once you build your standalone app and install it to your device, you will be able to open it with links to `myapp://`.

If your app is ejected, note that like some other parts of `app.json`, changing the `scheme` key after your app is already ejected will not have the desired effect. If you'd like to change the deep link scheme in your ejected app, see [this guide](#).

Expo.Linking module

To save you the trouble of inserting a bunch of conditionals based on the environment that you're in and hardcoding urls, we provide some helper methods in our extension of the `Linking` module. When you want to provide a service with a url that it needs to redirect back into your app, you can call `Expo.Linking.makeUrl()` and it will resolve to the following:

- *Published app in Expo client:* `exp://exp.host/@community/with-webbrowser-redirect`
- *Published app in standalone:* `myapp://`
- *Development:* `exp://wg-qka.community.app.exp.direct:80`

You can also change the returned url by passing optional parameters into `Expo.Linking.makeUrl()`. These will be used by your app to receive data, which we will talk about in the next section.

Handling links into your app

There are two ways to handle URLs that open your app.

1. If the app is already open, the app is foregrounded and a Linking event is fired

You can handle these events with `Linking.addEventListerner('url', callback)`.

2. If the app is not already open, it is opened and the url is passed in as the initialURL

You can handle these events with `Linking.getInitialURL` -- it returns a `Promise` that resolves to the url, if there is one.

See the examples below to see these in action.

Passing data to your app through the URL

To pass some data into your app, you can append it as a path or query string on your url.

`Expo.Linking.makeUrl(path, queryParams)` will construct a working url automatically for you. You can use it like this:

```
let redirectUrl = Expo.Linking.makeUrl('path/into/app', { hello: 'world', goodbye: 'now' });
```

This would return something like `myapp://path/into/app?hello=world&goodbye=now` for a standalone app.

When your app is opened using the deep link, you can parse the link with `Expo.Linking.parse()` to get back the path and query parameters you passed in.

When [handling the URL that is used to open/foreground your app](#), it would look something like this:

```
_handleUrl = url => {
  this.setState({ url });
  let { path, queryParams } = Expo.Linking.parse(url);
  alert(`Linked to app with path: ${path} and data: ${JSON.stringify(queryParams)}`);
};
```

If you opened a URL like `myapp://path/into/app?hello=world&goodbye=now`, this would alert `Linked to app with path: path/into/app and data: {hello: 'world', goodbye: 'now'}`.

Example: linking back to your app from WebBrowser

The example project [examples/with-webbrowser-redirect](#) demonstrates handling redirects from `WebBrowser` and taking data out of the query string. Try it out in [Expo](#).

Example: using linking for authentication

A common use case for linking to your app is to redirect back to your app after opening a `WebBrowser`. For example, you can open a web browser session to your sign in screen and when the user has successfully signed in, you can have your website redirect back to your app by using the scheme and appending the authentication token and other data to the URL.

Note: if try to use `Linking.openURL` to open the web browser for authentication then your app may be rejected by Apple on the grounds of a bad or confusing user experience. `WebBrowser.openBrowserAsync` opens the browser window in a modal, which looks and feels good and is Apple approved.

To see a full example of using `WebBrowser` for authentication with Facebook, see [examples/with-facebook-auth](#). Currently Facebook authentication requires that you deploy a small webserver to redirect back to your app (as described in the example) because Facebook does not let you redirect to custom schemes, Expo is working on a solution to make this easier for you. [Try it out in Expo](#).

Another example of using `WebBrowser` for authentication can be found at [expo/auth0-example](#).

Universal/deep links (without a custom scheme)

It is often desirable for regular HTTPS links (without a custom URL scheme) to directly open your app on mobile devices. This allows you to send notification emails with links that work as expected in a web browser on desktop, while opening the content in your app on mobile. iOS refers to this concept as "universal links" while Android calls it "deep links" (but in this section, we are specifically discussing deep links that do not use a custom URL scheme).

Universal links on iOS

To implement universal links on iOS, you must first set up verification that you own your domain. This is done by serving an Apple App Site Association (AASA) file from your webserver. The AASA must be served from `/apple-app-site-association` or `/.well-known/apple-app-site-association` (with no extension). The AASA contains JSON which specifies your Apple app ID and a list of paths on your domain that should be handled by your mobile app. For example, if you want links of the format `https://www.myapp.io/records/123` to be opened by your mobile app, your AASA would have the following contents:

```
{
  "applinks": {
    "apps": [],
    "details": [
      {
        "appID": "LKWJEF.io.myapp.example",
        "paths": ["/records/*"]
      }
    ]
  }
}
```

This tells iOS that any links to `https://www.myapp.io/records/*` (with wildcard matching for the record ID) should be opened directly by your mobile app. See [Apple's documentation](#) for further details on the format of the AASA.

Branch provides an [AASA validator](#) which can help you confirm that your AASA is correctly deployed and has a valid format.

Note that iOS will download your AASA when your app is first installed and when updates are installed from the App Store, but it will not refresh any more frequently. If you wish to change the paths in your AASA for a production app, you will need to issue a full update via the App Store so that all of your users' apps re-fetch your AASA and recognize the new paths.

After deploying your AASA, you must also configure your app to use your associated domain. First, you need to add the `associatedDomains` configuration to your `app.json`. Second, you need to edit your App ID on the Apple developer portal and enable the "Associated Domains" application service. You will also need to regenerate your provisioning profile after adding the service to the App ID.

At this point, opening a link on your mobile device should now open your app! If it doesn't, re-check the previous steps to ensure that your AASA is valid, the path is specified in the AASA, and you have correctly configured your App ID in the Apple developer portal. Once you've got your app opening, move to the [Handling links into your app](#) section for details on how to handle the inbound link and show the user the content they requested.

Deep links on Android

Implementing deep links on Android (without a custom URL scheme) is somewhat simpler than on iOS. You simply need to add `intentFilters` to the [Android section](#) of your `app.json`. The following basic configuration will cause your app to be presented in the standard Android dialog as an option for handling any record links to

`myapp.io`:

```
"intentFilters": [
  {
    "action": "VIEW",
    "data": [
      {
        "scheme": "https",
        "host": "*.myapp.io"
        "pathPrefix": "/records"
      },
    ],
    "category": [
      "BROWSEABLE",
      "DEFAULT"
    ]
  }
]
```

It may be desirable for links to your domain to always open your app (without presenting the user a dialog where they can choose the browser or a different handler). You can implement this with Android App Links, which use a similar verification process as Universal Links on iOS. First, you must publish a JSON file at `/.well-known/assetlinks.json` specifying your app ID and which links should be opened by your app. See [Android's documentation](#) for details about formatting this file. Second, add `"autoVerify": true` to the intent filter in your `app.json`; this tells Android to check for your `assetlinks.json` on your server and register your app as the automatic handler for the specified paths:

```
"intentFilters": [
  {
    "action": "VIEW",
    "autoVerify": true,
    "data": [
      {
        "scheme": "https",
        "host": "*.myapp.io"
        "pathPrefix": "/records"
      },
    ],
    "category": [
      "BROWSEABLE",
      "DEFAULT"
    ]
  }
]
```

When to *not* use deep links

This is the easiest way to set up deep links into your app because it requires a minimal amount of configuration.

The main problem is that if the user does not have your app installed and follows a link to your app with its custom scheme, their operating system will indicate that the page couldn't be opened but not give much more information. This is not a great experience. There is no way to work around this in the browser.

Additionally, many messaging apps do not autolink URLs with custom schemes -- for example, `exp://exp.host/@community/native-component-list` might just show up as plain text in your browser rather than as a link (<exp://exp.host/@community/native-component-list>).

An example of this is Gmail which strips the href property from links of most apps, a trick to use is to link to a regular https url instead of your app's custom scheme, this will open the user's web browser. Browsers do not usually strip the href property so you can host a file online that redirects the user to your app's custom schemes.

So instead of linking to `example://path/into/app`, you could link to <https://example.com/redirect-to-app.html> and `redirect-to-app.html` would contain the following code:

```
<script>window.location.replace("example://path/into/app");</script>
```

How Expo Works

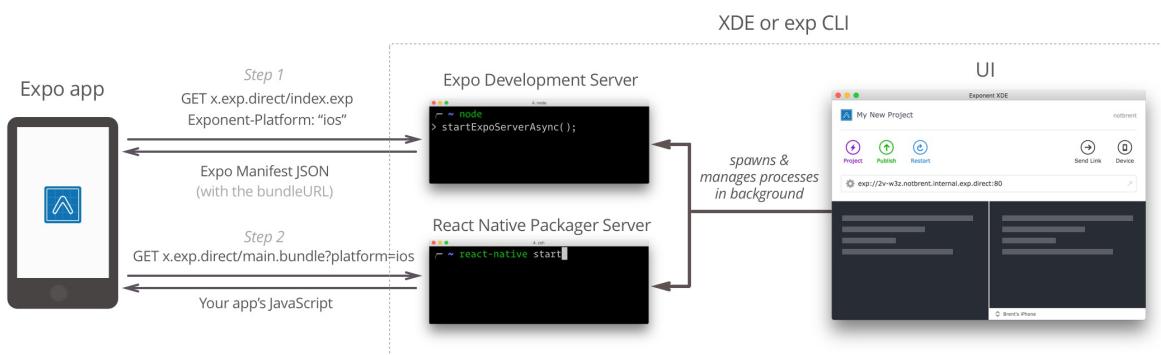
While it's certainly not necessary to know any of this to use Expo, many engineers like to know how their tools work. We'll walk through a few key concepts here, including:

- Local development of your app
- Publishing/deploying a production version of your app
- How Expo manages changes to its SDK
- Opening Expo apps offline

You can also browse the source, fork, hack on and contribute to the Expo tooling on github/@expo.

Serving an Expo project for local development

There are two pieces here: the Expo app and Expo CLI. When you start an app with Expo CLI, it spawns and manages two server processes in the background: the Expo Development Server and the React Native Packager Server.



Note: Expo CLI also spawns a tunnel process, which allows devices outside of your LAN to access the the above servers without you needing to change your firewall settings. If you want to learn more, see [ngrok](#).

Expo Development Server

This server is the endpoint that you hit first when you type the URL into the Expo app. Its purpose is to serve the **Expo Manifest** and provide a communication layer between Expo CLI and the Expo app on your phone or simulator.

Expo Manifest

The following is an example of a manifest being served through Expo CLI. The first thing that you should notice is there are a lot of identical fields to `app.json` (see the [Configuration with app.json](#) section if you haven't read it yet). These fields are taken directly from that file -- this is how the Expo app accesses your configuration.

```
{  
  "name": "My New Project",  
  "description": "A starter template",  
  "slug": "my-new-project",  
  "sdkVersion": "18.0.0",  
  "version": "1.0.0",  
}
```

```

"orientation":"portrait",
"primaryColor": "#cccccc",
"icon": "https://s3.amazonaws.com/exp-brand-assets/ExponentEmptyManifest_192.png",
"notification": {
  "icon": "https://s3.amazonaws.com/exp-us-standard/placeholder-push-icon.png",
  "color": "#000000"
},
"loading": {
  "icon": "https://s3.amazonaws.com/exp-brand-assets/ExponentEmptyManifest_192.png"
},
"entryPoint": "node_modules/expo/AppEntry.js",
"packagerOpts": {
  "hostType": "tunnel",
  "dev": false,
  "strict": false,
  "minify": false,
  "urlType": "exp",
  "urlRandomness": "2v-w3z",
  "lanType": "ip"
},
"xde": true,
"developer": {
  "tool": "xde"
},
"bundleUrl": "http://packager.2v-w3z.notbrent.internal.exp.direct:80/apps/new-project-template/main.bundle?platform=ios&dev=false&strict=false&minify=false&hot=false&includeAssetFileHashes=true",
"debuggerHost": "packager.2v-w3z.notbrent.internal.exp.direct:80",
"mainModuleName": "main",
"logUrl": "http://2v-w3z.notbrent.internal.exp.direct:80/logs"
}

```

Every field in the manifest is some configuration option that tells Expo what it needs to know to run your app. The app fetches the manifest first and uses it to show your app's loading icon that you specified in `app.json`, then proceeds to fetch your app's JavaScript at the given `bundleUrl` -- this URL points to the React Native Packager Server.

In order to stream logs to Expo CLI, the Expo SDK intercepts calls to `console.log`, `console.warn`, etc. and posts them to the `logUrl` specified in the manifest. This endpoint is on the Expo Development Server.

React Native Packager Server

If you use React Native without Expo, you would start the packager by running `react-native start` in your project directory. Expo starts this up for you and pipes `STDOUT` to Expo CLI. This server has two purposes.

The first is to serve your app JavaScript compiled into a single file and translating any JavaScript code that you wrote which isn't compatible with your phone's JavaScript engine. JSX, for example, is not valid JavaScript -- it is a language extension that makes working with React components more pleasant and it compiles down into plain function calls -- so `<HelloWorld />` would become `React.createElement(HelloWorld, {}, null)` (see [JSX in Depth](#) for more information). Other language features like `async/await` are not yet available in most engines and so they need to be compiled down into JavaScript code that will run on your phone's JavaScript engine, JavaScriptCore.

The second purpose is to serve assets. When you include an image in your app, you will use syntax like `<Image source={require('./assets/example.png')} />`, unless you have already cached that asset you will see a request in the Expo CLI logs like: `<START> processing asset request my-project/assets/example@3x.png`. Notice that it serves up the correct asset for your screen DPI, assuming that it exists.

Publishing/Deploying an Expo app in Production

When you publish an Expo app, we compile it into a JavaScript bundle with production flags enabled. That is, we minify the source and we tell the React Native packager to build in production mode (which in turn sets `_DEV_` to `false` amongst other things). After compilation, we upload that bundle, along with any assets that it requires (see [Assets](#)) to CloudFront. We also upload your [Manifest](#) (including most of your `app.json` configuration) to our server. When publishing is complete, we'll give you a URL to your app which you can send to anybody who has the Expo client.

Note: By default, all Expo projects are `unlisted`, which means that publishing does not make it publicly searchable or discoverable anywhere. It is up to you to share the link. You can change this setting in [app.json](#).

As soon as the publish is complete, the new version of your code is available to all your existing users. They'll download the updated version next time they open the app or refresh it, provided that they have a version of the Expo client that supports the `sdkVersion` specified in your `app.json`.

Updates are handled differently on iOS and Android. On Android, updates are downloaded in the background. This means that the first time a user opens your app after an update they will get the old version while the new version is downloaded in the background. The second time they open the app they'll get the new version. On iOS, updates are downloaded synchronously, so users will get the new version the first time they open your app after an update.

Note: To package your app for deployment on the Apple App Store or Google Play Store, see [Building Standalone Apps](#). Each time you update the SDK version you will need to rebuild your binary.

SDK Versions

The `sdkVersion` of an Expo app indicates what version of the compiled ObjC/Java/C layer of Expo to use. Each `sdkVersion` roughly corresponds to a release of React Native plus the Expo libraries in the SDK section of these docs.

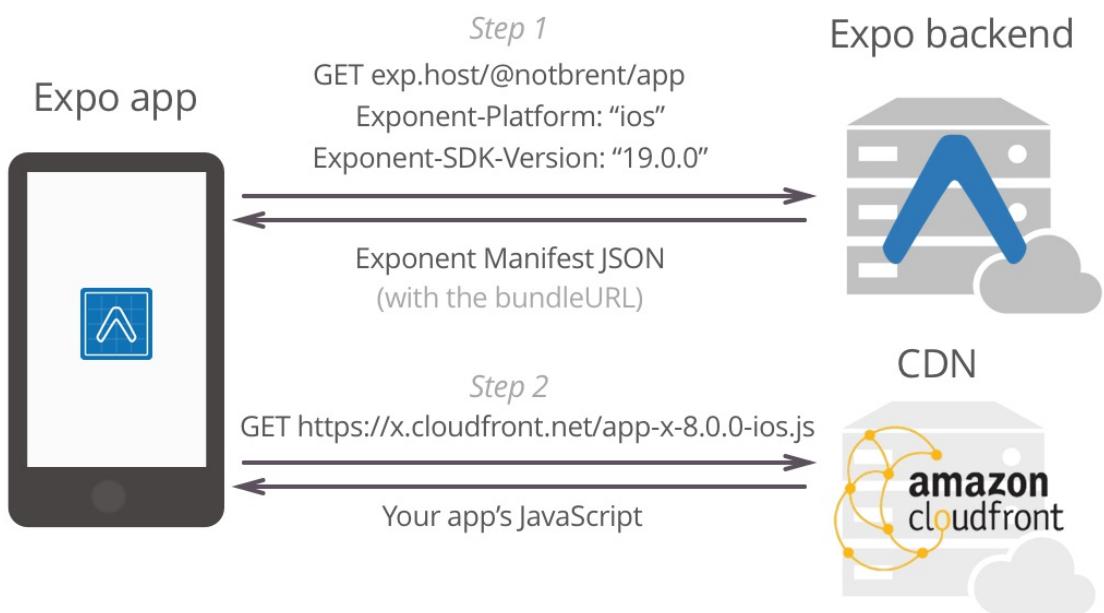
The Expo client app supports many versions of the Expo SDK, but an app can only use one at a time. This allows you to publish your app today and still have it work a year from now without any changes, even if we have completely revamped or removed an API your app depends on in a new version. This is possible because your app will always be running against the same compiled code as the day that you published it.

If you publish an update to your app with a new `sdkVersion`, if a user has yet to update to the latest Expo client then they will still be able to use the previous `sdkVersion`.

Note: It's likely that eventually we will formulate a policy for how long we want to keep around `sdkVersions` and begin pruning very old versions of the sdk from the client, but until we do that, everything will remain backwards compatible.

Opening a deployed Expo app

The process is essentially the same as opening an Expo app in development, only now we hit an Expo server to get the manifest, and manifest points us to CloudFront to retrieve your app's JavaScript.



Opening Expo Apps Offline

The Expo client will automatically cache the most recent version of every app it has opened. When you try to open an Expo app, it will always try and fetch the latest version, but if that fails for whatever reason (including being totally offline) then it will load the most recent cached version.

If you build a standalone app with Expo, that standalone binary will also ship with a "pre-cached" version of your JavaScript so that it can cold launch the very first time with no internet. Continue reading for more information about standalone apps.

Standalone Apps

You can also package your Expo app into a standalone binary for submission to the Apple iTunes Store or Google Play.

Under the hood, it's a modified version of the Expo client which is designed only to load a single URL (the one for your app) and which will never show the Expo home screen or brand. For more information, see [Building Standalone Apps](#).

Glossary of terms

app.json

`app.json` is a file that exists for every Expo project and it is used to configure your project, for example the name, icon, and splash screen. [Read more in "Configuration with app.json"](#)

create-react-native-app

Formerly the React Native equivalent of [create-react-app](#). This has since been replaced with `expo-cli`.

detach

The term "detach" was previously used in Expo to mean [ejecting](#) your app to use [ExpoKit](#).

eject

The term "eject" was popularized by [create-react-app](#), and it is used in Expo to describe leaving the cozy comfort of the standard Expo development environment, where you do not have to deal with build configuration or native code. When you "eject" from Expo, you have two choices:

- *Eject to ExpoKit*, where you get the native projects along with [ExpoKit](#), so you can continue building your project using the Expo APIs but your workflow now is the same as if you were building a React Native application without Expo. [Read more in "Ejecting to ExpoKit"](#).
- *Eject to plain React Native*, where you take a more extreme step than just ejecting to [ExpoKit](#) -- you lose access to Expo APIs and completely leave the Expo environment. [Read more about ejecting](#).

Emulator

Emulator is used to describe software emulators of Android devices on your computers. Typically iOS emulators are referred to as [Simulators](#).

Experience

A synonym for app that usually implies something more single-use and smaller in scope, sometimes artistic and whimsical.

Expo CLI

The command-line tool for working with Expo. [Read more](#).

Expo Client

The iOS and Android app that runs Expo apps. When you want to run your app outside of the Expo Client and deploy it to the App and/or Play stores, you can build a [Standalone App](#).

Expo Dev Tools

Expo Developer Tools is a web browser based UI included in [Expo CLI](#).

Expo SDK

The Expo SDK provides access to device/system functionality such as camera, push notification, contacts, file system, and more. Scroll to the SDK API reference in the documentation navigation to see a full list of APIs and to explore them. [Read more about the Expo SDK](#). [Find it on Github](#).

ExpoKit

ExpoKit is an Objective-C and Java library that allows you to use the [Expo SDK](#) and platform and your existing Expo project as part of a larger standard native project — one that you would normally create using Xcode, Android Studio, or `react-native init`. [Read more](#).

iOS

The operating system used on iPhone, iPad, and Apple TV. Expo currently runs on iOS for iPhone and iPad.

Linking

Linking can mean [deep linking](#) into apps similar to how you link to websites on the web or [linking native libraries into your ejected ExpoKit app](#).

Manifest

An Expo app manifest is similar to a [web app manifest](#) - it provides information that Expo needs to know how to run the app and other relevant data. [Read more in "How Expo Works"](#).

Native Directory

The React Native ecosystem has thousands of libraries. Without a purpose-built tool, it's hard to know what the libraries are, to search through them, to determine the quality, try them out, and filter out the libraries that won't work for your project (some don't work with Expo, some don't work with Android or iOS). [Native Directory](#) is a website that aims to solve this problem, we recommend you use it to find packages to use in your projects.

npm

[npm](#) is a package manager for JavaScript and the registry where the packages are stored. An alternative package manager, which we use internally at Expo, is [yarn](#).

Over the Air updates

Traditionally, apps for iOS and Android are updated by submitting an updated binary to the App and Play stores. Over the Air (OTA) updates allow you to push an update to your app without the overhead of submitting a new release to the stores. [Read more in "Publishing"](#).

Package Manager

Automates the process of installing, upgrading, configuring, and removing libraries, also known as dependencies, from your project. See [npm](#) and [yarn](#).

Publish

We use the word "publish" as a synonym for "deploy". When you publish an app, it becomes available at a persistent URL from the Expo client, or in the case of [Standalone apps](#), it updates the app [over the air](#).

React Native

"React Native lets you build mobile apps using only JavaScript. It uses the same design as React, letting you compose a rich mobile UI from declarative components." [Read more](#).

Shell app

Another term we occasionally use for [Standalone app](#).

Simulator

An emulator for iOS devices that you can run on macOS (or in [Snack](#)) to work on your app without having to have a physical device handy.

Slug

We use the word "slug" in [app.json](#) to refer to the name to use for your app in its url. For example, the [Native Component List](#) app lives at <https://expo.io/@community/native-component-list> and the slug is native-component-list.

Snack

[Snack](#) is an in-browser development environment where you can build Expo experiences without installing any tools on your phone or computer.

Standalone app

An application binary that can be submitted to the iOS App Store or Android Play Store. [Read more in "Building Standalone Apps"](#).

XDE

XDE was a desktop tool with a graphical user interface (GUI) for working with Expo projects. It's been replaced by [Expo CLI](#), which now provides both command line and web interfaces.

yarn

A package manager for JavaScript. [Read more](#)

Expo & "Create React Native App"

WARNING

[Create React Native App](#) has been replaced by the Expo-CLI. If you've already created your project with CRNA, you can read about migrating from CRNA to Expo-CLI [here](#).

Important Notes

- Expo CLI is a tool based on CRNA, made by the same team.
- It has all the same features, plus some additional benefits.
- Like CRNA, Expo CLI does not require an Expo user account.
- The `create-react-native-app` command will continue to work.

Why has Expo-CLI replaced CRNA?

- Just one tool to learn: previously developers would start with CRNA and then switch to `exp` or `XDE` for additional features like standalone builds. Expo CLI is as easy to get started with as CRNA, but also supports everything previously offered by these separate tools.
- Less confusing options: CRNA apps have always been loaded using the Expo app and able to use the Expo APIs in addition to the core React Native APIs. Users are sometimes confused about the differences between plain React Native, CRNA and Expo apps created with tools like `exp` or `XDE`. Installing the `expo-cli` package will make it clearer the additional functionality is provided by Expo.
- Developer experience: Expo CLI is ahead of CRNA in terms of features and developer experience, and we're continuously improving it.
- Maintenance: having these two projects as separate codebases requires more maintenance and CRNA has previously fell behind because of this. A single codebase helps us keep it up-to-date and fix issues as fast as possible.

App Icons

Your app's icon is what users see on the home screen of their devices, as well as in the App Store and Play Store. This is one topic where platform differences matter, and requirements can be strict. This guide offers details on how to make sure your App Icon looks as good as possible on all devices.

Configuring your App's Icon

The most straightforward way to provide an icon for your app is to provide the `icon` key in `app.json`. If you want to do the minimum possible, this key alone is sufficient. However, Expo also accepts platform-specific keys under `ios.icon` and `android.icon`. If either of these exist, they will take priority over the base `icon` key on their respective platform. Further customization of the Android icon is possible using the `android.adaptiveIcon` key, which will override both of the previously mentioned settings. Most production-quality apps will probably want to provide something slightly different between iOS and Android.

Icon Best Practices

iOS

- The icon you use for iOS should follow the [Apple Human Interface Guidelines](#) for iOS Icons.
- Use a png file.
- 1024x1024 is a good size. The Expo [build service](#) will generate the other sizes for you. The largest size it generates is 1024x1024.
- The icon must be exactly square, i.e. a 1023x1024 icon is not valid.
- Make sure the icon fills the whole square, with no rounded corners or other transparent pixels. The operating system will mask your icon when appropriate.

Android

- The Android Adaptive Icon is formed from two separate layers -- a foreground image and a background color or image. This allows the OS to mask the icon into different shapes and also support visual effects.
- The design you provide should follow the [Android Adaptive Icon Guidelines](#) for launcher icons.
- Use png files.
- The default background color is white; to specify a different background color, use the `android.adaptiveIcon.backgroundColor` field. You can instead specify a background image using the `android.adaptiveIcon.backgroundImage` field; ensure that it has the same dimensions as your foreground image.
- You may also want to provide a separate icon for older Android devices that do not support Adaptive Icons; you can do so with the `android.icon` field. This single icon would probably be a combination of your foreground and background layers.
- You may still want to follow some of the [Apple best practices](#) to ensure your icon looks professional, such as testing your icon on different wallpapers, and avoiding text besides your product's wordmark.
- Provide something that's at least 512x512 pixels. Since you already need 1024x1024 for iOS, it won't hurt to just provide that here as well.

Expo Client and Web

- If your app contains `privacy: public` in `app.json`, it will show up on your expo.io profile. We will mask your icon to have rounded corners in that circumstance, so if it already looks reasonable on iOS, it will probably look good here as well.

Assets

Images, fonts, videos, sounds, any other file that your app depends on that is not JavaScript is considered to be an *asset*. Just as on the web, assets are fetched or streamed over HTTP on demand. This is different from your typical mobile app, where assets are bundled with your application binary.

However, there is a distinction in Expo between an asset that you use with the `require` syntax because they are available at build time on your local filesystem, eg: `<Image source={require('./assets/images/example.png')} />`, and web images that you refer to by a web URL, eg: `<Image source={{uri: 'http://yourwebsite.com/logo.png'}} />`. We can make no guarantees about the availability of the images that you refer to with a web URI because we don't manage those assets. Additionally, we don't have the same amount of information about arbitrary web URIs: when your assets are available on the local filesystem, the packager is able to read some metadata (width, height, for example) and pass that through to your app, so you actually don't need to specify a width and height, for example. When specifying a remote web URL, you will need to explicitly specify a width and height, or it will default to 0x0. Lastly, as you will see later, caching behaviour is different in both cases.

The following is an explanation of the former type of assets: those that you have on your filesystem at build time. In the latter case, it is assumed that you are familiar with how to upload an image to somewhere on the web where it can be accessed by any web or mobile app.

Where assets live

In development

While you're working on a local copy of your project, assets are served from your local filesystem and are integrated with the JavaScript module system. So if I want to include an image I can `require` it, like I would if it were JavaScript code: `require('./assets/images/example.png')`. The only difference here is that we need to specify an extension -- without an extension, the module system will assume it is a JavaScript file. This statement evaluates at compile time to an object that includes metadata about the asset that can be consumed by the `Image` component to fetch it and render it: `<Image source={require('./assets/images/example.png')} />`

In production

Each time you publish your app, Expo will upload your assets to Amazon CloudFront, a blazing fast CDN. It does this in an intelligent way to ensure your deploys remain fast: if an asset has not changed since your previous deploy, it is skipped. You don't have to do anything for this to work, it is all automatically handled by Expo.

Performance

Some assets are too important to start your app without. Fonts often fall into this category. On the web the font loading problem is known by several acronyms: FOUT, FOIT, and FOFT, which stand for Flash of Unstyled Text, Flash of Invisible Text, and Flash of Faux Text ([read more here](#)). The default behaviour with the icon-font-powered `@expo/vector-icons` icons is a FOIT on first load, and on subsequent loads the font will be automatically cached. Users have higher standards for mobile than web, so you might want to take it a step further by preloading and caching the font and important images during the initial loading screen.

Error Handling

This guide details a few strategies available for reporting and recovering from errors in your project.

Handling Fatal JS Errors

If your app encounters a fatal JS error, Expo will report the error differently depending on whether your app is in development or production.

In Development: If you're serving your app from Expo CLI, the fatal JS error will be reported to the [React Native RedBox](#) and no other action will be taken.

In Production: If your published app encounters a fatal JS error, Expo will immediately reload your app. If the error happens very quickly after reloading, Expo will show a generic error screen with a button to try manually reloading.

Expo can also report custom information back to you after your app reloads. If you use `ErrorRecovery.setRecoveryProps`, and the app later encounters a fatal JS error, the contents of that method call will be passed back into your app's initial props upon reloading. See [Expo.ErrorRecovery](#).

Tracking JS Errors

We recommend using [Sentry](#) to track JS errors in production and configuring our post-publish hook to keep your source maps up to date.

What about Native Errors?

Since Expo's native code never changes with regard to your project, the native symbols aren't especially meaningful (they would show you a trace into the React Native core or into Expo's native SDK). In the vast majority of circumstances *, the JS error is what you care about.

Nonetheless, if you really want native crash logs and are deploying your app as a [standalone app](#), you can configure custom Fabric keys for Android. See [Configuration with app.json](#).

For iOS, right now we don't expose a way for you to see native crash logs from your Expo app. This is because we don't build iOS native code on demand, which would be a requirement for uploading your debug symbols to Fabric (or a similar service).

* There are a few circumstances where it's possible to crash native code by writing bad JS. Usually these are in areas where it would be performance-prohibitive to add native validation to your code, e.g. the part of the React Native bridge that converts JS objects into typed native values. If you encounter an inexplicable native crash, double check that your parameters are of the right type.

Preloading & Caching Assets

Assets are cached differently depending on where they're stored and how they're used. This guide offers best practices for making sure you only download assets when you need to. In order to keep the loading screen visible while caching assets, it's also a good idea to render `Expo.AppLoading` and only that component until everything is ready. See also: [Offline Support](#).

For images that saved to the local filesystem, use `Expo.Asset.fromModule(image).downloadAsync()` to download and cache the image. There is also a `loadAsync()` helper method to cache a batch of assets.

For web images, use `Image.prefetch(image)`. Continue referencing the image normally, e.g. with `<Image source={require('path/to/image.png')} />`.

Fonts are preloaded using `Expo.Font.loadAsync(font)`. The `font` argument in this case is an object such as the following: `{OpenSans: require('./assets/fonts/OpenSans.ttf')}`. `@expo/vector-icons` provides a helpful shortcut for this object, which you see below as `FontAwesome.font`.

```
import React from 'react';
import { AppLoading, Asset, Font } from 'expo';
import { View, Text, Image } from 'react-native';
import { FontAwesome } from '@expo/vector-icons';

function cacheImages(images) {
  return images.map(image => {
    if (typeof image === 'string') {
      return Image.prefetch(image);
    } else {
      return Asset.fromModule(image).downloadAsync();
    }
  });
}

function cacheFonts(fonts) {
  return fonts.map(font => Font.loadAsync(font));
}

export default class AppContainer extends React.Component {
  state = {
    isReady: false,
  };

  async _loadAssetsAsync() {
    const imageAssets = cacheImages([
      'https://www.google.com/images/branding/googlelogo/2x/googlelogo_color_272x92dp.png',
      require('./assets/images/circle.jpg'),
    ]);

    const fontAssets = cacheFonts([FontAwesome.font]);

    await Promise.all([...imageAssets, ...fontAssets]);
  }

  render() {
    if (!this.state.isReady) {
      return (
        <AppLoading
          startAsync={this._loadAssetsAsync}
          onFinish={() => this.setState({ isReady: true })}
          onError={console.warn}
      );
    }
  }
}
```

```
        />
    );
}

return (
    <View>
        <Text>Hello world, this is my app.</Text>
    </View>
);
}
```

See a full working example in [github/expo/new-project-template](https://github.com/expo/expo/tree/main/packages/react-native-template-app).

Icons

As trendy as it is these days, not every app has to use emoji for all icons -- maybe you want to pull in a popular set through an icon font like FontAwesome, Glyphicons or Ionicons, or you just use some PNGs that you carefully picked out on [The Noun Project](#) (Expo does not currently support SVGs). Let's look at how to do both of these approaches.

@expo/vector-icons

This library is installed by default on the template project that get through `expo init` -- it is part of the `expo` package. It includes popular icon sets and you can browse all of the icons using the [@expo/vector-icons directory](#).

```
import React from 'react';
import { Ionicons } from '@expo/vector-icons';

export default class IconExample extends React.Component {
  render() {
    return (
      <Ionicons name="md-checkmark-circle" size={32} color="green" />
    );
  }
}
```

This component loads the Ionicons font if it hasn't been loaded already, and renders a checkmark icon that I found through the vector-icons directory mentioned above. `@expo/vector-icons` is built on top of [react-native-vector-icons](#) and uses a similar API. The only difference is `@expo/vector-icons` uses a more idiomatic `import` style:

```
import { Ionicons } from '@expo/vector-icons'; instead of.. import Ionicons from 'react-native-vector-icons/Ionicons'; .
```

Note: As with [any custom font](#) in Expo, you may want to preload icon fonts before rendering your app. The font object is available as a static property on the font component, so in the case above it is `Ionicons.font`, which evaluates to `{ionicons: require('path/to/ionicons.ttf')}`. [Read more about preloading assets](#).

Custom Icon Fonts

First, make sure you import your custom icon font. [Read more about loading custom fonts](#). Once your font has loaded, you'll need to create an Icon Set. `@expo/vector-icons` exposes three methods to help you create an icon set.

createIconSet

Returns your own custom font based on the `glyphMap` where the key is the icon name and the value is either a UTF-8 character or its character code. `fontFamily` is the name of the font **NOT** the filename. See [react-native-vector-icons](#) for more details.

```
import { Font } from 'expo';
import { createIconSet } from '@expo/vector-icons';
```

```

const glyphMap = { 'icon-name': 1234, test: 'Δ' };
const CustomIcon = createIconSet(glyphMap, 'FontName');

export default class CustomIconExample extends React.Component {
  state = {
    fontLoaded: false
  }
  async componentDidMount() {
    await Font.loadAsync({
      'FontName': require('assets/fonts/custom-icon-font.ttf')
    });
    this.setState({fontLoaded: true});
  }
  render() {
    if (!this.state.fontLoaded) { return null; }

    return (
      <CustomIcon name="icon-name" size={32} color="red" />
    );
  }
}

```

createIconSetFromFontello

Convenience method to create a custom font based on a [Fontello](#) config file. Don't forget to import the font as described above and drop the `config.json` somewhere convenient in your project, using `Font.loadAsync`.

```

// Once your custom font has been loaded...
import { createIconSetFromFontello } from '@expo/vector-icons';
import fontelloConfig from './config.json';
const Icon = createIconSetFromFontello(fontelloConfig, 'FontName');

```

createIconSetFromIcoMoon

Convenience method to create a custom font based on an [IcoMoon](#) config file. Don't forget to import the font as described above and drop the `config.json` somewhere convenient in your project, using `Font.loadAsync`.

```

// Once your custom font has been loaded...
import { createIconSetFromIcoMoon } from '@expo/vector-icons';
import icoMoonConfig from './config.json';
const Icon = createIconSetFromIcoMoon(icoMoonConfig, 'FontName');

```

Icon images

If you know how to use the react-native `<Image>` component this will be a breeze.

```

import React from 'react';
import { Image } from 'react-native';

export default class SlackIcon extends React.Component {
  render() {
    return (
      <Image
        source={require('../assets/images/slack-icon.png')}
        fadeDuration={0}
        style={{width: 20, height: 20}}
    
```

```
    />
  );
}
```

Let's assume that our `slackIcon` class is located in `my-project/components/SlackIcon.js`, and our icon images are in `my-project/assets/images`, in order to refer to the image we use require and include the relative path. You can provide versions of your icon at various pixel densities and the appropriate image will be automatically used for you. In this example, we actually have `slack-icon@2x.png` and `slack-icon@3x.png`, so if I view this on an iPhone 6s the image I will see is `slack-icon@3x.png`. More on this in the [Images guide in the react-native documentation](#).

We also set the `fadeDuration` (an Android specific property) to `0` because we usually want the icon to appear immediately rather than fade in over several hundred milliseconds.

Using Custom Fonts

Both iOS and Android come with their own set of platform fonts but if you want to inject some more brand personality into your app, a well picked font can go a long way. In this guide we'll walk you through adding a custom font to your Expo app. We'll use [Open Sans](#) from [Google Fonts](#) in the example, and the process is identical for any other font, so feel free to adapt it to your use case. Before proceeding, go ahead and download [Open Sans](#)

Starting code

First let's start with a basic "Hello world!" app. Create a new project with `expo init` and change `App.js` to the following:

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

export default class App extends React.Component {
  render() {
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
        <Text style={{ fontSize: 56 }}>
          Hello, world!
        </Text>
      </View>
    );
  }
}
```

Try getting this basic app running before playing with Open Sans, so you can get any basic setup issues out of the way.

Downloading the font

Take the Open Sans zipfile that you downloaded, extract it and copy `OpenSans-Bold.ttf` into the assets directory in your project. The location we recommend is `your-project/assets/fonts`.

Loading the font in your app

To load and use fonts we will use the [Expo SDK](#), which comes pre-installed when you create a new Expo project, but if for some reason you don't have it, you can install with `npm install --save expo` in your project directory. Add the following `import` in your application code:

```
import { Font } from 'expo';
```

The `expo` library provides an API to access native functionality of the device from your JavaScript code. `Font` is the module that deals with font-related tasks. First, we must load the font from our assets directory using `Expo.Font.loadAsync()`. We can do this in the `componentDidMount()` lifecycle method of the `App` component. Add the following method in `App`: Now that we have the font files saved to disk and the Font SDK imported, let's add this code:

```
export default class App extends React.Component {
  componentDidMount() {
    Font.loadAsync({
      'open-sans-bold': require('./assets/fonts/OpenSans-Bold.ttf'),
    });
  }

  // ...
}
```

This loads Open Sans Bold and associates it with the name `'open-sans-bold'` in Expo's font map. Now we just have to refer to this font in our `Text` component.

Note: Fonts loaded through Expo don't currently support the `fontWeight` OR `fontStyle` properties -- you will need to load those variations of the font and specify them by name, as we have done here with bold.

Using the font in a `Text` component

With React Native you specify fonts in `Text` components using the `fontFamily` style property. The `fontFamily` is the key that we used with `Font.loadAsync`.

```
<Text style={{ fontFamily: 'open-sans-bold', fontSize: 56 }}>
  Hello, world!
</Text>
```

On next refresh the app seems to still not display the text with Open Sans Bold. You will see that it is still using the default system font. The problem is that `Expo.Font.loadAsync()` is an asynchronous call and takes some time to complete. Before it completes, the `Text` component is already rendered with the default font since it can't find the `'open-sans-bold'` font (which hasn't been loaded yet).

Waiting for the font to load before rendering

We need a way to re-render the `Text` component when the font has finished loading. We can do this by keeping a boolean value `fontLoaded` in the `App` component's state that keeps track of whether the font has been loaded. We render the `Text` component only if `fontLoaded` is `true`.

First we initialize `fontLoaded` to `false` in the `App` class constructor:

```
class App extends React.Component {
  state = {
    fontLoaded: false,
  };

  // ...
}
```

Next, we must set `fontLoaded` to `true` when the font is done loading. `Expo.Font.loadAsync()` returns a `Promise` that is fulfilled when the font is successfully loaded and ready to use. So we can use `async/await` with `componentDidMount()` to wait until the font is loaded, then update our state.

```
class App extends React.Component {
  async componentDidMount() {
    await Font.loadAsync({
      'open-sans-bold': require('./assets/fonts/OpenSans-Bold.ttf'),
    });

    this.setState({ fontLoaded: true });
  }

  // ...
}
```

Finally, we want to only render the `Text` component if `fontLoaded` is `true`. We can do this by replacing the `Text` element with the following:

```
<View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
{
  this.state.fontLoaded ? (
    <Text style={{ fontFamily: 'open-sans-bold', fontSize: 56 }}>
      Hello, world!
    </Text>
  ) : null
}
</View>
```

A `null` child element is simply ignored by React Native, so this skips rendering the `Text` component when `fontLoaded` is `false`. Now on refreshing the app you will see that `open-sans-bold` is used.

This technique is built into the Tabs template for convenience, as you can see [here](#).

Note: Typically you will want to load your apps primary fonts before the app is displayed to avoid text flashing in after the font loads. The recommended approach is to move the `Font.loadAsync` call to your top-level component.

Routing & Navigation

A "single page app" on the web is not an app with a single screen, that would indeed be useless most of the time; rather, it is an app that does not ask the browser to navigate to a new URL for each new screen. Instead, a "single page app" will use its own routing subsystem (eg: react-router) that decouples the screens that are being displayed from the URL bar. Often it will also update the URL bar too, but override the mechanism that will cause the browser to reload the page entirely. The purpose of this is for the experience to be smooth and "app-like".

This same concept applies to native mobile apps. When you navigate to a new screen, rather than refreshing the entire app and starting fresh from that screen, the screen is pushed onto a navigation stack and animated into view according to its configuration.

The library that we recommend to use for routing & navigation in Expo is [React Navigation](#). We recommend following the [fundamentals guide](#) in the [React Navigation documentation](#) to learn more about how to use it.

Configuring StatusBar

Expo and React Native provide APIs and configuration options for Android to configure the status bar for your app. These can be used to control the appearance of the status bar in your app.

Configuration (Android)

The configuration for Android status bar lives under the `androidStatusBar` key in `app.json`. It exposes the following options:

barStyle

This option can be used to specify whether the status bar content (icons and text in the status bar) is light, or dark. Usually a status bar with a light background has dark content, and a status bar with a dark background has light content.

The valid values are:

- `light-content` - The status bar content is light colored (usually white). This is the default value.
- `dark-content` - The status bar content is dark colored (usually dark grey). This is only available on Android 6.0 onwards. It will fallback to `light-content` in older versions.

backgroundColor

This option can be used to set a background color for the status bar.

Keep in mind that the Android status bar is translucent by default in Expo apps. But, when you specify an opaque background color for the status bar, it'll lose its translucency.

The valid value is a hexadecimal color string. e.g. - #C2185B

Working with 3rd-party Libraries

Expo makes the status bar translucent by default on Android which is consistent with iOS, and more in line with material design. Unfortunately some libraries don't support translucent status bar, e.g. - navigation libraries, libraries which provide a header bar etc.

If you need to use such a library, there are a few options:

Set the `backgroundColor` of the status bar to an opaque color

This will disable the translucency of the status bar. This is a good option if your status bar color never needs to change.

Example:

```
{  
  "expo": {  
    "androidStatusBar": {  
      "backgroundColor": "#C2185B"
```

```
    }
}
}
```

Use the `StatusBar` API from React Native

The `StatusBar` API allows you to dynamically control the appearance of the status bar. You can use it as component, or as an API. Check the documentation on the React Native website for examples.

Place an empty `View` on top of your screen

You can place an empty `View` on top of your screen with a background color to act as a status bar, or set a top padding. You can get the height of the status bar with `Constants.statusBarHeight`. Though this should be your last resort since this doesn't work very well when status bar's height changes.

Example:

```
import React from 'react';
import { StyleSheet, View } from 'react-native';
import { Constants } from 'expo';

const styles = StyleSheet.create({
  statusBar: {
    backgroundColor: "#C2185B",
    height: Constants.statusBarHeight,
  },
  // rest of the styles
});

const MyComponent = () => {
  <View>
    <View style={styles.statusBar} />
    {/* rest of the content */}
  </View>
}
```

If you don't need to set the background color, you can just set a top padding on the wrapping `View` instead.

Offline Support

Your app will encounter circumstances where the internet connection is sub-par or totally unavailable and it still needs to work reasonably well. This guide offers more information and best practices for providing a great experience while the device is offline.

Load JS updates in the background

When you [publish](#) an update to your app, your users will receive the new version of your JS over the air. The new version will download either next time the app starts, or next time you call `Updates.reload()`. This behavior also applies the very first time the user opens your app.

Expo offers multiple behaviors for how it should download your JS. It can either block the UI with a [splash screen](#) or [AppLoading component](#) until the new JS is downloaded, or it can immediately show an old version of your JS and download the update in the background. The former option is better if your users must have the latest version at all times; the latter option is better if you have a bad internet connection and need to show something right away.

To force JS updates to run in the background (rather than synchronously checking and downloading on app start), set `updates.fallbackToCacheTimeout` to `0` in `app.json`. You can also listen to see when a new version has finished downloading. For more information, see [Configuring OTA Updates](#).

Cache your assets after downloading

By default, all of your assets (images, fonts, etc.) are [uploaded to Expo's CDN](#) when you publish updates to your app, which allows you to update them over the air. Once they're downloaded, you can [cache them](#) so you don't need to download them a second time. If you publish changes, the cache will be invalidated and the changed version will be downloaded.

Bundle your assets inside your standalone binary

Expo can bundle assets into your standalone binary during the build process so that they will be available immediately, even if the user has never run your app before. This is important if:

- Your users may not have internet the first time they open your app, or
- If your app relies on a nontrivial amount of assets for the very first screen to function properly.

To bundle assets in your binary, use the `assetBundlePatterns` key in `app.json` to provide a list of paths in your project directory:

```
"assetBundlePatterns": [  
  "assets/images/*"  
,
```

Images with paths matching the given patterns will be bundled into your native binaries next time you run `expo build`.

Listen for changes in network availability

React Native exposes the [NetInfo API](#), which informs you if your device's reachability changes. You may want to change your UI (e.g. show a banner, or disable some functions) if you notice that there's no connection available.

Configuring OTA Updates

Expo provides various settings to configure how your app receives over-the-air (OTA) JavaScript updates. OTA updates allow you to publish a new version of your app JavaScript and assets without building a new version of your standalone app and re-submitting to app stores ([read more about the limitations](#)).

To perform an over-the-air update of your app, you simply run `expo publish`. If you're using release channels, specify one with `--release-channel <channel-name>` option. Please note that if you wish to update the SDK version which your app is using, you need to rebuild your app with `expo build:*` command and upload the binary file to the appropriate app store ([see the docs here](#)).

OTA updates are controlled by the `updates` [settings in app.json](#), which handle the initial app load, and the [Updates SDK module](#), which allows you to fetch updates asynchronously from your JS.

Automatic Updates

By default, Expo will check for updates automatically when your app is launched and will try to fetch the latest published version. If a new bundle is available, Expo will attempt to download it before launching the experience. If there is no network connection available, or it has not finished downloading in 30 seconds, Expo will fall back to loading a cached version of your app, and continue trying to fetch the update in the background (at which point it will be saved into the cache for the next app load).

With this automatic configuration, calling `Expo.Updates.reload()` will also result in Expo attempting to fetch the most up-to-date version of your app, so there is no need to use any of the other methods in the `Updates` module.

The timeout length is configurable by setting `updates.fallbackToCacheTimeout` (ms) in `app.json`. For example, a common pattern is to set `updates.fallbackToCacheTimeout` to `0`. This will allow your app to start immediately with a cached bundle while downloading a newer one in the background for future use. `Expo.Updates.addListener` provides a hook to let you respond when the new bundle is finished downloading.

Manual Updates

In standalone apps, it is also possible to turn off automatic updates, and to instead control updates entirely within your JS code. This is desirable if you want some custom logic around fetching updates (e.g. only over Wi-Fi).

Setting `updates.checkAutomatically` to `"ON_ERROR_RECOVERY"` in `app.json` will prevent Expo from automatically fetching the latest update every time your app is launched. Only the most recent cached version of your bundle will be loaded. It will only automatically fetch an update if the last run of the cached bundle produced a fatal JS error.

You can then use the `Expo.Updates` module to download new updates and, if appropriate, notify the user and reload the experience.

```
try {
  const update = await Expo.Updates.checkForUpdateAsync();
  if (update.isAvailable) {
    await Expo.Updates.fetchUpdateAsync();
    // ... notify user of update ...
    Expo.Updates.reloadFromCache();
  }
} catch (e) {
```

```
// handle or log error  
}
```

Note that `checkAutomatically: "ON_ERROR_RECOVERY"` will be ignored in the Expo client, although the imperative `Updates` methods will still function normally.

Disabling Updates

It is possible to entirely disable OTA JavaScript updates in a standalone app, by setting `updates.enabled` to `false` in `app.json`. This will ignore all code paths that fetch app bundles from Expo's servers. In this case, all updates to your app will need to be routed through the iOS App Store and/or Google Play Store.

This setting is ignored in the Expo client.

Push Notifications

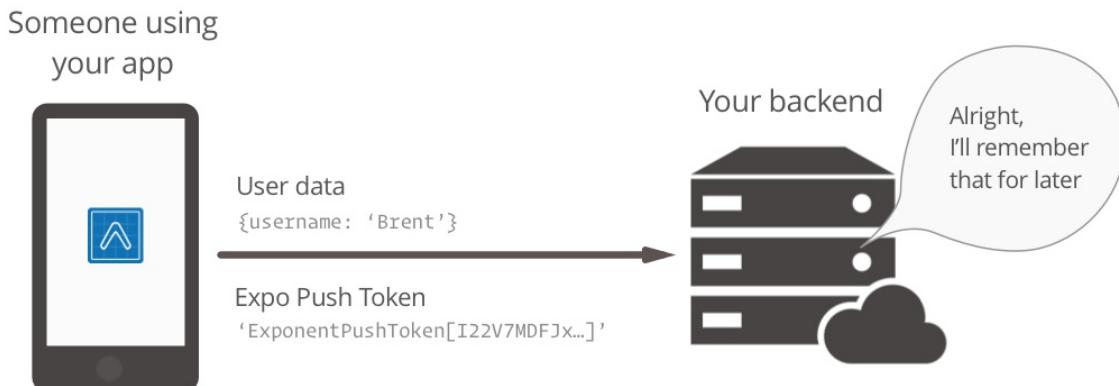
Push Notifications are an important feature to, as "*growth hackers*" would say, retain and re-engage users and monetize on their attention, or something. From my point of view it's just super handy to know when a relevant event happens in an app so I can jump back into it and read more. Let's look at how to do this with Expo. Spoiler alert: it's almost too easy.

Note: iOS and Android simulators cannot receive push notifications. To test them out you will need to use a real-life device. Additionally, when calling Permissions.askAsync on the simulator, it will resolve immediately with "undetermined" as the status, regardless of whether you choose to allow or not.

There are three main steps to wiring up push notifications: sending a user's Expo Push Token to your server, calling Expo's Push API with the token when you want to send a notification, and responding to receiving and/or selecting the notification in your app (for example to jump to a particular screen that the notification refers to).

1. Save the user's Expo Push Token on your server

In order to send a push notification to somebody, we need to know about their device. Sure, we know our user's account information, but Apple, Google, and Expo do not understand what devices correspond to "Brent" in your proprietary user account system. Expo takes care of identifying your device with Apple and Google through the Expo push token, which is unique each time an app is installed on a device. All we need to do is send this token to your server so you can associate it with the user account and use it in the future for sending push notifications.



```
import { Permissions, Notifications } from 'expo';

const PUSH_ENDPOINT = 'https://your-server.com/users/push-token';

async function registerForPushNotificationsAsync() {
  const { status: existingStatus } = await Permissions.getAsync(
    Permissions.NOTIFICATIONS
  );
  let finalStatus = existingStatus;

  // only ask if permissions have not already been determined, because
  // iOS won't necessarily prompt the user a second time.
  if (existingStatus !== 'granted') {
    // Android remote notification permissions are granted during the app
```

```

// install, so this will only ask on iOS
const { status } = await Permissions.askAsync(Permissions.NOTIFICATIONS);
finalStatus = status;
}

// Stop here if the user did not grant permissions
if (finalStatus !== 'granted') {
  return;
}

// Get the token that uniquely identifies this device
let token = await Notifications.getExpoPushTokenAsync();

// POST the token to your backend server from where you can retrieve it to send push notifications.
return fetch(PUSH_ENDPOINT, {
  method: 'POST',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    token: {
      value: token,
    },
    user: {
      username: 'Brent',
    },
  }),
});
}

```

2. Call Expo's Push API with the user's token

Push notifications have to come from somewhere, and that somewhere is your server, probably (you could write a command line tool to send them if you wanted, it's all the same). When you're ready to send a push notification, grab the Expo push token off of the user record and send it over to the Expo API using a plain old HTTPS POST request. We've taken care of wrapping that for you in a few languages:

Someone liked Brent's puppy photo! He will be stoked. Let's let him know.

Your backend

Brent's Expo Push Token

'ExponentPushToken[I22V7M...]'

message 'Puppy!'

data {photoId: 152}

Expo backend



notification

Google Cloud Messaging



notification



if iOS

- [expo-server-sdk-node](#) for Node.js. Maintained by the Expo team.
- [expo-server-sdk-python](#) for Python. Maintained by community developers.
- [expo-server-sdk-ruby](#) for Ruby. Maintained by community developers.
- [expo-server-sdk-rust](#) for Rust. Maintained by community developers.
- [ExpoNotificationsBundle](#) for Symfony. Maintained by SolveCrew.
- [exponent-server-sdk-php](#) for PHP. Maintained by community developers.
- [exponent-server-sdk-golang](#) for Golang. Maintained by community developers.
- [exponent-server-sdk-elixir](#) for Elixir. Maintained by community developers.

Check out the source if you would like to implement it in another language.

Note: For Android, you'll also need to upload your Firebase Cloud Messaging server key to Expo so that Expo can send notifications to your app. **This step is necessary** unless you are not creating your own APK and using just the Expo Client app from Google Play. Follow the guide on [Using FCM for Push Notifications](#) to learn how to create a Firebase project, get your FCM server key, and upload the key to Expo.

The [Expo push notification tool](#) is also useful for testing push notifications during development. It lets you easily send test notifications to your device.

3. Handle receiving and/or selecting the notification

For Android, this step is entirely optional -- if your notifications are purely informational and you have no desire to handle them when they are received or selected, you're already done. Notifications will appear in the system notification tray as you've come to expect, and tapping them will open/foreground the app.

For iOS, you would be wise to handle push notifications that are received while the app is foregrounded, because otherwise the user will never see them. Notifications that arrive while the app are foregrounded on iOS do not show up in the system notification list. A common solution is to just show the notification manually. For example, if you get a message on Messenger for iOS, have the app foregrounded, but do not have that conversation open, you will see the notification slide down from the top of the screen with a custom notification UI.

Thankfully, handling push notifications is straightforward with Expo, all you need to do is add a listener using the `Notifications` API.

```
import React from 'react';
import {
  Notifications,
} from 'expo';
import {
  Text,
  View,
} from 'react-native';

// This refers to the function defined earlier in this guide
import registerForPushNotificationsAsync from './registerForPushNotificationsAsync';

export default class AppContainer extends React.Component {
  state = {
    notification: {},
  };

  componentDidMount() {
    registerForPushNotificationsAsync();

    // Handle notifications that are received or selected while the app
    // is open. If the app was closed and then opened by tapping the
    // notification (rather than just tapping the app icon to open it),
    // this function will fire on the next tick after the app starts
    // with the notification data.
    this._notificationSubscription = Notifications.addListener(this._handleNotification);
  }

  _handleNotification = (notification) => {
    this.setState({notification: notification});
  };

  render() {
    return (
      <View style={{flex: 1, justifyContent: 'center', alignItems: 'center'}}>
        <Text>Origin: {this.state.notification.origin}</Text>
        <Text>Data: {JSON.stringify(this.state.notification.data)}</Text>
      </View>
    );
  }
}
```

Determining origin of the notification

Event listeners added using `Notifications.addListener` will receive an object when a notification is received ([docs](#)). The `origin` of the object will vary based on the app's state at the time the notification was received and the user's subsequent action. The table below summarizes the different possibilities and what the `origin` will be in each case.

Push was received when...	origin will be...

App is open and foregrounded	'received'
App is open and backgrounded, then notification not selected	n/a, no notification is passed to listener
App is open and backgrounded, then notification is selected	'selected'
App was not open, and then opened by selecting the push notification	'selected'
App was not open, and then opened by tapping the home screen icon	n/a, no notification is passed to listener

HTTP/2 API

Although there are server-side SDKs in several languages to help you send push notifications, you may want to directly send requests to our HTTP/2 API.

Sending notifications

Send a POST request to `https://exp.host/--/api/v2/push/send` with the following HTTP headers:

```
host: exp.host
accept: application/json
accept-encoding: gzip, deflate
content-type: application/json
```

The Expo server also optionally accepts gzip-compressed request bodies. This can greatly reduce the amount of upload bandwidth needed to send large numbers of notifications. The [Node SDK](#) automatically gzips requests for you.

This API currently does not require any authentication.

This is a "hello world" request using cURL (replace the placeholder push token with your own):

```
curl -H "Content-Type: application/json" -X POST "https://exp.host/--/api/v2/push/send" -d '{
  "to": "ExponentPushToken[xxxxxxxxxxxxxxxxxxxxxx]",
  "title": "hello",
  "body": "world"
}'
```

The HTTP request body must be JSON. It may either be a single message object or an array of up to 100 messages. **We recommend using an array when you want to send multiple messages to efficiently minimize the number of requests you need to make to Expo servers.** This is an example request body that sends two messages:

```
[{
  "to": "ExponentPushToken[xxxxxxxxxxxxxxxxxxxxxx]",
  "sound": "default",
  "body": "Hello world!"
}, {
  "to": "ExponentPushToken[yyyyyyyyyyyyyyyyyyyy]",
  "badge": 1,
  "body": "You've got mail"
}]
```

Upon success, the HTTP response will be a JSON object whose `data` field is an array of **push tickets**, each of which corresponds to the message at its respective index in the request. Continuing the above example, this is what a successful response body looks like:

```
{  
  "data": [  
    {"status": "ok", "id": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"},  
    {"status": "ok", "id": "yyyyyyyy-yyyy-yyyy-yyyyyyyyyyyy"}  
  ]  
}
```

If you send a single message that isn't wrapped in an array, the `data` field will be the push ticket also not wrapped in an array.

Push tickets

Each push ticket indicates whether Expo successfully received the notification and, when successful, a receipt ID to later retrieve a push receipt. When there is an error receiving a message, the ticket's status will be "error" and the ticket will contain information about the error and might not contain a receipt ID. More information about the response format is documented below.

Note: Even if a ticket says "ok", it doesn't guarantee that the notification will be delivered nor that the device has received the message; "ok" in a push ticket means that Expo successfully received the message and enqueued it to be delivered to the Android or iOS push notification service.

Push receipts

After receiving a batch of notifications, Expo enqueues each notification to be delivered to the iOS and Android push notification services (APNs and FCM, respectively). Most notifications are typically delivered within a few seconds. Sometimes it may take longer to deliver notifications, particularly if the iOS or Android push notification services are taking longer than usual to receive and deliver notifications, or if Expo's cloud infrastructure is under high load. Once Expo delivers a notification to the iOS or Android push notification service, Expo creates a **push receipt** that indicates whether the iOS or Android push notification service successfully received the notification. If there was an error delivering the notification, perhaps due to faulty credentials or service downtime, the push receipt will contain information about the error.

To fetch the push receipts, send a POST request to `https://exp.host/--/api/v2/push/getReceipts`. The request body must be a JSON object with a field name "ids" that is an array of receipt ID strings:

```
curl -H "Content-Type: application/json" -X POST "https://exp.host/--/api/v2/push/getReceipts" -d '{  
  "ids": ["xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx", "yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyyyy"]  
}'
```

The response body contains a mapping from receipt IDs to receipts:

```
{  
  "data": {  
    "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx": { "status": "ok" },  
    "yyyyyyyy-yyyy-yyyy-yyyy-yyyyyyyyyyyy": { "status": "ok" }  
  }  
}
```

You must check each push receipt, which may contain information about errors you need to resolve. For example, if a device is no longer eligible to receive notifications, Apple's documentation asks that you stop sending notifications to that device. The push receipts will contain information about these errors.

Note: Even if a receipt says "ok", it doesn't guarantee that the device has received the message; "ok" in a push receipt means that the Android or iOS push notification service successfully received the notification. If the recipient device is turned off, for example, the iOS or Android push notification service will try to deliver the message but the device won't necessarily receive it.

Message format

Each message must be a JSON object with the given fields:

```
type PushMessage = {
  /**
   * An Expo push token specifying the recipient of this message.
   */
  to: string,

  /**
   * A JSON object delivered to your app. It may be up to about 4KiB; the total
   * notification payload sent to Apple and Google must be at most 4KiB or else
   * you will get a "Message Too Big" error.
   */
  data?: Object,

  /**
   * The title to display in the notification. Devices often display this in
   * bold above the notification body. Only the title might be displayed on
   * devices with smaller screens like Apple Watch.
   */
  title?: string,

  /**
   * The message to display in the notification
   */
  body?: string,

  /**
   * Time to Live: the number of seconds for which the message may be kept
   * around for redelivery if it hasn't been delivered yet. Defaults to 0.
   *
   * On Android, we make a best effort to deliver messages with zero TTL
   * immediately and do not throttle them
   *
   * This field takes precedence over `expiration` when both are specified.
   */
  ttl?: number,

  /**
   * A timestamp since the UNIX epoch specifying when the message expires. This
   * has the same effect as the `ttl` field and is just an absolute timestamp
   * instead of a relative time.
   */
  expiration?: number,

  /**
   * The delivery priority of the message. Specify "default" or omit this field
   * to use the default priority on each platform, which is "normal" on Android
   * and "high" on iOS.
   *
   * On Android, normal-priority messages won't open network connections on
}
```

```

    * sleeping devices and their delivery may be delayed to conserve the battery.
    * High-priority messages are delivered immediately if possible and may wake
    * sleeping devices to open network connections, consuming energy.
    *
    * On iOS, normal-priority messages are sent at a time that takes into account
    * power considerations for the device, and may be grouped and delivered in
    * bursts. They are throttled and may not be delivered by Apple. High-priority
    * messages are sent immediately. Normal priority corresponds to APNs priority
    * level 5 and high priority to 10.
    */


// iOS-specific fields

/**
 * A sound to play when the recipient receives this notification. Specify
 * "default" to play the device's default notification sound, or omit this
 * field to play no sound.
 *
 * Note that on apps that target Android 8.0+ (if using `expo build`, built
 * in June 2018 or later), this setting will have no effect on Android.
 * Instead, use `channelId` and a channel with the desired setting.
 */
sound?: 'default' | null,

/**
 * Number to display in the badge on the app icon. Specify zero to clear the
 * badge.
 */
badge?: number,

/**
 * ID of the Notification Category through which to display this notification.
 *
 * To send a notification with category to the Expo Client, prefix the string
 * with the experience ID (`@user/experienceId:yourCategoryId`). For standalone/ejected
 * applications, use plain `yourCategoryId` .
 */
_category?: string

// Android-specific fields

/**
 * ID of the Notification Channel through which to display this notification
 * on Android devices. If an ID is specified but the corresponding channel
 * does not exist on the device (i.e. has not yet been created by your app),
 * the notification will not be displayed to the user.
 *
 * If left null, a "Default" channel will be used, and Expo will create the
 * channel on the device if it does not yet exist. However, use caution, as
 * the "Default" channel is user-facing and you may not be able to fully
 * delete it.
 */
channelId?: string
}

```

Response format

The response is a JSON object with two optional fields, `data` and `errors`. If there is an error with the entire request, the HTTP status code will be 4xx or 5xx and `errors` will be an array of error objects (usually just one):

```
{
  "errors": [{
```

```

        "code": "INTERNAL_SERVER_ERROR",
        "message": "An unknown error occurred."
    }]
}

```

If there are errors that affect individual messages but not the entire request, the HTTP status code will be 200, the `errors` field will be empty, and the `data` field will contain push tickets that describe the errors:

```

{
  "data": [
    {
      "status": "error",
      "message": "\\"ExponentPushToken[xxxxxxxxxxxxxxxxxxxx]\\\" is not a registered push notification recipient",
      "details": {
        "error": "DeviceNotRegistered"
      }
    },
    {
      "status": "ok",
      "id": "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXX"
    }
  ]
}

```

Note: You should check the ticket for each notification to determine if there was a problem delivering it to Expo. In particular, **do not assume a 200 HTTP status code means your notifications were sent successfully**; the granularity of push notification errors is finer than that of HTTP statuses.

The HTTP status code will be 200 also if all of the messages were successfully delivered to Expo and enqueued to be delivered to the iOS and Android push notification services.

Successful push receipts, and some types of failed ones, will contain an "id" field with the ID of a receipt to fetch later.

Receipt request format

Each receipt request must contain a field named "ids" that is an array of receipt IDs:

```
{
  "ids": string[]
}
```

Receipt response format

The response format for push receipts is similar to that of push tickets; it is a JSON object with two optional fields, `data` and `errors`. If there is an error with the entire request, the HTTP status code will be 4xx or 5xx and `errors` will be an array of error objects.

If there are errors that affected individual notifications but not the entire request, the HTTP status code will be 200, the `errors` field will be empty, and the `data` field will be a JSON object whose keys are receipt IDs and values are corresponding push receipts. If there is no push receipt for a requested receipt ID, the mapping won't contain that ID. This is an example response:

```
{
  "data": {
    "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXX": {
      "status": "error",
      "message": "The Apple Push Notification service failed to send the notification",
      "details": {

```

```
        "error": "DeviceNotRegistered"
    },
},
"YYYYYYYY-YYYY-YYYY-YYYY-YYYYYYYYYYYY": {
    "status": "ok"
}
}
```

Note: You should check each receipt to determine if there was an issue delivering the notification to the Android or iOS push notification service. In particular, **do not assume a 200 HTTP status code means your notifications were sent successfully**; the granularity of push notification errors is finer than that of HTTP statuses.

The HTTP status code will be 200 also if all of the messages were successfully delivered to the Android and iOS push notification services.

Important: in particular, look for an `details` object with an `error` field inside both push tickets and push receipts. If present, it may be one of these values: `DeviceNotRegistered`, `MessageTooBig`, `MessageRateExceeded`, and `InvalidCredentials`. You should handle these errors like so:

- `DeviceNotRegistered` : the device cannot receive push notifications anymore and you should stop sending messages to the corresponding Expo push token.
- `MessageTooBig` : the total notification payload was too large. On Android and iOS the total payload must be at most 4096 bytes.
- `MessageRateExceeded` : you are sending messages too frequently to the given device. Implement exponential backoff and slowly retry sending messages.
- `InvalidCredentials` : your push notification credentials for your standalone app are invalid (ex: you may have revoked them). Run `expo build:ios -c` to regenerate new push notification credentials for iOS.

If Expo couldn't deliver the message to the Android or iOS push notification service, the receipt's details may also include service-specific information. This is useful mostly for debugging and reporting possible bugs to Expo.

Expired Credentials

When your push notification credentials have expired, simply run `expo build:ios -c --no-publish` to clear your expired credentials and generate new ones. The new credentials will take effect within a few minutes of being generated. You do not have to submit a new build!

Using FCM for Push Notifications

Firebase Cloud Messaging is a popular option for delivering push notifications reliably and is required for all new standalone Android apps made with Expo. To set up your Expo Android app to get push notifications using your own FCM credentials, follow this guide closely.

Note that FCM is not currently available for Expo iOS apps.

Client Setup

1. If you have not already created a Firebase project for your app, do so now by clicking on **Add project** in the [Firebase Console](#).
2. In your new project console, click **Add Firebase to your Android app** and follow the setup steps. **Make sure that the Android package name you enter is the same as the value of `android.package` in your `app.json`.**
3. Download the `google-services.json` file and place it in your Expo app's root directory.
4. In your `app.json`, add an `android.googleServicesFile` field with the relative path to the `google-services.json` file you just downloaded. If you placed it in the root directory, this will probably look like

```
{  
  ...  
  "android": {  
    "googleServicesFile": "./google-services.json",  
    ...  
  }  
}
```

Finally, make a new build of your app by running `expo build:android`.

Uploading Server Credentials

In order for Expo to send notifications from our servers using your credentials, you'll need to upload your secret server key. You can find this key in the Firebase Console for your project:

1. At the top of the sidebar, click the **gear icon** to the right of **Project Overview** to go to your project settings.
2. Click on the **Cloud Messaging** tab in the Settings pane.
3. Copy the token listed next to **Server key**.
4. Run `expo push:android:upload --api-key <your-token-here>`, replacing `<your-token-here>` with the string you just copied. We'll store your token securely on our servers, where it will only be accessed when you send a push notification.

That's it -- users who run this new version of the app will now receive notifications through FCM using your project's credentials. You just send the push notifications as you normally would (see [guide](#)). We'll take care of choosing the correct service to send the notification.

Notification Channels

Notification channels are a new feature in Android Oreo that give users more control over the notifications they receive. Starting in Android Oreo, every local and push notification must be assigned to a single channel. Users can see all notification channels in their OS Settings, and they can customize the behavior of alerts on a per-channel basis.

Notification channels have no effect and are ignored on all iOS devices.

Designing channels

Channels give users more control over the various kinds of alerts they want to receive from your app. You should create a channel for each type of notification you might send, such as Alarms, Chat messages, Update notifications, or the like.

According to the Android developer documentation:

You should create a channel for each distinct type of notification you need to send. You can also create notification channels to reflect choices made by users of your app. For example, you can set up separate notification channels for each conversation group created by a user in a messaging app.

When you create a channel, you can specify various settings for its notifications, such as priority, sound, and vibration. After you create the channel, control switches entirely to the user, who can then customize these settings to their own liking for each channel. Your app can no longer change any of the settings. Although it is possible for your app to programmatically delete channels, Android does not recommend this and keeps a relic on the user's device.

You can read more about notification channels on the [Android developer website](#). It's a good idea to put some thought into designing channels that make sense and are useful customization tools for your users.

Creating and using channels

Creating a channel is easy -- before you create a local notification (or receive a push notification), simply call the following method:

```
if (Platform.OS === 'android') {
  Expo.Notifications.createChannelAndroidAsync('chat-messages', {
    name: 'Chat messages',
    sound: true,
  });
}
```

Creating a channel that already exists is essentially a no-op, so it's safe to call this each time your app starts up. For example, the `componentDidMount` of your app's root component might be a good place for this. However, note that you cannot change any settings of a notification channel after it's been created -- only the user can do this. So be sure to plan your channels carefully.

Then, when you want to send a notification for a chat message, either add the `channelId: 'chat-messages'` field to your [push notification message](#), or create a local notification like this:

```
Expo.Notifications.presentLocalNotificationAsync({
  title: 'New Message',
  body: 'Message!!!!',
  android: {
    channelId: 'chat-messages',
  },
});
```

Expo will then present your notification to the user through the `chat-messages` channel, respecting all of the user's settings for that channel.

If you create a notification and do not specify a `channelId`, Expo will automatically create a 'Default' channel for you and present the notification through that channel. If, however, you specify a `channelId` that has not yet been created on the device, **the notification will not be shown on Android 8+ devices**. Therefore, it's important to plan ahead and make sure that you create all of the channels you may need before sending out notifications.

On devices with Android 7 and below, which don't support notification channels, Expo will remember the relevant settings you created the channel with (in this case, `sound: true`) and apply them directly to the individual notification before presenting it to the user.

Updating an existing app to use channels

If you have an existing Android app that relies on `sound`, `vibrate`, or `priority` settings for notifications, you'll need to update it to take advantage of channels, as those settings no longer have any effect on individual notifications. If you do not rely on those settings, you may want to update it anyway in order to give users more control over the different types of notifications your app presents. (Note that the client-side `priority` setting involved here only affects the notification's UI behavior and is distinct from [the priority of a push notification](#), which is not affected by notification channels.)

To do this, first make sure you are using the latest minor update of `expo` for your SDK version. Notification channels are supported in SDKs 22 and above, so for example, if you're on SDK 27 you should `npm install / yarn add expo@^27.1.0`.

Next, plan out the notification channels your app will need. These may correspond to the different permutations of the `sound`, `vibrate` and `priority` settings you used on individual notifications.

Once you've decided on a set of channels, you need to add logic to your app to create them. We recommend simply creating all channels in `componentDidMount` of your app's root component; this way all users will be sure to get all channels and not miss any notifications.

For example, if this is your code before:

```
_createNotificationAsync = () => {
  Expo.Notifications.presentLocalNotificationAsync({
    title: 'Reminder',
    body: 'This is an important reminder!!!!',
    android: {
      priority: 'max',
      vibrate: [0, 250, 250, 250],
      color: '#FF0000',
    },
  });
}
```

You might change it to something like this:

```

componentDidMount() {
  // ...
  if (Platform.OS === 'android') {
    Expo.Notifications.createChannelAndroidAsync('reminders', {
      name: 'Reminders',
      priority: 'max',
      vibrate: [0, 250, 250, 250],
    });
  }
}

// ...

_createNotificationAsync = () => {
  Expo.Notifications.presentLocalNotificationAsync({
    title: 'Reminder',
    body: 'This is an important reminder!!!!',
    android: {
      channelId: 'reminders',
      color: '#FF0000',
    },
  });
}

```

This will create a channel called "Reminders" with default settings of `max` priority and the vibrate pattern `[0, 250, 250, 250]`. Android 8 users can change these settings whenever they want, or even turn off notifications completely for the "Reminders" channel. When `presentLocalNotificationAsync` is called, the OS will read the channel's settings and present the notification accordingly.

Send channel notification with Expo api service.

```

[{
  "to": "ExponentPushToken[xxxxxx]",
  "title": "test",
  "priority": "high",
  "body": "test",
  "sound": "default", // android 7.0 , 6, 5 , 4
  "channelId": "chat-messages", // android 8.0 later
}]

```

Using ClojureScript

Note: ClojureScript is not officially supported by the Expo team, this guide was written by [@tiensonqin](#), feel free to reach out to him on the [Expo Slack](#) if you have questions!

Quickstart

If you're already convinced about ClojureScript and Expo and know what to do once you have figwheel running, you can just read this section. Otherwise, we encourage you to read the entire guide.

```
lein new expo your-project
cd your-project && yarn install
lein figwheel
# Now in a new tab, open the project in a simulator
expo start --ios
```

Why ClojureScript?

- First-class immutable data structures
- Minimizing state and side-effects
- Practicality and pragmatism are always core values of ClojureScript
- Lisp!
- Great JavaScript interoperability

Why on Expo?

It all begins with a [Simple Made Easy](#) design choice: **you don't write native code**.

- You only write ClojureScript or JavaScript.
- You don't have to install or use Xcode or Android Studio or deal with any of the platform specific configuration and project files.
- Much easier to upgrade when there is no native code involved -- React Native JavaScript APIs are relatively stable compared to the native side. Expo will take care of upgrading the native modules and React Native versions, you only need to upgrade your ClojureScript or JavaScript code.
- You can write iOS apps on Linux or Windows (provided that you have an iPhone to test it with).
- It's dead simple to continually share your apps. Once you published your app, you got a link. It is up to you to share the link.

1. Create an Expo project

```
# Default to use Reagent / Re-frame
lein new expo your-project
```

```
# Or On Next
lein new expo your-project +om

cd your-project && yarn install
```

2. Connect to a REPL

CLI REPL

```
lein figwheel
```

Emacs REPL

1. Invoke cider-jack-in.
2. Run `(start-figwheel)` in the connected REPL.

Cursive REPL

The first time you connect to the repl, you'll need to create a Leiningen nREPL Configuration unless you have one already.

1. Click `Run->Edit configurations`.
2. Click the `+` button at the top left and choose Clojure REPL.
3. Choose a `Local REPL`.
4. Enter a name in the Name field (e.g. "REPL").
5. Choose the radio button `Use nREPL with Leiningen`.
6. Click the `OK` button to save your REPL config.

Once this is done, you can connect to the REPL.

In IntelliJ make sure your REPL config is selected and click the green **play** button to start your REPL.

Run `(start-figwheel)` in the connected REPL.

3. Start Expo server

Using Expo CLI

```
# Install Expo CLI if you have not already
npm install -g expo-cli

# Connect to iOS simulator
expo start --ios

# Or connect to Android devices or simulators
expo start --android
```

For more information, see [Expo CLI](#).

4. Publish your app

```
# Generate main.js
lein prod-build

expo publish
```

This will publish your app to a persistent URL on Expo, for example: <https://expo.io/@community/startr>

FAQ

How do I add custom native modules?

See [How do I add custom native code to my Expo project?](#).

Does it support Google Closure advanced compilation?

It's still experimental, but it already works for multiple projects.

Does it support source maps?

Yes.

Can I use npm modules?

React Native uses JavascriptCore, so modules using built-in node like stream, fs, etc wont work. Otherwise, you can just require like: `(js/require "SomeModule")`.

Do I need to restart the REPL after adding new Javascript modules or assets?

No, you do need to reload Javascript. To do that, select **Reload** from the Developer Menu. You can also press `⌘ + R` in the iOS Simulator, or press `R` twice on Android emulators.

Will it support Boot?

Not currently, but we are working on it.

Using GraphQL

Overview

GraphQL is a *query language* for APIs. It enables declarative data fetching and thus ties in perfectly with React/React Native as a declarative framework for building user interfaces. GraphQL can either complement or entirely replace the usage of REST APIs.

The main difference between REST and GraphQL is that RESTful APIs have *multiple endpoints* that return *fixed data structures* whereas a GraphQL server only exposes a *single endpoint* and returns *flexible data structures*. This works because a client that needs data from the server also submits its precise data requirements in each request which allows the server to tailor the response exactly according to the client's needs.

You can learn more about the differences between GraphQL and REST [here](#). To get a high-level overview and understand more about the architectural use cases of GraphQL, take a look at [this article](#).

GraphQL has a rapidly growing community. To stay up-to-date about everything that's happening in the GraphQL ecosystem, check out these resources:

- [GraphQL Weekly](#): Weekly newsletter about GraphQL
- [GraphQL Radio](#): Podcast discussing real-world use cases of GraphQL
- [GraphQL Europe](#): Europe's biggest GraphQL conference
- [Prisma blog](#): Technical deep dives and tutorials all around GraphQL development

For an in-depth learning experience, visit the [How to GraphQL](#) fullstack tutorial website.

Communicating with a GraphQL API

In this section, we'll explain the core concepts you need to know when working with a GraphQL API.

Fetching data with GraphQL queries

When an application needs to retrieve data from a GraphQL API, it has to send a *query* to the server in which it specifies the data requirements. Most GraphQL servers accept only HTTP POST requests where the query is put into the *body* of the request. Note however that GraphQL itself is actually *transport layer agnostic*, meaning that the client-server communication could also happen using other networking protocols than HTTP.

Here's an example query that a client might send in an Instagram-like application:

```
query {
  feed {
    id
    imageUrl
    description
  }
}
```

The keyword `query` in the beginning expresses the *operation type*. Besides `query`, there are two more operation types called `mutation` and `subscription`. Note that the default operation type of a request is in fact `query`, so you might as well remove it from the above request. `feed` is the *root field* of the query and everything that follows is

called the *selection set* of the query.

When a server receives the above query, it will *resolve* it, i.e. collect the required data, and package up the response in the same format of the query. Here's what a potential response could look like:

```
{  
  "data": {  
    "feed": [  
      {  
        "id": "1",  
        "description": "Nice Sunset",  
        "imageUrl": "http://example.org/sunset.png"  
      },  
      {  
        "id": "2",  
        "description": "Cute Cats",  
        "imageUrl": "http://example.org/cats.png"  
      }  
    ]  
  }  
}
```

The root of the returned JSON object is a field called `data` as defined in the official [GraphQL specification](#). The rest of the JSON object then contains exactly the information that the client asked for in the query. If the client for example hadn't included the `imageUrl` in the query's selection set, the server wouldn't have included it in its response either.

In case the GraphQL request fails for some reason, e.g. because the query was malformed, the server will not return the `data` field but instead return an array called `errors` with information about the failure. Notice that it can happen that the server returns both, `data` and `errors`. This can occur when the server can only partially resolve a query, e.g. because the user requesting the data only had the access rights for specific parts of the query's payload.

Creating, updating and deleting data with GraphQL mutations

Most of the time when working with an API, you'll also want to make changes to the data that's currently stored in the backend. In GraphQL, this is done using so-called *mutations*. A mutation follows the exact same syntactical structure as a query. In fact, it actually also *is* a query in that it combines a write operation with a directly following read operation. Essentially, the idea of a mutation corresponds to the PUT, POST and DELETE calls that you would run against a REST API but additionally allows you to fetch data in a single request.

Let's consider an example mutation to create a new post in our sample Instagram app:

```
mutation {  
  createPost(description: "Funny Birds", imageUrl: "http://example.org/birds.png") {  
    id  
  }  
}
```

Instead of the `query` operation type, this time we're using `mutation`. Then follows the *root field*, which in this case is called `createPost`. Notice that all fields can also take arguments, here we provide the post's `description` and `imageUrl` so the server knows what it should write into the database. In the payload of the mutation we simply specify the `id` of the new post that will be generated on the server-side.

After the server created the new post in the database, it will return the following sample response to the client:

```
{  
  "data": {  
    "createPost": {  
      "id": "1"  
    }  
  }  
}
```

The GraphQL schema

In this section, we'll discuss the backbone of every GraphQL server: The GraphQL schema.

For a technical deep dive all around the GraphQL schema, be sure to check out [this article](#).

The Schema Definition Language (SDL)

GraphQL has a [type system](#) that's used to define the capabilities of an API. These capabilities are written down in the *GraphQL schema* using the syntax of the [GraphQL Schema Definition Language](#) (SDL). Here's what the `Post` type from our previous examples looks like:

```
type Post {  
  id: ID!  
  description: String!  
  imageUrl: String!  
}
```

The syntax is pretty straightforward. We're defining a type called `Post` that has three properties, in GraphQL terminology these properties are called *fields*. Each field has a *name* and a *type*. The exclamation point following a type means that this field cannot be `null`.

Root types are the entry points for the API

Each schema has so-called [root types](#) that define the *entry points* into the API. These are the root types that you can define in your schema:

- `Query` : Specifies all the queries a GraphQL server accepts
- `Mutation` : Specifies all the mutations a GraphQL server accepts
- `Subscription` : Specifies all the subscriptions a GraphQL server accepts (subscriptions are used for realtime functionality, learn more [here](#))

To enable the `feed` query and `createPost` mutation that we saw in the previous examples, you'd have to write the root types as follows:

```
type Query {  
  feed: [Post!]!  
}  
  
type Mutation {  
  createPost(description: String!, imageUrl: String!): Post  
}
```

You can read more about the core GraphQL constructs [here](#).

Getting started with GraphQL

The first thing you need when getting started with GraphQL is of course a GraphQL server. As GraphQL itself is only a [specification](#), you can either implement your own server using one of the available [reference implementations](#) or take a shortcut by using a tool like [Apollo Launchpad](#).

The best way to get started with GraphQL in production is to use `graphql-yoga`, a flexible GraphQL server based on Express.js. `graphql-yoga` has a number of compelling features, such as support for [GraphQL Playground](#) and built-in GraphQL subscriptions for realtime functionality.

A great way to add a database to your GraphQL server is by using [Prisma](#). Prisma is an open-source GraphQL query engine that turns your database into a GraphQL API. Thanks to [Prisma bindings](#), it integrates nicely with your `graphql-yoga` server.

To learn how to build a GraphQL server, check out this [tutorial](#) or watch this 4-min demo [video](#).

Since GraphQL servers are commonly implemented with HTTP, you can simply use `fetch` to get started and send queries and mutations to interact with the server. However, when working with GraphQL on the frontend, you'll usually want to use a [GraphQL client](#) library. GraphQL clients generally provide handy abstractions and allow you to directly send queries and mutations without having to worry about lower-level networking details.

There are four major GraphQL clients available at the moment:

- [Apollo Client](#): Community-driven, flexible and powerful GraphQL client that's easy to understand and has an intuitive API.
- [Relay](#): Facebook's homegrown GraphQL client that's heavily optimized for performance and comes with a notable learning curve.
- [Urql](#): Simple GraphQL client for React.
- [graphql-request](#): Simple and lightweight GraphQL client that works in all JavaScript environments and can be used for simple use cases like scripting.

Apollo, Relay and Urql implement further features like caching, realtime support with GraphQL subscriptions or optimistic UI updates.

Learn how to integrate with Auth0 social providers in the [expo-auth0-example](#) repository.

Creating your own GraphQL server

The fastest way to get started with GraphQL is by using a [GraphQL boilerplate](#) project for the technology of your choice. GraphQL boilerplates provide the ideal starter kits for your GraphQL-based projects - no matter if backend-only or fullstack.

To get started, you can use the `graphql create` command (which is similar to `create-react-native-app`).

First, you need to install the [GraphQL CLI](#):

```
npm install -g graphql-cli
```

With the CLI installed, you can run the following command:

```
graphql create myapp
```

This will prompt you a list of the available boilerplates. Each technology has a `minimal`, a `basic` and an `advanced` version.

Choose `minimal` to learn what the most minimal version of a GraphQL server looks like. `basic` boilerplates come with an integrated database (based on Prisma). Finally, the `advanced` boilerplates additionally come with built-in authentication functionality for your users as well as support for realtime subscriptions.

To skip the interactive prompt, you can also pass the `--boilerplate` (short: `-b`) flag to the `graphql create` command and specify which starter kit you'd like to use. For example:

```
graphql create myapp --boilerplate node-advanced # The `advanced` boilerplate for Node.js  
# or  
graphql create myapp --boilerplate react-fullstack-basic # The `basic` boilerplate for a fullstack React app
```

Running a practical example with React, Apollo & GraphQL

If you want to get your hands dirty and learn how to get started with a practical example, check out the [basic](#) boilerplate for a fullstack React application.

Note There are currently no boilerplate projects for React Native. However, all the code that's used to interact with the GraphQL API from within the React app can be applied in a React Native application in an identical manner!

Run `graphql create` and specify `react-fullstack-basic` as your target boilerplate:

```
graphql create myapp --boilerplate react-fullstack-basic
```

The GraphQL CLI will now fetch the code from the corresponding [GitHub repository](#) and run an install script to configure everything that's required.

After having downloaded the code from the repo, the CLI will prompt you to choose where you want to deploy your Prisma database service. To get started quickly, select one of the *development clusters* (`prisma-eu1` or `prisma-us1`). If you have [Docker](#) installed, you can also deploy locally.

The install script will use the generated endpoint for the Prisma service and connect the GraphQL server with it by inserting it into `src/server/index.js`.

Once the command has finished, you first need to start the server and second start the React app:

```
cd myapp/server  
yarn start  
# the server is now running on http://localhost:4000;  
# to continue, open a new tab in your terminal  
# and navigate back to the root directory  
cd ..  
yarn start
```

The React app is now running on `http://localhost:3000` (the GraphQL server is running on `http://localhost:4000`).

To learn about the queries and mutations accepted by the GraphQL API, check out the GraphQL schema that's stored in `server/src/schema.graphql`.

As an example, here is how the `FeedQuery` component is implemented that displays a list of `Post` elements:

```
import React from 'react'
import Post from '../components/Post'
import { graphql } from 'react-apollo'
import gql from 'graphql-tag'

class FeedPage extends React.Component {
  componentWillReceiveProps(nextProps) {
    if (this.props.location.key !== nextProps.location.key) {
      this.props.feedQuery.refetch()
    }
  }

  render() {
    if (this.props.feedQuery.loading) {
      return (
        <div className="flex w-100 h-100 items-center justify-center pt7">
          <div>Loading (from {process.env.REACT_APP_GRAPHQL_ENDPOINT})</div>
        </div>
      )
    }

    return (
      <React.Fragment>
        <h1>Feed</h1>
        {this.props.feedQuery.feed &&
          this.props.feedQuery.feed.map(post => (
            <Post
              key={post.id}
              post={post}
              refresh={() => this.props.feedQuery.refetch()}
              isDraft={!post.isPublished}
            />
          )))
        {this.props.children}
      </React.Fragment>
    )
  }
}

const FEED_QUERY = gql`query FeedQuery {
  feed {
    id
    text
    title
    isPublished
  }
}`

export default graphql(FEED_QUERY, {
  name: 'feedQuery', // name of the injected prop: this.props.feedQuery...
  options: {
    fetchPolicy: 'network-only',
  },
})(FeedPage)
```

A mutation to create new `Post` elements is performed in the `CreatePage` component:

```
import React from 'react'
import { withRouter } from 'react-router-dom'
import { graphql } from 'react-apollo'
```

```

import gql from 'graphql-tag'

class CreatePage extends React.Component {
  state = {
    title: '',
    text: '',
  }

  render() {
    return (
      <div className="pa4 flex justify-center bg-white">
        <form onSubmit={this.handleSubmit}>
          <h1>Create Draft</h1>
          <input
            autoFocus
            className="w-100 pa2 mv2 br2 b--black-20 bw1"
            onChange={e => this.setState({ title: e.target.value })}
            placeholder="Title"
            type="text"
            value={this.state.title}
          />
          <textarea
            className="db w-100 ba bw1 b--black-20 pa2 br2 mb2"
            cols={50}
            onChange={e => this.setState({ text: e.target.value })}
            placeholder="Content"
            rows={8}
            value={this.state.text}
          />
          <input
            className={`pa3 bg-black-10 bn ${this.state.text &&
              this.state.title &&
              'dim pointer'}`}
            disabled={!this.state.text || !this.state.title}
            type="submit"
            value="Create"
          />{' '}
          <a className="f6 pointer" onClick={this.props.history.goBack}>
            or cancel
          </a>
        </form>
      </div>
    )
  }

  handlePost = async e => {
    e.preventDefault()
    const { title, text } = this.state
    await this.props.createDraftMutation({
      variables: { title, text },
    })
    this.props.history.replace('/drafts')
  }
}

const CREATE_DRAFT_MUTATION = gql`mutation CreateDraftMutation($title: String!, $text: String!) {
  createDraft(title: $title, text: $text) {
    id
    title
    text
  }
}`

const CreatePageWithMutation = graphql(CREATE_DRAFT_MUTATION, {

```

```
    name: 'createDraftMutation', // name of the injected prop: this.props.createDraftMutation...
})(CreatePage)

export default withRouter(CreatePageWithMutation)
```

Next Steps & Resources

- **GraphQL server basics:** Check out this tutorial series about GraphQL servers:
 - [GraphQL Server Basics \(Part 1\): The Schema](#)
 - [GraphQL Server Basics \(Part 2\): The Network Layer](#)
 - [GraphQL Server Basics \(Part 3\): Demystifying the info Argument in GraphQL Resolvers](#)
 - [Tutorial: How to build a GraphQL server with graphql-yoga](#)
- **Server deployment:** You can deploy your GraphQL servers to the web using [Now](#) or [Up](#)
- **Prisma database service:** Learn more about the ideas behind Prisma and best practices for building GraphQL servers [here](#)

Using Sentry

Sentry is a crash reporting and aggregation platform that provides you with "real-time insight into production deployments with info to reproduce and fix crashes".

It notifies you of exceptions that your users run into while using your app and organizes for you to triage from their web dashboard. Reported exceptions include sourcemapped stacktraces and other relevant context (device id, platform, Expo verison, etc.) automatically; you can also provide other context that is specific to your application, like the current route and user id.

Why Sentry?

- Sentry treats React Native as a first-class citizen and we have collaborated with Sentry to make sure Expo is too.
- It's easy to set up and use.
- It scales to meet the demands of even the largest projects.
- It works on most platforms, so you can use the same service for reporting your server, CLI, or desktop app errors as you use for your Expo app.
- We trust it for our projects at Expo.
- It is free for up to 10,000 events per month.

Add Sentry to your Expo project

Sign up for a Sentry account and create a project

- [Sign up for a Sentry account](#)
- Once you have signed up, you will be prompted to create a project. Enter the name of your project and continue.
- Copy your "Public DSN", you will need it shortly.
- Go to the [Sentry API](#) section and create an auth token. You can use the default configuration, this token will never be made available to users of your app. Ensure you have `project:write` selected under scopes. Copy your auth token and save it for later.
- Go to your project dashboard by going to [sentry.io](#) and selecting your project. Next go to the settings tab and copy the name of your project, we will need this. The "legacy name" will not work for our purposes.
- Go to your organization settings by going to [sentry.io](#), press the button in the top left of your screen with the arrow beside it and select "organization settings". Copy the name of your organization. The "legacy name" will not work for our purposes.

Install and configure Sentry

- Make sure you're using a new version of Node which supports async/await (Node 7.6+)
- In your project, install the Expo integration: `npm i sentry-expo --save`
- Add the following in your app's main file (`App.js` by default).

```
import Sentry from 'sentry-expo';

// Remove this once Sentry is correctly setup.
```

```
Sentry.enableInExpoDevelopment = true;

Sentry.config('your Public DSN goes here').install();
```

- Open `app.json` and add a `postPublish` hook :

```
{
  "expo": {
    // ... your existing configuration

    "hooks": {
      "postPublish": [
        {
          "file": "sentry-expo/upload-sourcemaps",
          "config": {
            "organization": "your organization's short name here",
            "project": "your project name here",
            "authToken": "your auth token here"
          }
        }
      ]
    }
  }
}
```

The correct `authToken` value can be generated from the [Sentry API page](#).

Publish your app with sourcemaps

With the `postPublish` hook in place, now all you need to do is hit publish and the sourcemaps will be uploaded automatically. We automatically assign a unique release version for Sentry each time you hit publish, based on the version you specify in `app.json` and a release id on our backend -- this means that if you forget to update the version but hit publish, you will still get a unique Sentry release. If you're not familiar with publishing on Expo, you can [read more about it here](#).

Error reporting semantics

In order to ensure that errors are reported reliably, Sentry defers reporting the data to their backend until the next time you load the app after a fatal error rather than trying to report it upon catching the exception. It saves the stacktrace and other metadata to `AsyncStorage` and sends it immediately when the app starts.

Disabled by default in dev

Unless `Sentry.enableInExpoDevelopment = true` is set before calling `sentry.config({...}).install()`, all your dev/local errors will be ignored and only app releases will report errors to Sentry. You can call methods like `Sentry.captureException(new Error('Oops!'))` but these methods will be no-op.

Learn more about Sentry

Sentry does more than just catch fatal errors, learn more about how to use Sentry from their [JavaScript usage docs](#).

Create a Splash Screen

A splash screen, also known as a launch screen, is the first screen that a user sees when opening your app, and it stays visible while the app is loading. You can control when the splash screen disappears by using the [AppLoading component](#) or [SplashScreen module](#).

Customize the splash screen for your app

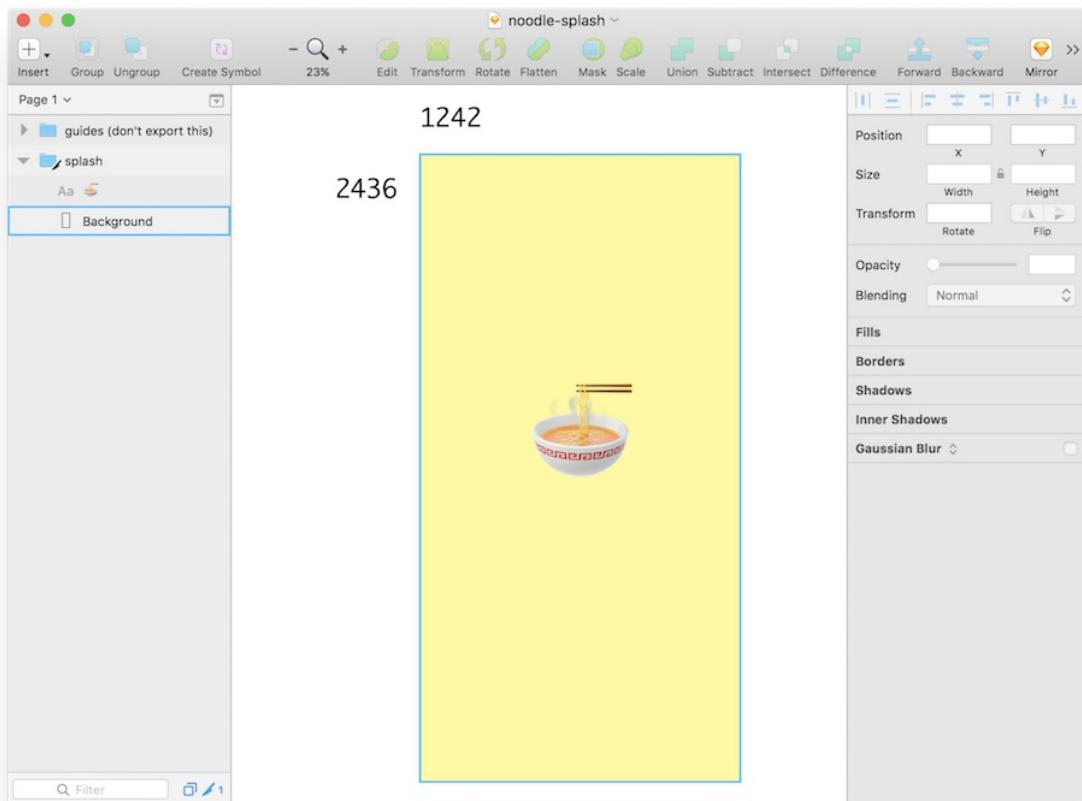
The default splash screen is a blank white screen. This might work for you, if it does, you're in luck! If not, you're also in luck because it's quite easy to customize using `app.json` and the `splash` key. Let's walk through it.

Make a splash image

The [iOS Human Interface Guidelines](#) list the static launch image sizes. I'll go with 1242 pixels wide and 2436 pixels tall -- this is the width of the iPhone 8 Plus (the widest iPhone) and the height of the iPhone X (the tallest iPhone). Expo will resize the image for you depending on the size of the device, and we can specify the strategy used to resize the image with `splash.resizeMode`.

Android screen sizes vary greatly with the massive variety of devices on the market. One strategy to deal with this is to look at the most common resolutions and design around that - [you can see some statistics on this published by Unity here](#). Given that we can resize and crop our splash image automatically, it looks like we can stick with our dimensions, as long as we don't depend on the splash image fitting the screen exactly. This is convenient because we can use one splash image for both iOS and Android - less for you to read in this guide and less work for you to do.

You can work off of [this Sketch template](#) if you like. I did, and I changed the background color to a faint yellow and put a Noodle emoji in the middle. It's worth noting that the splash image supports transparency, although we didn't use it here.



Export the image as a PNG and put it in your project directory. I'll assume it's in the `assets` directory and named `splash.png`.

`splash.image`

Open your `app.json` and add the following inside of the `"expo"` field:

```
"splash": {  
  "image": "./assets/splash.png"  
}
```

Now re-open the Expo client and open your app, and you should see your beautiful splash screen. There may be a delay before it shows up, see "[Differences between environments](#)" below for more information on that.

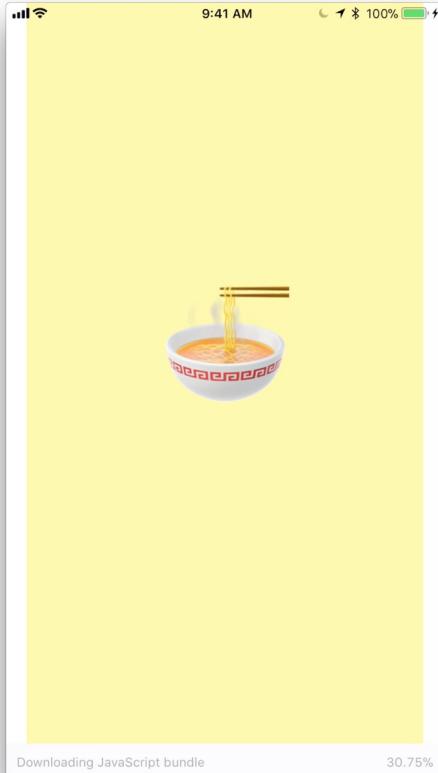
Note: It's required to close and re-open the Expo client app on iOS in order to see changes to the splash screen in the manifest. This is a known issue that we are working to resolve. On Android, you need to press the refresh button from the notification drawer.

`splash.backgroundColor`

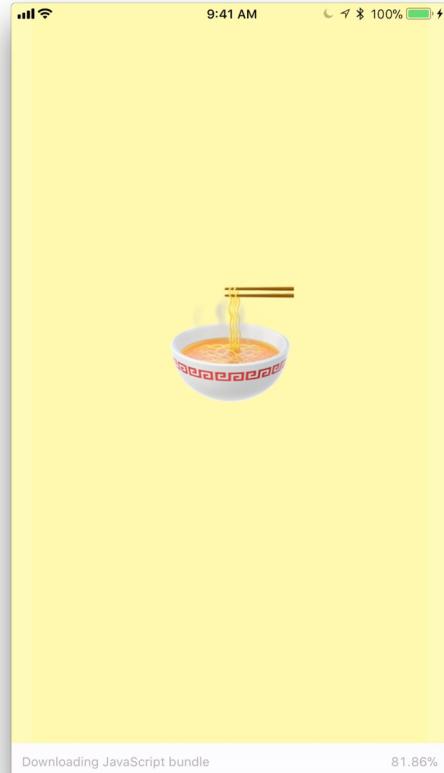
If you set a background color other than white for your splash image, you may see white border around it. This is due to the `splash.resizeMode` property (which we will discuss shortly) and the default background color, which is `#ffffff` (white). Let's resolve this by setting the `splash.backgroundColor` to be the same as our splash image background color.

```
"splash": {  
  "image": "./assets/splash.png",  
  "backgroundColor": "#FEF9B0"  
}
```

Without
splash.backgroundColor

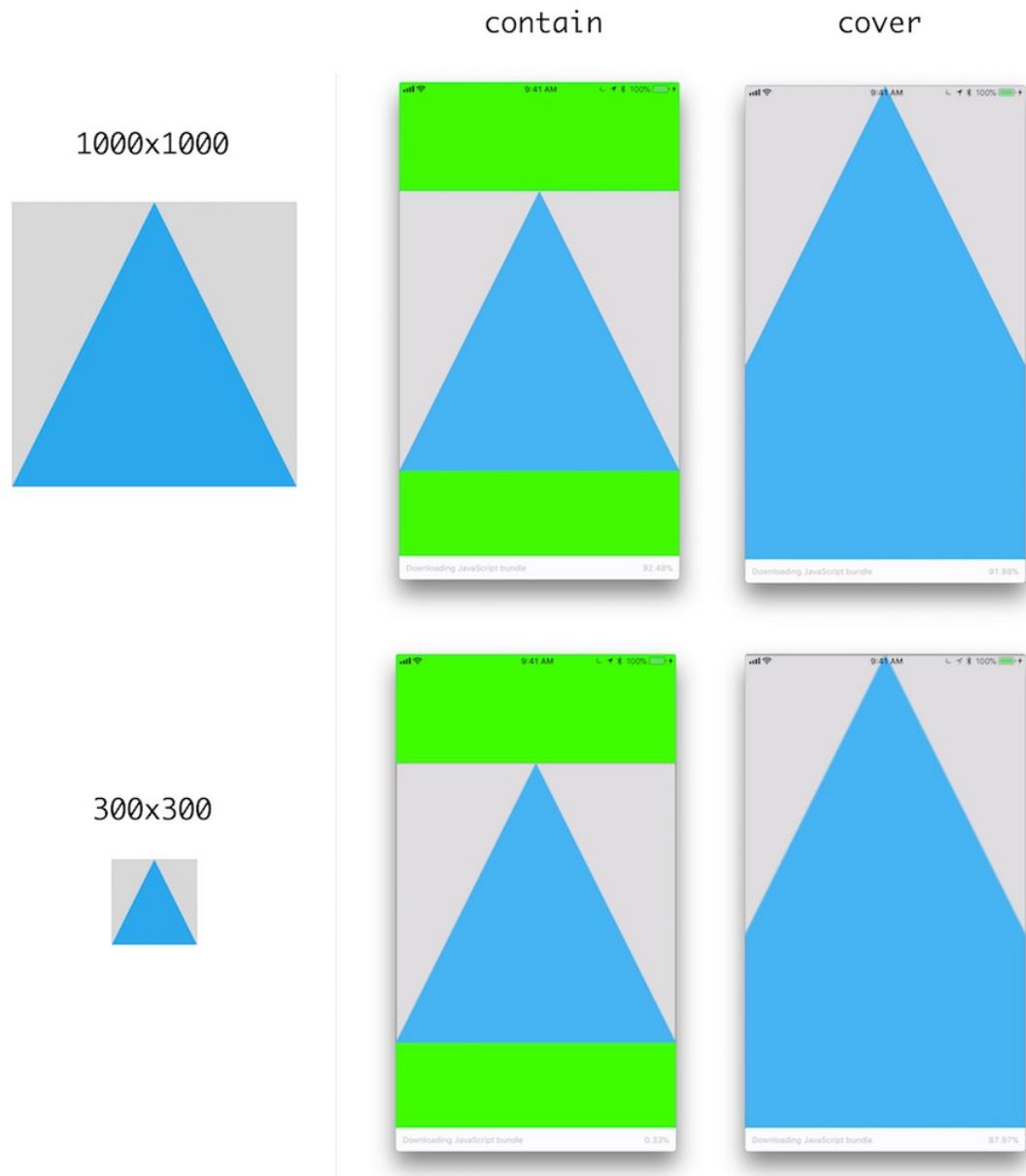


With splash.backgroundColor set
to the same yellow as the image



splash.resizeMode

Any splash image that you provide will be resized to maintain its aspect ratio and to fit the resolution of the user's device. There are two strategies that can be used for resizing: `contain` (default) and `cover`. In both cases, the splash image is within the splash screen. These work the same as the React Native `<Image>` component's `resizeMode` style equivalents, as demonstrated in the following diagram.



Applying this to our noodles example, let's remove the `backgroundColor` and try it out:

```

"splash": {
  "image": "./assets/splash.png",
  "resizeMode": "cover"
}
  
```

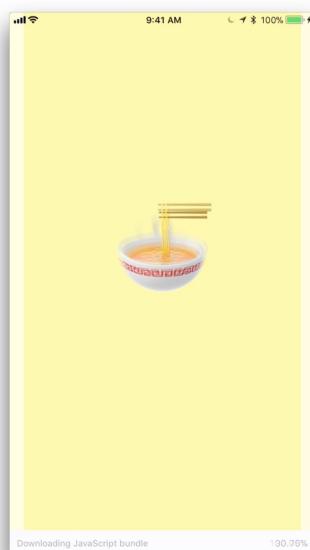
(1) resizeMode contain
with white backgroundColor



(2) resizeMode cover
with white backgroundColor



Image (1) overlaid
on image (2)



Notice that in the last example, we stretched the image to fill the entire width, while maintaining the aspect ratio, and so the noodles emoji ended up being larger than it was when `resizeMode` was set to `contain`. If you are still unclear about the difference between `contain` and `cover`, [this blog post describes precisely what they mean](#).

Customizing the configuration for iOS and Android

Any of the splash options can be configured on a per-platform basis by nesting the configuration under the `android` or `ios` keys within `app.json` (the same as how you would customize an icon for either platform). In addition to this, certain configuration options are only available on iOS or Android.

- On iOS, you can set `ios.splash.tabletImage` if you would like to have a different splash image on iPads.
- On Android, you can set splash images for [different device DPIs](#), from `mdpi` to `xxxhdpi`.

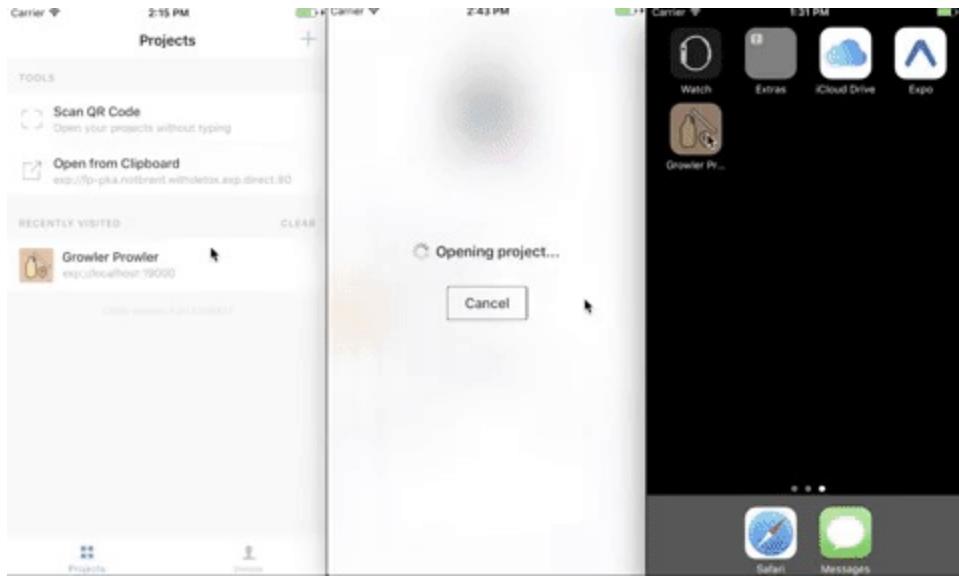
Using `AppLoading` and/or `SplashScreen`

As long as `AppLoading` is the only component rendered in your application, your splash screen will remain visible. We recommend using `AppLoading` while caching assets or fetching any data from `AsyncStorage` to set the app up. However, if you want to control the moment of splash screen visibility change use `SplashScreen`.

Read more about [AppLoading](#) and [SplashScreen](#).

Differences between environments - iOS

Your app can be opened from the Expo client or in a standalone app, and it can be either published or in development. There are slight differences in the splash screen behavior between these environments.



- **On the left**, we are in the Expo client and loading an app that is currently in development. Notice that on the bottom of the splash screen you see an information bar that shows information relevant to preparing the JavaScript and downloading it to the device. We see an orange screen before the splash image appears, because the background color is set immediately but the image needs to be downloaded.
- **In the middle**, we are in the Expo client and we are loading a published app. Notice that again the splash image does not appear immediately.
- **On the right**, we are in a standalone app. Notice that the splash image appears immediately.

Differences between environments - Android

Splash screen behaves in most cases exactly the same as in iOS case.

There is a slight difference when it comes down to **standalone Android applications**. In this scenario extra attention should be paid to `android.splash` section configuration inside `app.json`.

Depending on the `resizeMode` you will get the following behavior:

- **cover** - In this mode your app will be leveraging Android's ability to present a static bitmap at the very beginning of the application start. Unfortunately, Android (unlike iOS) is not supporting stretching provided image, so the application will just present given image centered on the screen. By default `splash.image` would be used as the `mdpi` resource. It's up to you to provide graphics that meet your expectations and fit the screen dimension. To achieve this, use different resolutions for **different device DPIs**, from `mdpi` to `xxxhdpi`.
- **contain** - As described in `cover` mode it isn't possible to dynamically adjust image to the screen size at the very beginning of the application start. Therefore, in this mode, at first only background color will be presented and then, when some view hierarchy is mounted, `splash.image` will be shown.

Ejected ExpoKit apps

If you run `expo eject` (choosing the ExpoKit option) on iOS and your app already uses the splash API, the resulting ExpoKit project will contain an Interface Builder launch screen file configured correctly for your app. After this, if you want to make changes to your app's splash screen, just edit the Interface Builder file directly.

For people who run `expo eject` without the splash API, we add `isSplashScreenDisabled: YES` in your EXShell plist (iOS). If you later decide to use a splash screen in your ejected iOS project, add an Interface Builder file called `LaunchScreen` to your Xcode project, and delete this key from the plist.

Known issues

The following exists are known to us and will be resolved shortly.

- iOS splash screen status bar is white in standalone apps but dark in Expo client. It should be dark in standalone apps by default too, and also it should be customizable.

Migrating from the `loading` API

The `loading` API is deprecated as of SDK 22 and has a strictly worse user experience, so we recommend you change over to `splash` as soon as you have time - the `loading` API will be removed in favor of `splash` in SDK 25.

Setting up Continuous Integration

Continuous Integration (CI) and Continuous Delivery (CD) are concepts which can help you to build and deploy with confidence. It's the idea of automating as much as you can, like running tests or creating new releases.

CI/CD is a relatively broad idea and can get as complex as you can make it. In this guide, we will create a basic setup for testing (CI) and deployments (CD). Also, the configuration for Bitbucket Pipelines, Gitlab CI, and Travis CI are provided. Other CI/CD vendors can be used too; everything is executable through CLI.

Test with Jest

Testing is an essential part of an average CI workflow. This process gives you the confidence when shipping by running all automated tests on every relevant change. Without this, you can't be sure if a proposed change breaks the expected behavior of any existing functionality.

Before we can run the tests, we need to install the dependencies. You can use yarn, npm-install or even the faster npm-ci command. After this, we can run the tests with Jest. Unfortunately, we can't use the original npm-test script shipped with expo-cli. This script is designed to start a daemon that watches for file changes and reruns the tests. In CI environments we need Jest to run the tests once and exit with a (un)successful status code. Also, it's also a good idea to explicitly tell Jest it is running in a CI environment. Jest will handle snapshots more strictly.

To summarize we will set up the CI to run the following two scripts.

```
$ npm ci  
$ npx jest --ci
```

- ▶ Travis CI
- ▶ Gitlab CI
- ▶ Bitbucket Pipelines

Improving Jest performance

As you might have noticed already, the tests in CI are a bit slower compared to running them locally. It's slower because your hardware is more powerful than the CI hardware. Jest can leverage the use of parallel testing with such equipment. Also, Some vendors limit the hardware resources or offer "premium" services for more power. Luckily there is a relatively easy way to improve the speed of Jest; using the power of caching.

There is no definitive way of telling how much it improves. Using the expo-cli tabs project as an example, it can speed up by a factor of 4x - 5x.

- ▶ Travis CI
- ▶ Gitlab CI
- ▶ Bitbucket Pipelines

Deploy to Expo

Now that we have a proper CI workflow in place, we will focus on the Continuous Deployment (CD) part. In this process, we will make a new build and push it to Expo. Combined with Over The Air (OTA) updates, this can create a simple but effective CD infrastructure. Just like the CI part, we first need to install the dependencies. After this, we need to authenticate at Expo and "publish" a new build.

Prepare Expo CLI

To interact with the Expo API, we need to install the Expo CLI. You can use an environment with this library preinstalled, or you can add it to the project as a development dependency. The latter is the easiest way but might increase the installation time. For vendors that charge you per minute, it might be worth creating a prebuilt environment.

To install the Expo CLI into your project, you can execute this script.

```
$ npm install --save-dev expo-cli
```

Prepare authentication

Next, we will configure the publishing step of your application to Expo. Before we can do this, we need to authenticate as the owner of the app. This is possible by storing the username or email with the password in the environment. Every vendor has implemented their storage mechanism for sensitive data like passwords, although they are very similar. Most vendors make use of environment variables which are "injected" into the environment and scrubbed from the logs to keep it safe.

To perform the authentication, we will add this script to our configuration:

```
$ npx expo login -u <EXPO USERNAME> -p <EXPO PASSWORD>
```

Publish new builds

After having the CLI library and authentication in place, we can finally create the build step. In this step, we will create a new build and send it to Expo. It finalizes the whole workflow of creating, testing and shipping your application.

To create the builds, we will add this script to our configuration:

```
$ npx expo publish --non-interactive
```

- ▶ Travis CI
- ▶ Gitlab CI
- ▶ Bitbucket Pipelines
- ▶ CircleCI

Next steps

CI and CD are concepts which are far from fully covered in this guide. The best thing you can do to get familiar with these subjects is to make stuff yourself. Here are some extra links that might help you further.

Useful subjects

- Release channels
- Building standalone apps
- Configuring OTA Updates

Official documentation CI/CD vendors

- [Gitlab CI](#)
- [Travis CI](#)
- [Bitbucket Pipelines](#)

Extra tutorials

- [Setting up Expo and Bitbucket Pipelines](#)

Example repositories from this guide

- [Github](#)
- [Gitlab](#)
- [Bitbucket](#)

Testing on physical devices

The best way to interactively test and verify the behavior and feel of your app as you change code is by loading it in the Expo Client App, as described in other guides.

There are several ways to get the client app into a test environment.

Install Expo Client App from a device's App Store

This is the simplest and easiest way, and the one we recommend for all developers.

Build Expo Client App on your computer and side-load it onto a device

[The client apps are open-source](#), and their readme contains instructions on how to compile them locally and install them onto devices attached to your computer by a USB cable.

Run Expo Client App on your computer in a device Emulator/Simulator

[The client apps github repository](#) also includes instruction on how to build and run the iOS Client App in the Xcode Device Simulator, and the Android Client App in the Android Studio Device Emulator.

Using Firebase

Firebase Database is a popular NoSQL cloud database that allows developers realtime synchronization of live data. With multi-platform support, synchronizing data between users and clients is pretty seamless, but it can also be used more generally as a generic persistent NoSQL data backing if you don't care about realtime updates. It has a very flexible rules syntax to allow minute control over data access as well.

Luckily, the Firebase JavaScript SDK starting from version 3.1+ has almost full support for React Native, so adding it to our Expo app is super easy. The one caveat covered later in this guide is that the user login components typically provided by the Firebase SDKs will **not** work for React Native, and thus we will have to work around it.

See firebase.google.com/docs/database for more general information and the [official Firebase blog post announcing React Native compatibility](#)

Note: This guide only covers Firebase Realtime Database, and not the other services under the larger Google Firebase umbrella. Firebase Cloud Storage is currently not supported, but we are [working on upstreaming a Blob implementation](#) to React Native that would make this possible. For more background on why other Firebase services are not supported, please read [Brent Vatne's response on Canny](#)

1. Firebase SDK Setup

First we need to setup a Firebase Account and create a new project. We will be using the JavaScript SDK provided by Firebase, so pull it into your Expo project.

```
npm install --save firebase .
```

The Firebase console will provide you with an api key, and other identifiers for your project needed for initialization. [firebase-web-start](#) has a detailed description of what each field means and where to find them in your console.

```
import * as firebase from 'firebase';

// Initialize Firebase
const firebaseConfig = {
  apiKey: "<YOUR-API-KEY>",
  authDomain: "<YOUR-AUTH-DOMAIN>",
  databaseURL: "<YOUR-DATABASE-URL>",
  storageBucket: "<YOUR-STORAGE-BUCKET>"
};

firebase.initializeApp(firebaseConfig);
```

Temporarily Bypass Default Security Rules

By default Firebase Database has a security rule setup such that all devices accessing your data must be authenticated. We obviously haven't setup any authentication yet, so we can disable it for now while we setup the rest of our app.

Go into the Firebase console for Database, and under the Rules tab you should see a default set of rules already provided for you. Change the rules to:

```
{  
  "rules": {  
    ".read": true,  
    ".write": true  
  }  
}
```

See [Sample Firebase Rules](#) for good sets of rules for your data, including unauthenticated.

Note It is important to note that this is temporary for development, and these rules should be thoroughly assessed before releasing an application.

2. Storing Data and Receiving Updates

Storing data through Firebase can be pretty simple. Imagine we're creating a game where highscores are stored in Firebase for everyone to see. We could create a users bucket in our data that is referenced by each user. Setting their highscore would be straightforward.

```
function storeHighScore(userId, score) {  
  firebase.database().ref('users/' + userId).set({  
    highscore: score  
  });  
}
```

Now let's say we wanted another client to listen to updates to the high score of a specific user. Firebase allows us to set a listener on a specific data reference and get notified each time there is an update to the data. In the example below, every time a highscore is updated for the given user, it will print it to console.

```
setupHighscoreListener(userId) {  
  firebase.database().ref('users/' + userId).on('value', (snapshot) => {  
    const highscore = snapshot.val().highscore;  
    console.log("New high score: " + highscore);  
  });  
}
```

3. User Authentication

This was all pretty simple and works fairly out of the box for what Firebase JavaScript SDK provides. There is one caveat however. We skipped the authentication rules for simplicity at the beginning. Firebase SDKs provide authentication methods for developers, so they don't have to reimplement common login systems such as Google or Facebook login.

This includes UI elements in the Web, Android, and iOS SDK versions for Firebase, however, these UI components do not work with React Native and **should not** be called. Thankfully, Firebase gives us ways to authenticate our data access given that we provide user authentication ourselves.

Login Methods

We can choose different login methods that make sense to our application. The login method choice is orthogonal to the Firebase Database access, however, we do need to let Firebase know how we have setup our login system such that it can correctly assign authentication tokens that match our user accounts for data access

control. You can use anything you want, roll your own custom login system, or even forego it altogether if all your users can have unrestricted access.

Facebook Login

A common login system many developers opt for is a simple Facebook login that users are already familiar with. Expo provides a great Facebook login component already, so we just need to plug that in.

See the Facebook section of our docs for information on how to set this up. This works just as well with Google and [several others](#)).

Tying Sign-In Providers with Firebase

Once you have added Facebook login to your Expo app, we need to adjust the Firebase console to check for it. Under the Authentication section in the console in the Sign-In Method tab, enable Facebook as a sign-in provider.

You can add whichever provider makes sense for you, or even add multiple providers. We will stick with Facebook for now since we already have a simple drop-in Expo component already built.

Reenable Data Access Security Rule

We need to re-enable the Data Security Rule in our Firebase console again to check for user authentication. This time our rules will be slightly more complicated.

For our example, let's say we want everyone to be able to read the high score for any user, but we want to restrict writes to only the user who the score belongs to. You wouldn't want anyone overwriting your highscore, would you?

```
{
  "rules": {
    "users": {
      "$uid": {
        ".read": true,
        ".write": "$uid === auth.uid"
      }
    }
  }
}
```

Listening for Authentication

We are now ready to connect the Facebook login code with our Firebase Database implementation.

```
firebase.initializeApp(config);

// Listen for authentication state to change.
firebase.auth().onAuthStateChanged((user) => {
  if (user != null) {
    console.log("We are authenticated now!");
  }

  // Do other things
});

async function loginWithFacebook() {
  const { type, token } = await Expo.Facebook.logInWithReadPermissionsAsync(
```

```

'<APP_ID>',
{ permissions: ['public_profile'] }
);

if (type === 'success') {
  // Build Firebase credential with the Facebook access token.
  const credential = firebase.auth.FacebookAuthProvider.credential(token);

  // Sign in with credential from the Facebook user.
  firebase.auth().signInWithCredential(credential).catch((error) => {
    // Handle Errors here.
  });
}
}

```

The Facebook login method is similar to what you see in the Facebook login guide, however, the token we receive from a successful login can be passed to the Firebase SDK to provide us with a Firebase credential via `firebase.auth.FacebookAuthProvider.credential`. We can then sign-in with this credential via `firebase.auth().signInWithCredential`.

The `firebase.auth().onAuthStateChanged` event allows us to set a listener when the authentication state has changed, so in our case, when the Facebook credential is used to successfully sign in to Firebase, we are given a user object that can be used for authenticated data access.

Authenticated Data Updates

Now that we have a user object for our authenticated user, we can adapt our previous `storeHighScore()` method to use the `uid` of the user object as our user reference. Since the `user.uid`'s are generated by Firebase automatically for authenticated users, this is a good way to reference our users bucket.

```

function storeHighScore(user, score) {
  if (user != null) {
    firebase.database().ref('users/' + user.uid).set({
      highscore: score
    });
  }
}

```

Overview

Distributing your app to your users is a critical process in the development of any product. This documentation will cover topics related to app distribution, such as how to:

- [Create native builds](#) you can submit to the Apple App Store and Google Play Store
- Prepare your app for [deployment to the App Store and Play Store](#) and avoid common rejections.
- Manage different [release environments](#) for your app

Building Standalone Apps

The purpose of this guide is to help you create standalone binaries of your Expo app for iOS and Android which can be submitted to the Apple App Store and Google Play Store.

An Apple Developer account is needed to build an iOS standalone app, but a Google Play Developer account is not needed to build the Android standalone app. If you'd like to submit to either app store, you will need a developer account on that store.

It's a good idea to read the best practices about [Deploying to App Stores](#) to ensure your app is in good shape to get accepted into the Apple and Google marketplaces. We can generate builds for you, but it's up to you to make your app awesome.

1. Install Expo CLI

Expo CLI is the tool for developing and building Expo apps. Run `npm install -g expo-cli` (or `yarn global add expo-cli`) to get it.

If you haven't created an Expo account before, you'll be asked to create one when running the build command.

Windows users must have WSL enabled. We recommend picking Ubuntu from the Windows Store. Be sure to Launch Ubuntu at least once. After that, use an Admin powershell to run: `Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux`

2. Configure app.json

```
{
  "expo": {
    "name": "Your App Name",
    "icon": "./path/to/your/app-icon.png",
    "version": "1.0.0",
    "slug": "your-app-slug",
    "sdkVersion": "XX.0.0",
    "ios": {
      "bundleIdentifier": "com.yourcompany.yourappname"
    },
    "android": {
      "package": "com.yourcompany.yourappname"
    }
  }
}
```

- The iOS `bundleIdentifier` and Android `package` fields use reverse DNS notation, but don't have to be related to a domain. Replace `"com.yourcompany.yourappname"` with whatever makes sense for your app.
- You're probably not surprised that `name`, `icon` and `version` are required.
- `slug` is the url name that your app's JavaScript is published to. For example: `expo.io/@community/native-component-list`, where `community` is my username and `native-component-list` is the slug.
- The `sdkVersion` tells Expo what Expo runtime version to use, which corresponds to a React Native version. Although `"xx.0.0"` is listed in the example, you already have an `sdkVersion` in your app.json and should not change it except when you want to update to a new version of Expo.

There are other options you might want to add to `app.json`. We have only covered what is required. For example, some people like to configure their own build number, linking scheme, and more. We highly recommend you read through [Configuration with app.json](#) for the full spec. This is also your last chance to double check our [recommendations](#) for App Store metadata.

3. Start the build

Run `expo build:android` or `expo build:ios`. If you don't already have a packager running for this project, `expo` will start one for you.

If you choose to build for Android

The first time you build the project you will be asked whether you'd like to upload a keystore or have us handle it for you. If you don't know what a keystore is, you can have us generate one for you. Otherwise, feel free to upload your own.

If you choose to let Expo generate a keystore for you, we **strongly recommend** that you later run `expo fetch:android:keystore` and backup your keystore to a safe location. Once you submit an app to the Google Play Store, all future updates to that app **must** be signed with the same keystore to be accepted by Google. If, for any reason, you delete your project or clear your credentials in the future, you will not be able to submit any updates to your app if you have not backed up your keystore.

```
[exp] No currently active or previous builds for this project.
```

```
Would you like to upload a keystore or have us generate one for you?
```

```
If you don't know what this means, let us handle it! :)
```

- 1) Let Expo handle the process!
- 2) I want to upload my own keystore!

Note: If you choose the first option and later decide to upload your own keystore, we currently offer an option to clear your current Android keystore from our build servers by running `expo build:android --clear-credentials`. **This is irreversible, so only run this command if you know what you are doing!** You can download a backup copy of the keystore by running `expo fetch:android:keystore`. If you do not have a local copy of your keystore, you will be unable to publish new versions of your app to the Play Store. Your only option would be to generate a new keystore and re-upload your application as a new application. You can learn more about how code signing and keystores work [in the Android documentation](#).

If you choose to build for iOS

You are given a choice of letting the `expo` client create the necessary credentials for you, while still having a chance to provide your own overrides. Your Apple ID and password is used locally and never saved on Expo's servers.

```
$ expo build:ios
[16:44:37] Checking if current build exists...

[16:44:37] No currently active or previous builds for this project.
[16:44:37]
We need your Apple ID/password to manage certificates, keys
and provisioning profiles from your Apple Developer account.
```

```
Note: Expo does not keep your Apple ID or your Apple ID password.
```

```
? What's your Apple ID? xxx@yyy.zzz
? Password? [hidden]
✓ Authenticated with Apple Developer Portal successfully!
[16:44:46] You have 4 teams associated with your account
? Which team would you like to use? 3) ABCDEFGHIJ "John Turtle" (Individual)
✓ Ensured App ID exists on Apple Developer Portal!
[16:44:59] We do not have some credentials for you: Apple Distribution Certificate, Apple Push Notifications service
key, Apple Provisioning Profile
? How would you like to upload your credentials? (Use arrow keys)
> Expo handles all credentials, you can still provide overrides
  I will provide all the credentials and files needed, Expo does limited validation
```

We ask you if you'd like us to handle your Distribution Certificate or use your own. If you have previously used `expo-cli` for building a standalone app for a different project, then we'll ask you if you'd like to reuse your existing Distribution Certificate. Similar to the Android keystore, if you don't know what a Distribution Certificate is, just let us handle it for you. If you do need to upload your own certificates, we recommend following [this excellent guide on making a P12 file](#). **Note:** this guide recommends leaving the P12's password blank, but a P12 password is required to upload your own certificate to Expo's service. Please enter a password when prompted. We'll ask you similar questions about your choice regarding Push Notifications service key. Remember that Push Notifications service keys can be reused across different Expo apps as well.

Note: The Expo build service supports both normal App Store distribution as well as enterprise distribution. To use the latter, you must be a member of the "[Apple Developer Enterprise Program](#)". Only normal Apple developer accounts can build apps that can be submitted to the Apple App Store, and only enterprise developer accounts can build apps that can be distributed using enterprise distribution methods. When you call `expo build:ios`, you just need to choose the correct team, it will be labeled `(In-House)`. At this time, the standalone app builder does not support "ad hoc" distribution certificates or provisioning profiles.

Switch to Push Notification Key on iOS

If you are using Push Notifications Certificate and want to switch to Push Notifications Key you need to start build with `--clear-push-cert`. We will remove certificate from our servers and generate Push Notifications Key for you.

4. Wait for it to finish building

When one of our building machines will be free, it'll start building your app. You can check how long you'll wait on [Turtle status](#) site. We'll print a url you can visit (such as `expo.io/builds/some-unique-id`) to watch your build logs. Alternatively, you can check up on it by running `expo build:status`. When it's done, you'll see the url of a `.apk` (Android) or `.ipa` (iOS) file -- this is your app. Copy and paste the link into your browser to download the file.

If you would like to, we can also call your webhook once the build has finished. You can set up a webhook for your project using `expo webhooks:set --event build --url <webhook-url>` command. You will be asked to type a webhook secret. It has to be at least 16 characters long and it will be used to calculate the signature of the request body which we send as the value of the `Expo-Signature` HTTP header. You can use the signature to verify a webhook request is genuine. We promise you that we keep your secret securely encrypted in our database.

We call your webhook using an HTTP POST request and we pass data in the request body. Expo sends your webhook with JSON object with following fields:

- `status` - a string specifying whether your build has finished successfully (can be either `finished` OR `errored`)
- `id` - the unique ID of your build

- `artifactUrl` - the URL to the build artifact (we only include this field if the build is successful)

Additionally, we send an `Expo-Signature` HTTP header with the hash signature of the payload. You can use this signature to verify the request is from Expo. The signature is a hex-encoded HMAC-SHA1 digest of the request body, using your webhook secret as the HMAC key.

This is how you can implement your server:

```
import crypto from 'crypto';
import express from 'express';
import bodyParser from 'body-parser';
import safeCompare from 'safe-compare';

const app = express();
app.use(bodyParser.text({ type: '*/*' }));
app.post('/webhook', (req, res) => {
  const expoSignature = req.headers['expo-signature'];
  // process.env.SECRET_WEBHOOK_KEY has to match <webhook-secret> value set with `expo webhooks:set ...` command
  const hmac = crypto.createHmac('sha1', process.env.SECRET_WEBHOOK_KEY);
  hmac.update(req.body);
  const hash = `sha1=${hmac.digest('hex')}`;
  if (!safeCompare(expoSignature, hash)) {
    res.status(500).send("Signatures didn't match!");
  } else {
    // do sth here
    res.send('OK!');
  }
});
app.listen(8080, () => console.log('Listening on port 8080'));
```

You can always change your webhook URL and/or webhook secret using the same command you used to set up the webhook for the first time. To see what your webhook is currently set to, you can use `expo webhooks:show` command. If you would like us to stop sending requests to your webhook, simply run `expo webhooks:clear` in your project.

Note: We enable bitcode for iOS, so the `.ipa` files for iOS are much larger than the eventual App Store download available to your users. For more information, see [App Thinning](#).

5. Test it on your device or simulator

- You can drag and drop the `.apk` into your Android emulator. This is the easiest way to test out that the build was successful. But it's not the most satisfying.
- **To run it on your Android device**, make sure you have the Android platform tools installed along with `adb`, then just run `adb install app-filename.apk` with [USB debugging enabled on your device](#) and the device plugged in.
- **To run it on your iOS Simulator**, first build your expo project with the simulator flag by running `expo build:ios -t simulator`, then download the tarball with the link given upon completion when running `expo build:status`. Unpack the `.tar.gz` by running `tar -xvzf your-app.tar.gz`. Then you can run it by starting an iOS Simulator instance, then running `xcrun simctl install booted <app path>` and `xcrun simctl launch booted <app identifier>`.
- **To test a device build with Apple TestFlight**, download the `.ipa` file to your local machine. You are ready to upload your app to TestFlight. Within TestFlight, click the plus icon and create a New App. Make sure your `bundleIdentifier` matches what you've placed in `app.json`.

Note: You will not see your build here just yet! You will need to use Xcode or Application Loader to upload your IPA first. Once you do that, you can check the status of your build under `Activity`. Processing an app can take 10-15 minutes before it shows up under available builds.

6. Submit it to the appropriate store

Read the guide on [Uploading Apps to the Apple App Store and Google Play](#).

7. Update your app

For the most part, when you want to update your app, just Publish again from Expo CLI. Your users will download the new JS the next time they open the app. To ensure your users have a seamless experience downloading JS updates, you may want to enable [background JS downloads](#). However, there are a couple reasons why you might want to rebuild and resubmit the native binaries:

- If you want to change native metadata like the app's name or icon
- If you upgrade to a newer `sdkVersion` of your app (which requires new native code)

To keep track of this, you can also update the binary's `versionCode` and `buildNumber`. It is a good idea to glance through the [app.json documentation](#) to get an idea of all the properties you can change, e.g. the icons, deep linking url scheme, handset/tablet support, and a lot more.

If you run into problems during this process, we're more than happy to help out! [Join our Forums](#) and let us know if you have any questions.

App signing

Process is automated for iOS and Android, but in both cases you can choose to provide your own overrides. Both `expo build:ios` and `expo build:android` commands generate signed applications ready to be uploaded into respective stores.

Android

How it works

Google requires all Android apps to be digitally signed with a certificate before they are installed on a device or updated. Usually a private key and its public certificate are stored in a keystore. In the past, APKs uploaded to the store were required to be signed with the **app signing certificate** (certificate that will be attached to the app in store), and if the keystore was lost there was no way to recover or reset it. If you opt in to App Signing by Google Play you need to upload an APK signed with an **upload certificate**, and Google Play will strip that signature and replace it with one generated using the **app signing certificate**. Both the upload keystore and keystore with the **app signing key** are essentially the same mechanism, but if your upload keystore is lost or compromised, you can contact the Google Play support team to reset the key.

From the build process's perspective, there is no difference whether an app is signed with an **upload certificate** or an **app signing certificate**. Either way, `expo build:android` will generate an APK signed with the keystore currently assigned to your application. If you want to generate an upload keystore manually, you can do that the same way you created your original keystore.

See [here](#) to find more information about this process.

1. Using App Signing by Google Play (recommended)

Create a new application and allow Google Play to handle your **app signing key**. The certificate used to sign the first APK uploaded to the store will be your **upload certificate** and each new release needs to be signed with it.

If you want to use Google Play App Signing in an existing app, run `expo opt-in-google-play-signing` and follow its instructions. After this process, the original keystore will be backed up to your current working directory and credentials for that keystore will be printed on the screen. Remove that keystore only when you are sure that everything works correctly.

In case you lose your upload keystore (or it's compromised), you can ask Google Support Team to reset your upload key.

- If you want Expo to handle creating the upload certificate:
 - `expo build:android --clear-credentials` and select the option `Let Expo handle the process!`, which generates a new keystore and signs a new APK with it
 - `expo fetch:android:upload-cert` extracts public certificate from the keystore into `.pem` file
 - add the upload certificate to the Google Play console
- If you want to handle it create the upload certificate yourself:
 - Generate a new keystore
 - Extract the upload certificate with

```
keytool -export -rfc  
-keystore your-upload-keystore.jks  
-alias upload-alias  
-file output_upload_certificate.pem
```

- o Add the upload certificate to the Google Play console

2. Signing APKs with an app signing key (deprecated)

The first time you run `expo build:android`, you can choose for Expo to generate a keystore or manually specify all of the required credentials. These credentials are used to sign APKs created by Expo services. generated APKs will signed.

Deploying to App Stores

This guide offers best practices around submitting your Expo app to the Apple iTunes Store and Google Play Store. To learn how to generate native binaries for submission, see [Building Standalone Apps](#).

Although you can share your published project through the Expo Client and on your [expo.io](#) profile, submitting a standalone app to the Apple and Google stores is necessary to have a dedicated piece of real estate on your users' devices. Submitting to these stores carries stronger requirements and quality standards than sharing a toy project with a few friends, because it makes your app available through a much wider distribution platform.

Disclaimer: Especially in the case of Apple, review guidelines and rules change all the time, and Apple's enforcement of various rules tends to be finicky and inconsistent. We can't guarantee that your particular project will be accepted by either platform, and you are ultimately responsible for your app's behavior. However, Expo apps are native apps and behave just like any other apps, so if you've created something awesome, you should have nothing to worry about!

Make sure your app works on many form factors

It's a good idea to test your app on a device or simulator with a small screen (e.g. an iPhone SE) as well as a large screen (e.g. an iPhone X). Ensure your components render the way you expect, no buttons are blocked, and all text fields are accessible.

Try your app on tablets in addition to handsets. Even if you have `ios.supportsTablet: false` configured, your app will still render at phone resolution on iPads and must be usable.

Make app loading seamless

- Add a [splash screen](#), the very first thing your users see after they select your app.
- Use [AppLoading](#) to ensure your interface is ready before the user sees it.
- [Preload and cache your assets](#) so your app loads quickly, even with a poor internet connection.

Play nicely with the system UI

- Configure the [status bar](#) so it doesn't clash with your interface.
- Use [native gestures](#) whenever possible.
- Use interface elements that make sense on the device. For example, see the [iOS Human Interface Guidelines](#).

Tailor your app metadata

- Add a great [icon](#). Icon requirements between iOS and Android differ and are fairly strict, so be sure and familiarize yourself with that guide.
- Customize your [primaryColor](#).
- Make sure your app has a valid iOS [Bundle Identifier](#) and [Android Package](#). Take care in choosing these, as you will not be able to change them later.

- Use `versionCode` and `buildNumber` to distinguish different binaries of your app.

Privacy Policy

- Starting October 3, 2018, all new iOS apps and app updates will be required to have a privacy policy in order to pass the App Store Review Guidelines.
- Additionally, a number of developers have reported warnings from Google if their app does not have a privacy policy, since by default all Expo apps contain code for requesting the Android Advertising ID. Though this code may not be executed depending on which Expo APIs you use, we still recommend that all apps on the Google Play Store include a privacy policy as well.

iOS-specific guidelines

- All apps in the iTunes Store must abide by the [App Store Review Guidelines](#).
- Apple will ask you whether your app uses the IDFA. Because Expo depends on Segment Analytics, the answer is yes, and you'll need to check a couple boxes on the Apple submission form. See [Segment's Guide](#) for which specific boxes to fill in.

Common App Rejections

- It's helpful to glance over [Common App Rejections](#).
- Binaries can get rejected for having poorly formatted icons, so double check the [App Icon guide](#).
- Apple can reject your app if elements don't render properly on an iPad, even if your app doesn't target the iPad form factor. Be sure and test your app on an iPad (or iPad simulator).
- Occasionally people get a message from Apple which mentions an IPv6 network. Typically this is just Apple's way of informing you what kind of network they tested on, and the actual "IPv6" detail is a red herring. All of Expo's iOS code uses `NSURLSession`, which is IPv6-compatible. [More info](#).

System permissions dialogs on iOS

If your app asks for `system permissions` from the user, e.g. to use the device's camera, or access photos, Apple requires an explanation for how your app makes use of that data. Expo will automatically provide a boilerplate reason for you, such as "Allow cool-app to access the camera." If you would like to provide more information, you can override these values using the `ios.infoPlist` key in `app.json`, for example:

```
"infoPlist": {
  "NSCameraUsageDescription": "This app uses the camera to scan barcodes on event tickets."
},
```

The full list of keys Expo provides by default can be seen [here](#). Unlike with Android, on iOS it is not possible to filter the list of permissions an app may request at a native level. This means that by default, your app will ship with all of these default boilerplate strings embedded in the binary. You can provide any overrides you want in the `infoPlist` configuration. Because these strings are configured at the native level, they will only be published when you build a new binary with `expo build`.

Localizing system dialogs on iOS

If your app uses a language besides English, you can optionally provide [localized](#) strings for the system dialogs. For example, in `app.json`, you can provide

```
"locales": {  
    "ru": "./languages/russian.json"  
}
```

...where `russian.json` looks like:

```
{  
    "NSContactsUsageDescription": "Hello Russian words"  
}
```

Release Channels

Introduction

Use release channels in Expo to send out different versions of your application to your users by giving them a URL or configuring your standalone app. You should use release channels if:

- You have an app in production and need a testing environment.
- You have multiple versions of your app.

Publish with Channels

Publish your release by running:

```
expo publish --release-channel <your-channel>
```

with the `exp` cli. Your users can see this release in the Expo client app with a parameterized URL `https://exp.host/@username/yourApp?release-channel=<your-channel>`. If you do not specify a channel, you will publish to the `default` channel.

Build with Channels

Build your standalone app by running

```
expo build:ios --release-channel <your-channel>
expo build:android --release-channel <your-channel>
```

with the `exp` cli. The binary produced will only pull releases published under the specified channel. If you do not specify a channel, your binary will pull releases from the `default` channel.

Access Channel from Code

You can access the channel your release is published under with the `releaseChannel` field in the [manifest object](#).

`Constants.manifest.releaseChannel` does NOT exist in dev mode. It does exist, however when you explicitly publish / build with it.

Example Workflow

Consider a situation where you have a Staging stack for testing on Expo Client, and a Production stack for pushing through TestFlight, then promoting to the AppStore.

On the staging stack, run `expo publish --release-channel staging`. Your test users can see the staging version of your app by specifying the channel in the query parameter of the URL (ie) `https://exp.host/@username/yourApp?release-channel=staging`, then opening the URL in their web browser, and finally scanning the QR code with the Expo client. Alternatively, they can open that URL directly on their mobile device.

On the production stack, release v1 of your app by running `expo publish --release-channel prod-v1`. You can build this version of your app into a standalone ipa by running `expo build:ios --release-channel prod-v1`. You can push updates to your app by publishing to the `prod-v1` channel. The standalone app will update with the most recent compatible version of your app on the `prod-v1` channel.

If you have a new version that you don't want v1 users getting, release v2 of your app by running `expo publish --release-channel prod-v2` and building it with `expo build:ios --release-channel prod-v2`. Users with the `prod-v2` ipa will only be pulling releases from that channel.

You can continue updating v1 of your app with `expo publish --release-channel prod-v1`, and users who haven't updated to the latest `prod-v2` ipa in the Apple App Store will continue receiving the latest `prod-v1` releases.

Using Release Channels with ExpoKit

Since `expo build` does not apply to ExpoKit projects, you can edit the native project's release channel manually by modifying the `releaseChannel` key in `EXShell.plist` (iOS) or the `RELEASE_CHANNEL` value in `AppConstants.java` (Android).

Using Release Channels for Environment Variable Configuration

Environment variables don't exist explicitly, but you can utilize release channels to make that happen!

Say you have a workflow of releasing builds like this:

- `expo publish --release-channel prod-v1`
- `expo publish --release-channel prod-v2`
- `expo publish --release-channel prod-v3`
- `expo publish --release-channel staging-v1`
- `expo publish --release-channel staging-v2`

You can create a function that looks for the specific release and sets the correct variable.

```
function getApiUrl(releaseChannel) {
  if (releaseChannel === undefined) return App.apiUrl.dev // since releaseChannels are undefined in dev, return your default.
  if (releaseChannel.indexOf('prod') !== -1) return App.apiUrl.prod // this would pick up prod-v1, prod-v2, prod-v3
  if (releaseChannel.indexOf('staging') !== -1) return App.apiUrl.staging // return staging environment variables
}
```

Advanced Release Channels

Introduction

For a quick introduction to release channels, read [this](#).

When you publish your app by running `expo publish --release-channel staging`, it creates:

- a release, identified by a `publicationId` for Android and iOS platforms. A release refers to your bundled source code and assets at the time of publication.
- a link to the release in the `staging` channel, identified by a `channelId`. This is like a commit on a git branch.

For simplicity, the rest of this article will refer to just the `ios` releases, but you could swap out `ios` for `android` at any point and everything would still be true.

See past publishes

You can see everything that you've published with `expo publish:history`.

Example command and output

```
expo publish:history --platform ios
```

publishedTime	appVersion	sdkVersion	platform	channel	channelId	publicat
2018-01-05T23:55:04.603Z	1.0.0	24.0.0	ios	staging	9133d577	d9bd6b8

To see more details about this particular release, you can run `expo publish:details`

Example command and output

```
expo publish:details --publish-id d9bd6b80
```

Release Description	
fullName	@quinlanj/test-app
hash	7808b1b734e8b756639df30683c3ec73
packageName	test-app
packageUsername	quinlanj
platform	ios
publishedTime	2018-01-05T23:55:04.603Z
publishingUsername	quinlanj
revisionId	ZddJlK7arm
s3Key	@quinlanj/test-app/1.0.0/7808b1b734e8b756639df30683c3ec73-24.0.0-ios.js
s3Url	https://exp-bundles.s3.amazonaws.com/%40quinlanj/test-app/1.0.0/7808b1b734e8b756639df30683c3ec73-24.0.0-ios.js
sdkVersion	24.0.0
version	1.0.0
publicationId	d9bd6b80

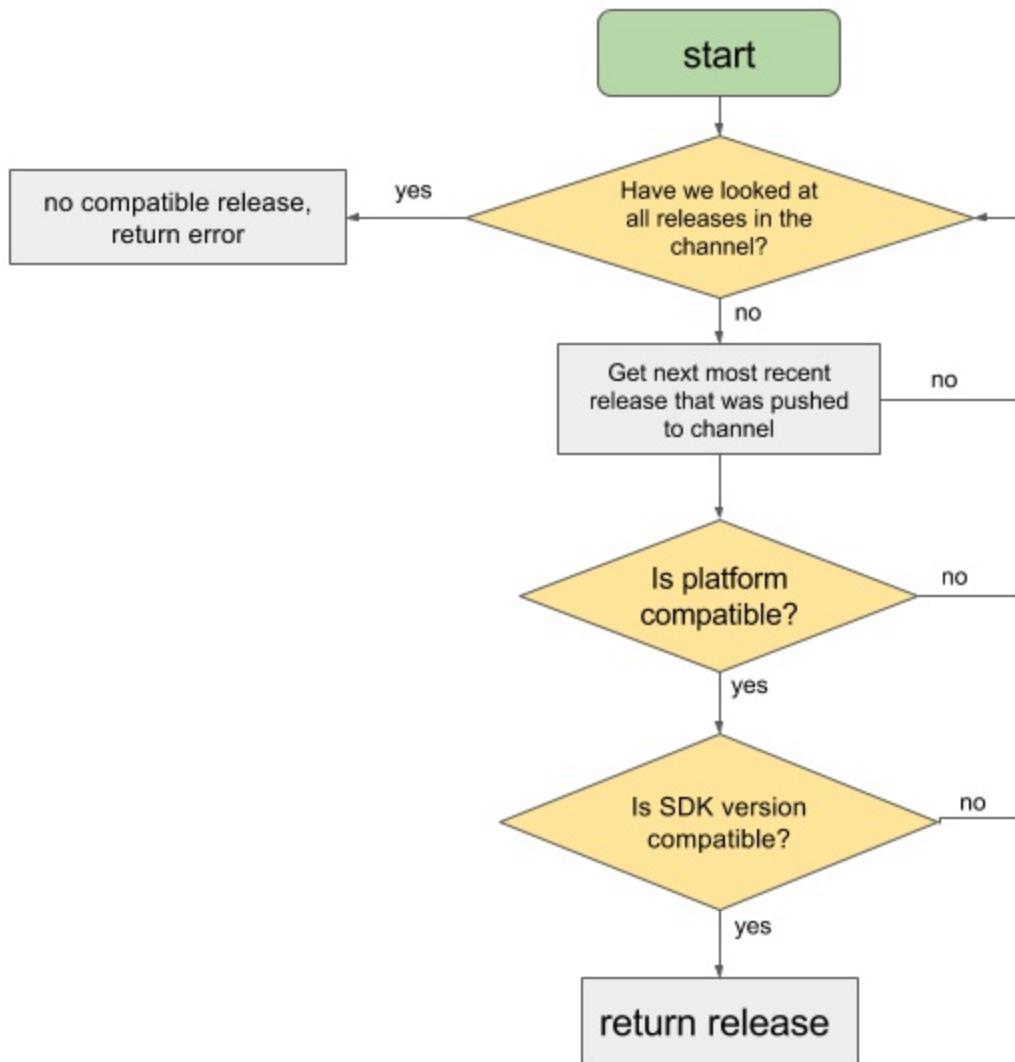
Manifest Details	
description	This project is really great.
icon	./assets/icon.png
iconUrl	https://d1wp6m56sqw74a.cloudfront.net/~assets/fa6577fecc0a7838f15a254577639984
ios	{"supportsTablet":true}
name	test-app
orientation	portrait
platforms	["ios", "android"]
privacy	public
sdkVersion	24.0.0
slug	test-app
splash	{"backgroundColor": "#ffffff", "image": "./assets/splash.png", "imageUrl": "https://d1wp6m56sqw74a.cloudfront.net/~assets/43ec0dcbe5a156bf9e650bb8c15e7af6", "resizeMode": "contain"}
version	1.0.0

What version of the app will my users get?

Your users will get the most recent compatible release that was pushed to a channel. Factors that affect compatibility:

- sdkVersion
- platform

The following flowchart shows how we determine which release to return to a user:



Promoting a release to a new channel

Example use case: you previously published a release to `staging` and everything went well in your testing. Now you want this release to be active in another channel (ie) `production`

We run `expo publish:set` to push our release to the `production` channel. `expo publish:set --publish-id d9bd6b80 --release-channel production`

Continuing from the previous section, we can see that our release is available in both the `staging` and the `production` channels.

```
expo publish:history --platform ios
```

publishedTime	appVersion	sdkVersion	platform	channel	channelId	public
2018-01-05T23:55:04.603Z	1.0.0	24.0.0	ios	staging	9133d577	d9bd6t
2018-01-05T23:55:04.603Z	1.0.0	24.0.0	ios	production	6e406223	d9bd6t

Rollback a channel entry

Example use case: you published a release to your `production` channel, only to realize that it includes a major regression for some of your users, so you want to revert back to the previous version.

Continuing from the previous section, we rollback our `production` channel entry with `expo publish:rollback`.

```
expo publish:rollback --channel-id 6e406223
```

Now we can see that our release is no longer available in the production channel.

```
expo publish:history --platform ios
```

publishedTime	appVersion	sdkVersion	platform	channel	channelId	publicat
2018-01-05T23:55:04.603Z	1.0.0	24.0.0	ios	staging	9133d577	d9bd6b8

Release channels CLI tools

Publish history

```
Usage: expo publish:history [--release-channel <channel-name>] [--count <number-of-logs>]

View a log of your published releases.

Options:
  -c, --release-channel <channel-name> Filter by release channel. If this flag is not included, the most recent publications will be shown.
  -count, --count <number-of-logs>      Number of logs to view, maximum 100, default 5.
  -r, --raw                           Produce some raw output.
  -p, --platform <ios|android>       Filter by platform, android or ios.
```

Publish details

```
Usage: expo publish:details --publish-id <publish-id>
View the details of a published release.

Options:
  --publish-id <publish-id> Publication id. (Required)
  -r, --raw                         Produce some raw output.
```

Publish rollback

```
Usage: expo publish:rollback --channel-id <channel-id>

Rollback an update to a channel.

Options:
  --channel-id <channel-id> The channel id to rollback in the channel. (Required)
```

Publish set

```
Usage: expo publish:set --release-channel <channel-name> --publish-id <publish-id>
```

Set a published release to be served from a specified channel.

Options:

-c, --release-channel <channel-name>	The channel to set the published release. (Required)
-p, --publish-id <publish-id>	The id of the published release to serve from the channel. (Required)

Hosting An App on Your Servers

WARNING: This feature is in beta.

Normally, when over-the-air (OTA) updates are enabled, your app will fetch JS bundles and assets from Expo's CDN. However, there will be situations when you will want to host your JS bundles and assets on your own servers. For example, OTA updates are slow or unusable in countries that have blocked Expo's CDN providers on AWS and Google Cloud. In these cases, you can host your app on your own servers to better suit your use case.

For simplicity, the rest of this article will refer to hosting an app for the Android platform, but you could swap out Android for iOS at any point and everything would still be true.

Export app

First, you'll need to export all the static files of your app so they can be served from your CDN. To do this, run `expo export --public-url <server-endpoint>` in your project directory and it will output all your app's static files to a directory named `dist`. In this guide, we will use <https://expo.github.io/self-hosting-example> as our example server endpoint. Asset and bundle files are named by the md5 hash of their content. Your output directory should look something like this now:

```
.
```

```
├── android-index.json
```

```
└── ios-index.json
```

```
├── assets
```

```
│   └── 1eccbc4c41d49fd81840aef3eaabe862
```

```
└── bundles
```

```
    ├── android-01ee6e3ab3e8c16a4d926c91808d5320.js
```

```
    └── ios-ee8206cc754d3f7aa9123b7f909d94ea.js
```

Hosting your static files

Once you've exported your app's static files, you can host the contents on your own server. For example, in your `dist` output directory, an easy way to host your own files is to push the contents to Github. You can enable [Github Pages](#) to make your app available at a base URL like <https://username.github.io/project-name>. To host your files on Github, you'd do something like this:

```
# run this from your project directory
expo export --public-url https://expo.github.io/self-hosting-example

# commit output directory contents to your repo
cd dist
git init && git remote add origin git@github.com:expo/self-hosting-example.git
git add * && git commit -m "Update my app with this JS bundle"
git push origin master
```

To setup a QR code to view your hosted app, or if you want to host your files locally, follow the instructions below in the 'Loading QR Code/URL in Development' section.

Build standalone app

In order to configure your standalone binary to pull OTA updates from your server, you'll need to define the URL where you will host your `index.json` file. Pass the URL to your hosted `index.json` file to the `expo build` command.

For iOS builds, run the following commands from your terminal: `expo build:ios --public-url <path-to-ios-index.json>`, Where the `public-url` option will be something like <https://expo.github.io/self-hosting-example/ios-index.json>

For Android builds, run the following commands from your terminal: `expo build:android --public-url <path-to-android-index.json>`, where the `public-url` option will be something like <https://expo.github.io/self-hosting-example/android-index.json>

Loading QR Code/URL in Development

You can also load an app hosted on your own servers as a QR code/URL into the Expo mobile client for development purposes.

QR code:

The URI you'll use to convert to QR code will be deeplinked using the `exp` protocol. Both `exp` and `exp` deeplink into the mobile app and perform a request using HTTPS and HTTP respectively. You can create your own QR code using an online QR code generator from the input URI.

Here's an example of how you'd do this with a remote server:

URI: `exp://expo.github.io/self-hosting-example/android-index.json`

QR code: Generate the URI from a website like <https://www.qr-code-generator.com/>

Here's an example of how you'd do this from localhost:

Run `expo export` in dev mode and then start a simple HTTP server in your output directory:

```
# Find your local IP address with `ipconfig getifaddr en0`  
# export static app files  
expo export --public-url http://`ipconfig getifaddr en0`:8000 --dev  
  
# cd into your output directory  
cd dist  
  
# run a simple http server from output directory  
python -m SimpleHTTPServer 8000
```

URI: `exp://192.xxx.xxx.xxx:8000/android-index.json` (find your local IP with a command like `ipconfig getifaddr en0`)

QR code: Generate a QR code using your URI from a website like <https://www.qr-code-generator.com/>

URL

If you are loading in your app into the expo client by passing in a URL string, you will need to pass in an URL pointing to your json file.

Here is an example URL from a remote server: <https://expo.github.io/self-hosting-example/android-index.json>

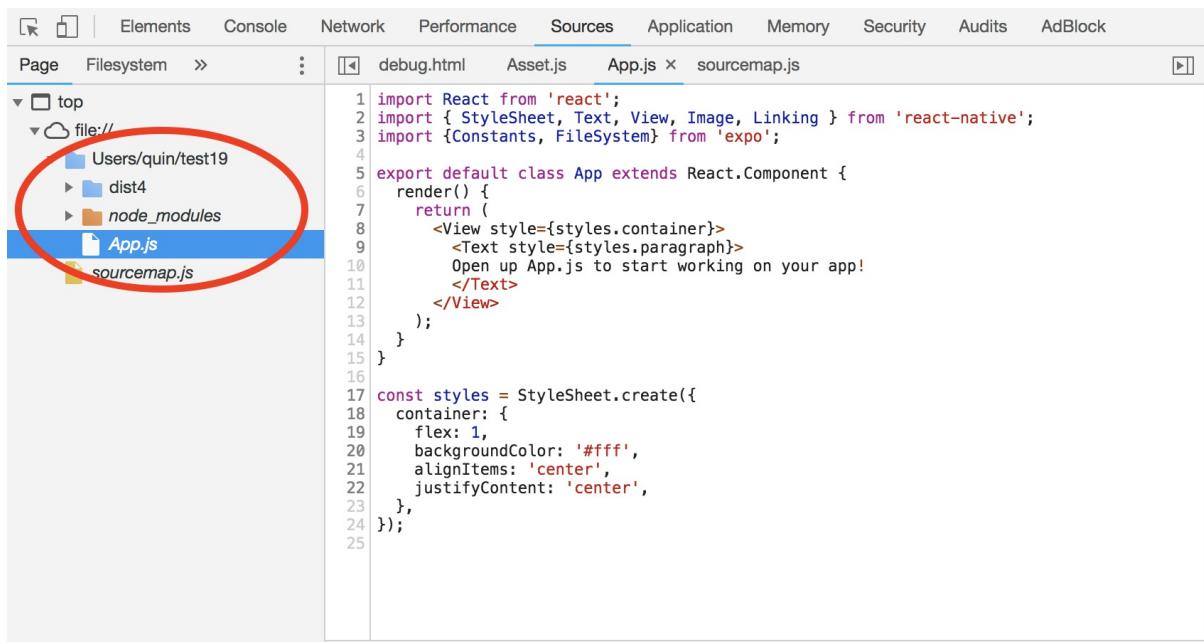
Here is an example URL from localhost: <http://localhost:8000/android-index.json>

Advanced Topics

Debugging

When we bundle your app, minification is always enabled. In order to see the original source code of your app for debugging purposes, you can generate source maps. Here is an example workflow:

1. Run `expo export --dump-sourcemap`. This will also export your bundle sourcemaps in the `bundles` directory.
2. A `debug.html` file will also be created at the root of your output directory.
3. In Chrome, open up `debug.html` and navigate to the `Source` tab. In the left tab there should be a resource explorer with a red folder containing the reconstructed source code from your bundle.



Multimanifests

As new Expo SDK versions are released, you may want to serve multiple versions of your app from your server endpoint. For example, if you first released your app with SDK 29 and later upgraded to SDK 30, you'd want users with your old standalone binary to receive the SDK 29 version, and those with the new standalone binary to receive the SDK 30 version.

In order to do this, you can run `expo export` with some merge flags to combine previously exported apps into a single multiversion app which you can serve from your servers.

Here is an example workflow:

1. Release your app with previous Expo SDKs. For example, when you released SDK 29, you can run `expo export --output-dir sdk29`. This exports the current version of the app (SDK 29) to a directory named `sdk29`.

2. Update your app and include previous Expo SDK versions. For example, if you've previously released SDK 28 and 29 versions of your app, you can include them when you release an SDK 30 version by running `expo export --merge-src-dir sdk29 --merge-src-dir sdk28`. Alternatively, you could also compress and host the directories and run `expo export --merge-src-url https://example.com/sdk29.tar.gz --merge-src-url https://example.com/sdk28.tar.gz`. This creates a multiversion app in the `dist` output directory. The `asset` and `bundle` folders contain everything that the source directories had, and the `index.json` file contains an array of the individual `index.json` files found in the source directories.

Asset Hosting

By default, all assets are hosted from an `assets` path resolving from your `public-url` (e.g. <https://expo.github.io/self-hosting-example/assets>). You can override this behavior in the `assetUrlOverride` field of your `android-index.json`. All relative URL's will be resolved from the `public-url`.

Special fields

Most of the fields in the `index.json` files are the same as in `app.json`. Here are some fields that are notable in `index.json`:

- `revisionId`, `commitTime`, `publishedTime`: These fields are generated by `expo export` and used to determine whether or not an OTA update should occur.
- `bundleUrl`: This points to the path where the app's bundles are hosted. They are also used to determine whether or not an OTA update should occur.
- `slug`: This should not be changed. Your app is namespaced by `slug`, and changing this field will result in undefined behavior in the Expo SDK components such as `Filesystem`.
- `assetUrlOverride`: The path which assets are hosted from. It is by default `./assets`, which is resolved relative to the base `public-url` value you initially passed in.

Building Standalone Apps on Your CI

NOTE: macOS is required to build standalone iOS apps.

This guide describes an advanced feature of Expo. In most cases you can build standalone Expo apps using Expo's build services as described in the guide on [Building Standalone Apps](#).

If you prefer to not rely on our builders stability and you don't like waiting in the queue to get your standalone app build then you can build your Expo project on your own. The only thing you need is Turtle CLI. Turtle CLI is a command line interface for building Expo standalone apps. You can use it both on your CI and your private computer.

Install Turtle CLI

Prerequisites

You'll need to have these things installed:

- bash
- Node.js (version 8 or newer) - [download the latest version of Node.js](#).

For Android builds

- Java Development Kit (version 8)
- gulp-cli (run `npm install -g gulp-cli` to get it)

For iOS builds

- macOS
- Xcode (version 9.4.1 or newer) - make sure you have run it at least once and you have agreed to the license agreements. Alternatively you can run `sudo xcodebuild -license`.
- fastlane - [see how to install it](#)

Turtle CLI

Install Turtle CLI by running:

```
$ npm install -g turtle-cli
```

Then run `turtle setup:ios` and/or `turtle setup:android` to verify everything is installed correctly. This step is optional and is also performed during the first run of the Turtle CLI. Please note that the Android setup command downloads, installs, and configures the appropriate versions of the Android SDK and NDK.

If you would like to make the first build even faster, you can supply the Expo SDK version to the setup command like so: `turtle setup:ios --sdk-version 30.0.0`. This tells Turtle CLI to download additional Expo-related dependencies for the given SDK version.

All Expo-related dependencies will be installed in a directory named `.turtle` within your home directory. This directory may be removed safely if you ever need to free up some disk space.

Publish your project

In order to build your standalone Expo app, you first need to have successfully published your project. See the guide on [how to publish your project](#) with Expo CLI or [how to host an app on your servers](#).

Start the build

If you choose to publish your app to Expo servers, you must have an Expo developer account and supply your credentials to the `turtle-cli`. The recommended approach is to define two environment variables called `EXPO_USERNAME` and `EXPO_PASSWORD` with your credentials, though you may also pass these values to the build command from the command line. We recommend using the environment variables to help keep your credentials out of your terminal history or CI logs.

Building for Android

Before starting the build, prepare the following things:

- Keystore
- Keystore alias
- Keystore password and key password

To learn how to generate those, see the guide on [Building Standalone Apps](#) first.

Set the `EXPO_ANDROID_KEYSTORE_PASSWORD` and `EXPO_ANDROID_KEY_PASSWORD` environment variables with the values of the keystore password and key password, respectively.

Then, start the standalone app build:

```
$ turtle build:android \
--keystore-path /path/to/your/keystore.jks \
--keystore-alias PUT_KEYSTORE_ALIAS_HERE
```

If the build finishes successfully you will find the path to the build artifact in the last line of the logs.

If you want to print the list of all available command arguments, please run `turtle build:android --help`.

Building for iOS

Prepare the following unless you're building only for the iOS simulator:

- Apple Team ID - (a 10-character string like "Q2DBWS92CA")
- Distribution Certificate .p12 file (+ password)
- Provisioning Profile

To learn how to generate those, see the guide on [Building Standalone Apps](#) first.

Set the `EXPO_IOS_DIST_P12_PASSWORD` environment variable with the value of the Distribution Certificate password.

Then, start the standalone app build:

```
$ turtle build:ios \
--team-id YOUR_TEAM_ID \
--dist-p12-path /path/to/your/dist/cert.p12 \
--provisioning-profile-path /path/to/your/provisioning/profile.mobileprovision
```

If the build finishes successfully you will find the path to the build artifact in the last line of the logs.

If you want to print the list of all available command arguments, please run `turtle build:ios --help`.

CI configuration file examples

See [expo/turtle-cli-example](#) repository for examples of how to use Turtle CLI with popular CI services (i.e. [CircleCI](#) and [Travis CI](#)).

Uploading Apps to the Apple App Store and Google Play

Disclaimer: This feature works properly only on macOS.

This guide will help you upload your Expo standalone apps to Apple TestFlight and to Google Play. You'll need a paid developer account for each platform for which you wish to upload and publish an app. You can create an Apple Developer account on [Apple's developer site](#) and a Google Play Developer account on the [Google Play Console sign-up page](#).

1. Build a standalone app

To learn how to build native binaries, see [Building Standalone Apps](#) or [Building Standalone Apps on Your CI](#).

2. Start the upload

To upload the previously built standalone app to the appropriate app store, you simply run `expo upload:android` or `expo upload:ios`. However, you have a few options for choosing which app binary you want to upload (remember to choose one at the time):

- `--latest` - chosen by default, uploads the latest build for the given platform found on the Expo servers
- `--id <id>` - uploads a build with the given ID
- `--path <path>` - uploads a build from the local file system

2.1. If you choose to upload your Android app to Google Play

Important: You have to create a Google Service Account and download its JSON private key. After that, you'll have to create an app on the [Google Play Console](#) and upload your app manually at least once.

Creating a Google Service Account

1. Open the [Google Play Console](#).
2. Click the **Settings** menu entry, followed by **API access**.
3. Click the **CREATE SERVICE ACCOUNT** button. If you see a message saying API access is not enabled for your account, you must first link your Google Play developer account with a Google Developer Project. On this page, either link it to an existing project if you have one, or click **CREATE NEW PROJECT** to link with a new one.
4. Follow the **Google API Console** link in the dialog
 - i. Click the **CREATE SERVICE ACCOUNT** button
 - ii. Enter the name of this service account in the field titled "Service account name". We recommend a name that will make it easy for you to remember that it is for your Google Play Console account. Also, enter the service account ID and description of your choice.
 - iii. Click **Select a role** and choose **Service Accounts > ServiceAccount User**
 - iv. Check the **Furnish a new private key** checkbox

- v. Make sure the "Key type" field is set to **JSON**
 - vi. Click **SAVE** to close the dialog
 - vii. Make a note of the filename of the JSON file downloaded to your computer. You'll need this to upload your app later. Be sure to keep this JSON file secure, as it provides API access to your Google Play developer account.
5. Return to the **API access** page on the **Google Play Console** and ensure it shows your new service account.
 6. Click on **Grant Access** for the newly added service account
 7. Choose **Release Manager** from the newly added service account
 8. Click **ADD USER** to close the dialog

Manually uploading your app for the first time

Before using `expo-cli` for uploading your standalone app builds, you have to upload your app manually at least once. [See here for the instructions on how to do it.](#)

Using expo-cli to upload the further builds of your app

After these steps, you can make use of `expo-cli` to upload your further app builds to Google Play.

To upload your Android app to Google Play, run `expo upload:android`. You can set following options when uploading an Android standalone app:

- `--key <key>` (**required**) - path to the JSON key used to authenticate with the Google Play Store (created in the previous steps)
- `--track <track>` - the track of the application to use, choose from: production, beta, alpha, internal, rollout (default: internal)

2.2. If you choose to upload your iOS app to TestFlight

Using expo-cli

To upload your iOS app to TestFlight, run `expo upload:ios`. You can set following options when uploading an iOS standalone app:

- `--apple-id <apple-id>` (**required**) - your Apple ID login. Alternatively you can set the `EXPO_APPLE_ID` environment variable.
- `--apple-id-password <apple-id-password>` (**required**) - your Apple ID password. Alternatively you can set the `EXPO_APPLE_ID_PASSWORD` environment variable.
- `--app-name <app-name>` - your app display name, will be used to name an app on App Store Connect
- `--sku <sku>` - a unique ID for your app that is not visible on the App Store, will be generated unless provided
- `--language <language>` - primary language (e.g. English, German; run `expo upload:ios --help` to see the list of available languages) (default: English)

Manually uploading your app

In order to see your app on Testflight, you will first need to submit your .IPA file to Apple using Application loader. In order to do this, there are a few prerequisite steps which you may not have followed previously if this is your first app submission to Apple:

1. Make sure you have logged into iTunes connect at least once with your Apple ID and accepted the terms.

2. Login to <https://appleid.apple.com>
3. Generate an app specific password by going to Accounts > Manage > Generate App Specific Password.
Make a note of this password as it will be needed later.
4. Start XCode but do not load any project
5. From the XCode menu in the menu bar, select 'Open Developer Tool' and then 'Application Loader'
6. Once Application Loader launches, login with your Apple ID and the app specific password generated in step 3
7. Follow the steps to agree to the necessary terms.
8. Once you have agreed to all terms, double-click on the light-grey panel in the center (above the words 'Deliver Your App').
9. Follow the steps to upload your IPA to Apple.

You can check the status of your app submission to TestFlight in App Store Connect

(<http://appstoreconnect.apple.com>):

1. Login to <http://appstoreconnect.apple.com> with your Apple ID and regular password (NOT your app specific password)
2. Select 'My Apps' and you should see your app listed.
3. Click 'TestFlight' from the menu bar at the top.
4. This will show your current app builds that are available for testing.
5. In order to test the app on your device, you will need to install the TestFlight iOS app from the App Store, and sign in using your Apple ID.

App Transfers

When you upload an iOS app to App Store Connect and distribute it through the App Store, it becomes (semi-)permanently associated with your Apple Developer account. This means that any future updates to your app must go through your Apple account. Apple provides a process called [App Transfer](#) as a way around this, for cases in which you want to transfer the ownership and maintainability of one of your apps to a different Apple Developer account.

One of the [criteria for an app to be eligible to transfer](#) is that "No version of the app can use a Passbook entitlement." . Until December 11, 2018, Expo apps built with SDK 30 or 31 included the `Payments` module, which necessitates linking a file called `PassKit.framework` . This file includes the native APIs for interacting with the native iOS payments functionality. All SDK 30 and 31 apps built before this date included this file -- this is because we always include the native code for all modules in every build, even modules that you don't import in your JS. (Among other reasons, this makes it possible for you to use different modules in OTA updates without having to build a new binary.)

While no apps built with Expo have ever had any Passbook entitlements enabled, we've received a number of reports indicating that simply including `PassKit.framework` in a project is enough to make an app permanently ineligible to transfer. We removed the `Payments` module from all iOS builds beginning December 12, 2018. However, if you built an SDK 30 or 31 iOS app with `expo build:ios` before this date, and uploaded the resulting .ipa file to App Store Connect, your app will likely be ineligible to transfer using the App Transfer process. If you attempt to transfer your app but receive a rejection due to "use of the PassKit Integration API", then your app is affected by this issue.

For affected apps, the only way to transfer an app to a different account is to resubmit the app under the new Apple Developer account with a new `bundleIdentifier` (binary reassignment). The transferred app will effectively be a new and separate app on the App Store and on users' devices. We suggest that once this process is completed, you submit a final update to the old app which notifies users about the new app listing and directs them to download it.

If you never submitted an SDK 30 or 31 .ipa file to App Store Connect, or if you only did so with .ipa files built on or after December 12, 2018, your app should be eligible for App Transfer.

Overview

ExpoKit is an Objective-C and Java library that allows you to use the Expo platform and your existing Expo project as part of a larger standard native project -- one that you would normally create using Xcode, Android Studio, or `react-native init`.

Because Expo already provides the ability to [render native binaries for the App Store](#), most Expo developers do not need to use ExpoKit. In some cases, projects need to make use of third-party Objective-C or Java native code that is not included in the core Expo SDK. ExpoKit provides this ability.

This documentation will discuss:

- [Ejecting](#) your normal (JS) Expo project into a JS-and-native ExpoKit project
- [Changing your workflow](#) to use custom native code with ExpoKit
- Other [advanced ExpoKit topics](#)

Ejecting to ExpoKit

ExpoKit is an Objective-C and Java library that allows you to use the Expo platform and your existing Expo project as part of a larger standard native project -- one that you would normally create using Xcode, Android Studio, or `react-native init`.

What is this for?

If you created an Expo project and you want a way to add custom native modules, this guide will explain how to use ExpoKit for that purpose.

Normally, Expo apps are written in pure JS and never "drop down" to the native iOS or Android layer. This is core to the Expo philosophy and it's part of what makes Expo fast and powerful to use.

However, there are some cases where advanced developers need native capabilities outside of what Expo offers out-of-the-box. The most common situation is when a project requires a specific Native Module which is not supported by React Native Core or the Expo SDK.

In this case, Expo allows you to *eject* your pure-JS project from the Expo iOS/Android clients, providing you with native projects that can be opened and built with Xcode and Android Studio. Those projects will have dependencies on ExpoKit, so everything you already built will keep working as it did before.

We call this "ejecting" because you still depend on the Expo SDK, but your project no longer lives inside the standard Expo client. You control the native projects, including configuring and building them yourself.

Should I eject to ExpoKit?

You might want to eject if:

- Your Expo project needs a native module that Expo doesn't currently support. We're always expanding the [Expo SDK](#), so we hope this is never the case. But it happens, especially if your app has very specific and uncommon native demands.

You should not eject if:

- All you need is to distribute your app in the iTunes Store or Google Play. Expo can [build binaries for you](#) in that case. If you eject, we can't automatically build for you any more.
- You are uncomfortable writing native code. Ejected apps will require you to manage Xcode and Android Studio projects.
- You enjoy the painless React Native upgrades that come with Expo. After your app is ejected, breaking changes in React Native will affect your project differently, and you may need to figure them out for your particular situation.
- You require Expo's push notification services. After ejecting, since Expo no longer manages your push certificates, you'll need to manage your own push notification pipeline.
- You rely on asking for help in the Expo community. In your native Xcode and Android Studio projects, you may encounter questions which are no longer within the realm of Expo.

Instructions

The following steps are for converting a pure-JS Expo project (such as one created with Expo CLI) into a native iOS and Android project which depends on ExpoKit.

After you eject, all your JS files will stay the same, but we'll additionally create `ios` and `android` directories in your project folder. These will contain Xcode and Android Studio projects respectively, and they'll have dependencies on React Native and on Expo's core SDK.

You'll still be able to develop and test your project with Expo CLI, and you'll still be able to publish your Expo JS code the same way. However, if you add native dependencies that aren't included in Expo, other users won't be able to run those features of your app with the main Expo app. You'll need to distribute the native project yourself.

1. Install Expo CLI

If you don't have it, run `npm install -g expo-cli` to get our command line library.

If you haven't used Expo CLI with an Expo account before, the eject command will ask you to create one.

2. Make sure you have the necessary configuration options in `app.json`

Ejecting requires the same configuration options as building a standalone app. [Follow these instructions before continuing to the next step.](#)

3. Eject

From your project directory, run `expo eject`. This will download the required dependencies and build native projects under the `ios` and `android` directories.

4. Set up and Run your native project

Congrats, you now have a native project with ExpoKit! Follow the directions under [Developing with ExpoKit](#) to get things set up and running.

5. Make native changes

You can do whatever you want in the Xcode and Android Studio projects.

To add third-party native modules for React Native, non-Expo-specific instructions such as `react-native link` should be supported. [Read more details about changing native dependencies in your ExpoKit project.](#)

6. Distribute your app

Publishing your JS from Expo CLI will still work. Users of your app will get the new JS on their devices as soon as they reload their app; you don't need to rebuild your native code if it has not changed.

If you do make native changes, people who don't have your native code may encounter crashes if they try to use features that depend on those changes.

If you decide to distribute your app as an `ipa` or `apk`, it will automatically hit your app's published URL instead of your development Expo CLI URL. Read [advanced details about your app's JS url](#).

In general, before taking your app all the way to production, it's a good idea to glance over the [Advanced ExpoKit Topics](#) guide.

Developing With ExpoKit

ExpoKit is an Objective-C and Java library that allows you to use the Expo platform with a native iOS/Android project.

Before you read this guide

To create an ExpoKit project:

1. Create a pure-JS project with Expo CLI (also projects that were created with `exp`, XDE or `create-react-native-app` will work)
2. Then use `expo eject` to add ExpoKit (choose the "ExpoKit" option).

Make sure to perform these steps before continuing in this guide. The remainder of the guide will assume you have created an ExpoKit project.

Setting up your project

By this point you should have a JS app which additionally contains `ios` and `android` directories.

1. Check JS dependencies

- Your project's `package.json` should contain a `react-native` dependency pointing at Expo's fork of React Native. This should already be configured for you.
- Your JS dependencies should already be installed (via `npm install` OR `yarn`).

2. Run the project with Expo CLI

Run `expo start` from the project directory.

This step ensures that the React Native packager is running and serving your app's JS bundle for development. Leave this running and continue with the following steps.

3. iOS: Configure, build and run

This step ensures the native iOS project is correctly configured and ready for development.

- Make sure you have the latest Xcode.
- If you don't have it already, install [CocoaPods](#), which is a native dependency manager for iOS.
- Run `pod install` from your project's `ios` directory.
- Open your project's `xcworkspace` file in Xcode.
- Use Xcode to build, install and run the project on your test device or simulator. (this will happen by default if you click the big "Play" button in Xcode.)

Once it's running, the iOS app should automatically request your JS bundle from the project you're serving from Expo CLI.

4. Android: Build and run

Open the `android` directory in Android Studio, then build and run the project on an Android device or emulator.

When opening the project, Android Studio may prompt you to upgrade the version of Gradle or other build tools, but don't do this as you may get unexpected results. ExpoKit always ships with the latest supported versions of all build tools.

If you prefer to use the command line, you can run `./gradlew installDevKernelDebug` from inside the `android` directory to build the project and install it on the running device/emulator.

Once the Android project is running, it should automatically request your development url from Expo CLI. You can develop your project normally from here.

Continuing with development

Every time you want to develop, ensure your project's JS is being served by Expo CLI (step 2), then run the native code from Xcode or Android Studio respectively.

Your ExpoKit project is configured to load your app's published url when you build it for release. So when you want to release it, don't forget to publish, like with any normal (non-ExpoKit) project.

Changing Native Dependencies

iOS

Your ExpoKit project manages its dependencies with [CocoaPods](#).

Many libraries in the React Native ecosystem include instructions to run `react-native link`. These are supported with ExpoKit for iOS.

- If the library supports CocoaPods (has a `.podspec` file), just follow the normal instructions and run `react-native link`.
- If the library doesn't support CocoaPods, `react-native link` may fail to include the library's header files. If you encounter build issues locating the `<React/*>` headers, you may need to manually add `Pods/Headers/Public` to the **Header Search Paths** configuration for your native dependency in Xcode. If you're not familiar with Xcode, search Xcode help for "configure build settings" to get an idea of how those work. **Header Search Paths** is one such build setting. The target you care to configure is the one created by `react-native link` inside your Xcode project. You'll want to determine the relative path from your library to `Pods/Headers/Public`.

Android

Many libraries in the React Native ecosystem include instructions to run `react-native link`. These are supported with ExpoKit for Android.

Upgrading ExpoKit

ExpoKit's release cycle follows the Expo SDK release cycle. When a new version of the Expo SDK comes out, the release notes include upgrade instructions for the normal, JS-only part of your project. Additionally, you'll need to update the native ExpoKit code.

Note: Please make sure you've already updated your JS dependencies before proceeding with the following instructions. Additionally, there may be version-specific breaking changes not covered here.

iOS

- Open up `ios/Podfile` in your project, and update the `ExpoKit` tag to point at the `release` corresponding to your SDK version. Run `pod update` then `pod install`.
- Open `ios/your-project/Supporting/EXSDKVersions.plist` in your project and change all the values to the new SDK version.

If upgrading from SDK 31 or below, you'll need to refactor your `AppDelegate` class as we moved its Expo-related part to a separate `EXStandaloneAppDelegate` class owned by `ExpoKit` to simplify future upgrade processes as much as possible. As of SDK 32, your `AppDelegate` class needs to subclass `EXStandaloneAppDelegate`.

If you have never made any edits to your Expo-generated `AppDelegate` files, then you can just replace them with these new template files:

- `AppDelegate.h`
- `AppDelegate.m`

If you override any `AppDelegate` methods to add custom behavior, you'll need to either refactor your `AppDelegate` to subclass `EXStandaloneAppDelegate` and call `super` methods when necessary, or start with the new template files above and add your custom logic again (be sure to keep the calls to `super` methods).

If upgrading from SDK 30 or below, you'll also need to change `platform :ios, '9.0'` to `platform :ios, '10.0'` in `ios/Podfile`.

Android

- Go to <https://expo.io/--/api/v2/versions> and find the `expokitNpmPackage` key under `sdkVersions.[NEW SDK VERSION]`.
- Update your version of expokit in `package.json` to the version in `expokitNpmPackage` and `yarn/npm install`.
- If upgrading to SDK 31 or below, go to `MainActivity.java` and replace `Arrays.asList("[OLD SDK VERSION]")` with `Arrays.asList("[NEW SDK VERSION]")`. If upgrading to SDK 32 or above, simply remove the entire `public List<String> sdkVersions()` method from `MainActivity.java`.
- Go to `android/app/build.gradle` and replace `compile('host.exp.exponent:expoview:[OLD SDK VERSION]@aar')` { with `compile('host.exp.exponent:expoview:[NEW SDK VERSION]@aar')` { .

If upgrading from SDK31 or below:

1. add the following lines to `android/app/build.gradle`:

```
api 'host.exp.exponent:expo-app-loader-provider:1.0.0'
api 'host.exp.exponent:expo-core:2.0.0'
api 'host.exp.exponent:expo-constants-interface:2.0.0'
api 'host.exp.exponent:expo-constants:2.0.0'
api 'host.exp.exponent:expo-errors:1.0.0'
api 'host.exp.exponent:expo-file-system-interface:2.0.0'
api 'host.exp.exponent:expo-file-system:2.0.0'
api 'host.exp.exponent:expo-image-loader-interface:2.0.0'
api 'host.exp.exponent:expo-permissions:2.0.0'
api 'host.exp.exponent:expo-permissions-interface:2.0.0'
api 'host.exp.exponent:expo-sensors-interface:2.0.0'
api 'host.exp.exponent:expo-react-native-adapter:2.0.0'
api 'host.exp.exponent:expo-task-manager:1.0.0'
api 'host.exp.exponent:expo-task-manager-interface:1.0.0'
```

```

// Optional universal modules, could be removed
// along with references in MainActivity
api 'host.exp.exponent:expo-ads-admob:2.0.0'
api 'host.exp.exponent:expo-app-auth:2.0.0'
api 'host.exp.exponent:expo-analytics-segment:2.0.0'
api 'host.exp.exponent:expo-barcode-scanner-interface:2.0.0'
api 'host.exp.exponent:expo-barcode-scanner:2.0.0'
api 'host.exp.exponent:expo-camera-interface:2.0.0'
api 'host.exp.exponent:expo-camera:2.0.0'
api 'host.exp.exponent:expo-contacts:2.0.0'
api 'host.exp.exponent:expo-face-detector:2.0.0'
api 'host.exp.exponent:expo-face-detector-interface:2.0.0'
api 'host.exp.exponent:expo-font:2.0.0'
api 'host.exp.exponent:expo-gl-cpp:2.0.0'
api 'host.exp.exponent:expo-gl:2.0.0'
api 'host.exp.exponent:expo-google-sign-in:2.0.0'
api 'host.exp.exponent:expo-local-authentication:2.0.0'
api 'host.exp.exponent:expo-localization:2.0.0'
api 'host.exp.exponent:expo-location:2.0.0'
api 'host.exp.exponent:expo-media-library:2.0.0'
api 'host.exp.exponent:expo-print:2.0.0'
api 'host.exp.exponent:expo-sensors:2.0.0'
api 'host.exp.exponent:expo-sms:2.0.0'
api 'host.exp.exponent:expo-background-fetch:1.0.0'

```

2. Ensure that in `MainActivity.java` , `expoPackages` method looks like this:

```

@Override
public List<Package> expoPackages() {
    return ((MainApplication) getApplication()).getExpoPackages();
}

```

3. In `MainApplication.java` , replace

```
public class MainApplication extends ExpoApplication {
```

with

```
public class MainApplication extends ExpoApplication implements AppLoaderPackagesProviderInterface<ReactPackage> {
```

4. Add the following lines in `MainApplication.java` :

```

import expo.core.interfaces.Package;
import expo.loaders.provider.interfaces.AppLoaderPackagesProviderInterface;
import expo.modules.ads.admob.AdMobPackage;
import expo.modules.analytics.segment.SegmentPackage;
import expo.modules.appauth.AppAuthPackage;
import expo.modules.backgroundfetch.BackgroundFetchPackage;
import expo.modules.barcodescanner.BarCodeScannerPackage;
import expo.modules.camera.CameraPackage;
import expo.modules.constants.ConstantsPackage;
import expo.modules.contacts.ContactsPackage;
import expo.modules.facedetector.FaceDetectorPackage;
import expo.modules.filesystem.FileSystemPackage;
import expo.modules.font.FontLoaderPackage;
import expo.modules.gl.GLPackage;
import expo.modules.google.signin.GoogleSignInPackage;
import expo.modules.localauthentication.LocalAuthenticationPackage;
import expo.modules.localization.LocalizationPackage;
import expo.modules.location.LocationPackage;

```

```

import expo.modules.medialibrary.MediaLibraryPackage;
import expo.modules.permissions.PermissionsPackage;
import expo.modules.print.PrintPackage;
import expo.modules.sensors.SensorsPackage;
import expo.modules.sms.SMSPackage;
import expo.modules.taskManager.TaskManagerPackage;

...

public List<Package> getExpoPackages() {
    return Arrays.<Package>asList(
        new CameraPackage(),
        new ConstantsPackage(),
        new SensorsPackage(),
        new FileSystemPackage(),
        new FaceDetectorPackage(),
        new GLPackage(),
        new GoogleSignInPackage(),
        new PermissionsPackage(),
        new SMSPackage(),
        new PrintPackage(),
        new ConstantsPackage(),
        new MediaLibraryPackage(),
        new SegmentPackage(),
        new FontLoaderPackage(),
        new LocationPackage(),
        new ContactsPackage(),
        new BarCodeScannerPackage(),
        new AdMobPackage(),
        new LocalAuthenticationPackage(),
        new LocalizationPackage(),
        new AppAuthPackage(),
        new TaskManagerPackage(),
        new BackgroundFetchPackage()
    );
}

```

If upgrading from SDK 30 or below, remove the following lines from `android/app/build.gradle`:

```

implementation 'com.squareup.okhttp3:okhttp:3.4.1'
implementation 'com.squareup.okhttp3:okhttp-urlconnection:3.4.1'
implementation 'com.squareup.okhttp3:okhttp-ws:3.4.1'

```

If upgrading from SDK 28 or below, you'll also need to follow these instructions:

- Change all instances of `android\detach-scripts` and `android/detach-scripts` to `node_modules\expokit\detach-scripts` and `node_modules/expokit/detach-scripts` respectively in `android/app/expo.gradle`.
- Add `maven { url "$rootDir/../node_modules/expokit/maven" }` under `allprojects.repositories` in `android/build.gradle`.
- In `android/app/build.gradle`, replace

```

compile('host.exp.exponent:expoview:[SDK VERSION]@aar') {
    transitive = true
}

```

with

```

compile('host.exp.exponent:expoview:[SDK VERSION]@aar') {
    transitive = true
    exclude group: 'com.squareup.okhttp3', module: 'okhttp'
    exclude group: 'com.squareup.okhttp3', module: 'okhttp-urlconnection'

```

}

Advanced ExpoKit Topics

This guide goes deeper into a few [ExpoKit](#) topics that aren't critical right out of the box, but that you may encounter down the road. If you're not familiar with ExpoKit, you might want to read [the ExpoKit guide](#) first.

Un-ejecting

It is possible to manually "un-eject" your project, for example if you want to return to a JS-only state, or if you want to repeatedly eject for testing purposes. Since your project won't be ejected any more, you will no longer be able to use custom native code.

Warning: The following instructions will permanently remove the native iOS and Android code from your project, including any changes you've made. We strongly recommend committing your changes to version control before trying this.

To un-eject:

- Delete the `ios` and `android` directories from your project.
- Delete the `isDetached` and `detach` keys from your project's `app.json`.

You can now use your project like a normal Expo project (with no ExpoKit).

Verifying Bundles (iOS only)

When we serve your JS over-the-air to your ExpoKit project, we include a signature so that your project can verify that the JS actually came from our servers.

By default, projects that use ExpoKit have this feature disabled on iOS and enabled on Android. We encourage you to enable it on iOS so that your code is verified for all of your users.

To enable code verification in your native project with ExpoKit:

- Fulfill one of these two requirements (you only need one):
 - Use a non-wildcard bundle identifier when provisioning the app (recommended)
 - Enable **Keychain Sharing** in your Xcode project settings under **Capabilities**. (faster to set up)
- In `ios/your-project/Supporting/EXShell.plist`, set `isManifestVerificationBypassed` to `NO` (or delete this key entirely).

Configuring the JS URL

In development, your ExpoKit project will request your local build from Expo CLI. You can see this configuration in `EXBuildConstants.plist` (iOS) or `ExponentBuildConstants` (Android). You shouldn't need to edit it, because it's written automatically when you serve the project.

In production, your ExpoKit project will request your published JS bundle. This is configured in `EXShell.plist` (iOS) and `MainActivity.java` (Android). If you want to specify custom behavior in iOS, you can also set the `[ExpoKit sharedInstance].publishedManifestUrlOverride` property.

Changing the Deep Link Scheme

If you do not have a `scheme` specified in `app.json` at the time of ejecting, Expo will automatically generate a random one for you. If you'd like to switch to a different scheme after ejecting, there are a few places where you need to find an occurrence of your old scheme and replace it with the new one:

1. `app.json` (the `"scheme"` field)
2. `ios/<your-project-name>/Supporting/Info.plist` (under the first occurrence of `CFBundleURLSchemes`)
3. `android/app/src/main/AndroidManifest.xml` (in a line that looks like `<data android:scheme="<your-scheme-here>"/>`, under `MainActivity`, OR `LauncherActivity` for older projects)
4. `android/app/src/main/java/host/exp/exponent/generated/AppConstants.java` (the `SHELL_APP_SCHEME` variable)

Enabling Optional Expo Modules on iOS

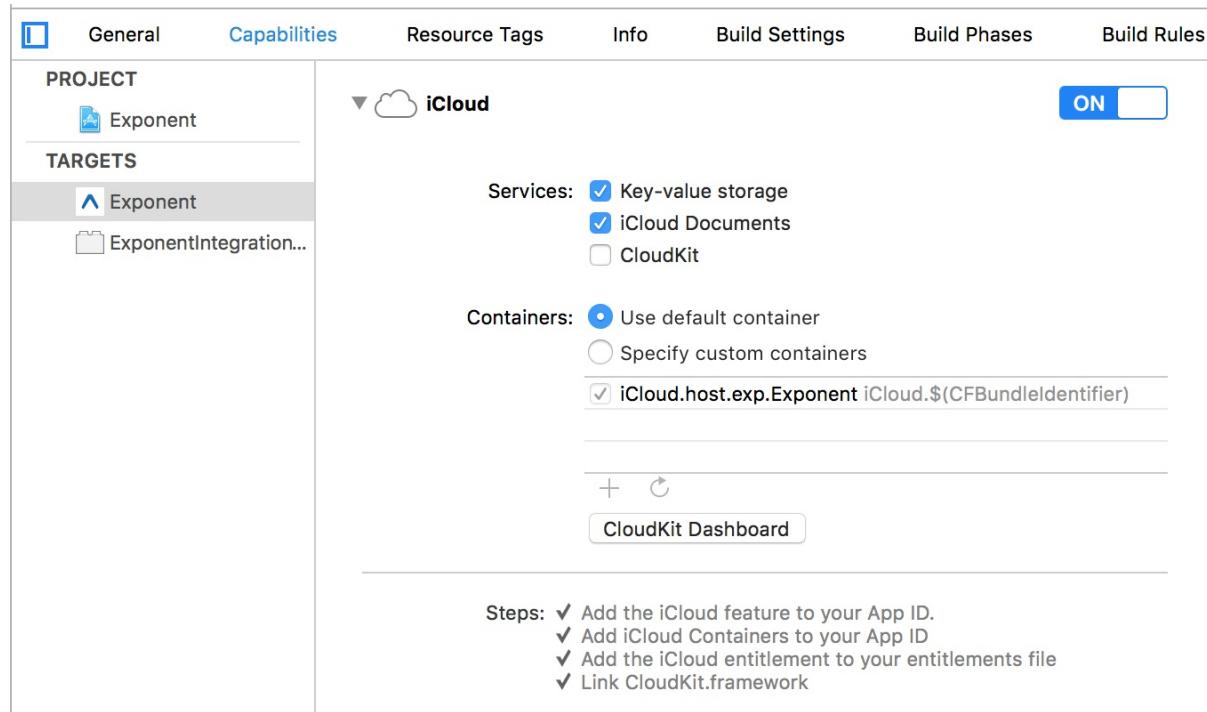
To enable FaceDetector, ARKit, or Payments in your iOS app, see [Universal Modules and ExpoKit](#).

Using DocumentPicker

In iOS Expokit projects, the DocumentPicker module requires the iCloud entitlement to work properly. If your app doesn't have it already, you can add it by opening the project in Xcode and following these steps:

- In the project go to the `Capabilities` tab.
- Set the iCloud switch to on.
- Check the `iCloud Documents` checkbox.

If everything worked properly your screen should look like this:



Using Google Maps

If you integrate Google Maps to your ExpoKit app with the MapView component, you may need to follow additional instructions to provide your Google Maps API key. See the [MapView docs](#).

Universal Modules and ExpoKit

Universal Modules are pieces of the Expo SDK with some special properties:

- They are optional; you can remove them from your ExpoKit build if you don't need their native code.
- They can run as standalone libraries without Expo.

Not all Expo SDK modules are Universal Modules. Right now, only a small part of our SDK has this property. We're continually expanding the number of our APIs that are available as universal modules.

Omitting Unneeded Modules

When you [create an ExpoKit project](#), we automatically add most of the same native APIs that are available in the Expo Client app. Each of these APIs is supported by some native code which increases the size of your native binary.

You can remove any Expo Universal Module from your ExpoKit project if you don't think you need it. This means it will no longer be available in your native binary; if you write some JS which tries to import this API, you might cause a fatal error in your app. If you send an [OTA update](#) to your app which contains API calls that aren't present in your native binary, you might cause a fatal error.

Omitting Universal Modules is currently supported on iOS but not Android.

If you aren't sure what this guide is for or whether you need this, you are probably better off just leaving it alone. Otherwise you risk causing crashes in your app by ripping out needed APIs.

iOS

To omit a Universal Module from your iOS ExpoKit project, remove the respective dependency from `ios/Podfile`. Then re-run `pod install` and rebuild your native code.

These modules are included by default, but can be omitted

- GL (`EXGL` and `EXGL-CPP`)
- SMS composer (`EXSMS`)
- Accelerometer, DeviceMotion, Gyroscope, Magnetometer, Pedometer (`EXSensors`)

These modules are included by default, but can be dangerously omitted

Some modules implement core Expo functionality through a generic interface. For example, our `Permissions` module implements `expo-permissions-interface`. If you remove the `Permissions` module, the project will build, but it may not run unless you add some other code which provides Expo `Permissions` functionality.

- Camera (`EXCamera`)
- Constants (`EXConstants`)
- FileSystem (`EXFileSystem`)
- Permissions (`EXPermissions`)

Android

Omitting Universal Modules is not currently supported on Android.

Adding Optional Modules on iOS

A few Expo modules are not included by default in ExpoKit iOS projects, nor in Standalone iOS Apps produced by `expo build`. Typically this is either because they add a disproportionate amount of bloat to the binary, or because they include APIs that are governed by extra Apple review guidelines. Right now those modules are:

- `FaceDetector` (`EXFaceDetector`)
- `ARKit`
- `Payments`

If you want to use any of these modules in your Expo iOS app, you need to eject to ExpoKit rather than using `expo build`. (It's on our roadmap to improve this.)

To add `FaceDetector`:

1. Add `expo-face-detector` to `package.json` and install JS dependencies.
2. Add `pod 'EXFaceDetector', path: '../node_modules/expo-face-detector/ios'` to your `Podfile`.
3. Re-run `pod install`.

To add `Payments` or `AR`, add the [respective subspec](#) to your `ExpoKit` dependency in your `Podfile`, and re-run `pod install`.

Hello World

To get started with a bare React Native project, run `expo init` and choose one of the bare templates. We'll use the minimum template here. This guide assumes that you have Xcode and/or Android Studio installed and working.

```
# If you don't have expo-cli yet, get it
npm i -g expo-cli
# If you don't have react-native-cli yet, get it
npm i -g react-native-cli
# This is a shortcut to skip the UI for picking the template
expo init --template bare-minimum
```

Next, let's get the project running. Go into your project directory and run `react-native run-ios` OR `react-native run-android` — hurray! Your project is working.

Using `@unimodules/core`

Bare template projects come with `@unimodules/core` installed and configured. This package gives you access to some commonly useful APIs, like `Asset`, `Constants`, `FileSystem`, and `Permissions`. You can import these from `@unimodules/core` like so:

```
import { Asset, Constants, FileSystem, Permissions } from '@unimodules/core';
```

Install a Unimodule

We're going to install `expo-web-browser`, it's a useful little package for showing a modal web browser using the appropriate native APIs on each platform.

```
npm install expo-web-browser
```

Open up `App.js` and add a button that, when pressed, opens up a web browser. Here's some code for you.

```
import * as React from 'react';
import { Button, View } from 'react-native';
import * as WebBrowser from 'expo-web-browser';

export default class App extends React.Component {
  render() {
    return (
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
        <Button
          title="Open a web browser"
          onPress={() => {
            WebBrowser.openBrowserAsync('https://expo.io');
          }}
        />
      </View>
    );
  }
}
```

This will not yet work because we haven't linked the native code that powers it. To do this, we need to follow the instructions in the [expo-web-browser README](#) to configure it for iOS and Android. Let's do it.

iOS configuration

The iOS side is easiest, so let's do it first. Bare projects are initialized using [Cocoapods](#), a dependency manager for iOS projects. If you don't have Cocoapods installed already, [install it](#). Now let's go into `ios/Podfile` and add `pod 'EXWebBrowser', path: '../node_modules/expo-web-browser/ios'` on a new line after `use_unimodules!`. Close the file and run `pod install` in the `ios` directory. Now you can run `react-native run-ios` again from the root of the project and it should work as expected!

Android configuration

There's not much context needed beyond what the README says here, just follow the steps below:

1. Append the following lines to `android/settings.gradle`:

```
include ':expo-web-browser'
project(':expo-web-browser').projectDir = new File(rootProject.projectDir, '../node_modules/expo-web-browser/android')
)
```

1. Insert the following lines inside the dependencies block in `android/app/build.gradle`:

```
api project(':expo-web-browser')
```

2. In `MainApplication.java`, import the package and add it to the `ReactModuleRegistryProvider` list:

```
import expo.modules.expo.modules.webbrowser.WebBrowserPackage;

private final ReactModuleRegistryProvider mModuleRegistryProvider = new ReactModuleRegistryProvider(Arrays.<Pack
age>asList(
// Your other packages will be here
new WebBrowserPackage(),
Arrays.<SingletonModule>asList()));
```

Now go ahead and run `react-native run-android`. Press the button, watch the browser open. Success! Happy times.

What now?

Most of the Expo APIs are available in bare React Native projects and can be installed using a process very similar to the above. Go ahead and browse the [API Reference](#) section and follow the installation instructions linked to there, then read the API documentation and enjoy. Good luck building your app!

Overview

The Expo SDK provides access to system functionality such as contacts, camera, and social login. It is provided by the npm package `expo`. Install it by running `npm install --save expo` in the root directory of the project. Then you can import modules from it in your JavaScript code as follows:

```
import { Contacts } from 'expo';
```

You can also import all Expo SDK modules:

```
import * as Expo from 'expo';
```

This allows you to write `Expo.Contacts.getContactsAsync()`, for example.

SDK Version

Each month there is a new Expo SDK release that typically updates to the latest version of React Native and includes a variety of bugfixes, features and improvements to the Expo APIs. It's often useful to know what version of React Native your Expo project is running on, so the following table maps Expo SDK versions to their included React Native version.

Expo SDK Version	React Native Version
32.0.0	0.57.1
31.0.0	0.57.1
30.0.0	0.55.4
29.0.0	0.55.4
28.0.0	0.55.4
27.0.0	0.55.2
26.0.0	0.54.2
25.0.0	0.52.0
24.0.0	0.51.0
23.0.0	0.50.0
22.0.0	0.49.4
21.0.0	0.48.4
20.0.0	0.47.1
19.0.0	0.46.1

Overview

The Expo SDK provides access to system functionality such as contacts, camera, and social login. It is provided by the npm package `expo`. Install it by running `npm install --save expo` in the root directory of the project. Then you can import modules from it in your JavaScript code as follows:

```
import { Contacts } from 'expo';
```

You can also import all Expo SDK modules:

```
import * as Expo from 'expo';
```

This allows you to write `Expo.Contacts.getContactsAsync()`, for example.

SDK Version

Each month there is a new Expo SDK release that typically updates to the latest version of React Native and includes a variety of bugfixes, features and improvements to the Expo APIs. It's often useful to know what version of React Native your Expo project is running on, so the following table maps Expo SDK versions to their included React Native version.

Expo SDK Version	React Native Version
32.0.0	0.57.1
31.0.0	0.57.1
30.0.0	0.55.4
29.0.0	0.55.4
28.0.0	0.55.4
27.0.0	0.55.2
26.0.0	0.54.2
25.0.0	0.52.0
24.0.0	0.51.0
23.0.0	0.50.0
22.0.0	0.49.4
21.0.0	0.48.4
20.0.0	0.47.1
19.0.0	0.46.1

ImageManipulator

An API to modify images stored on the local file system.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { ImageManipulator } from 'expo';  
  
// in bare apps:  
import * as ImageManipulator from 'expo-image-manipulator';
```

ImageManipulator.manipulateAsync(uri, actions, saveOptions)

Manipulate the image provided via `uri`. Available modifications are `rotating`, `flipping` (mirroring), `resizing` and `cropping`. Each invocation results in a new file. With one invocation you can provide a set of actions to perform over the image. Overwriting the source file would not have an effect in displaying the result as images are cached.

Arguments

- **uri (string)** -- URI of the file to manipulate. Should be in the app's scope.
- **actions (array)** --

An array of objects representing manipulation options. Each object should have *only one* of the following keys that corresponds to specific transformation:

- **resize (object)** -- An object of shape `{ width, height }`. Values correspond to the result image dimensions. If you specify only one value, the other will be calculated automatically to preserve image ratio.
- **rotate (number)** -- Degrees to rotate the image. Rotation is clockwise when the value is positive and counter-clockwise when negative.
- **flip (object)** -- An object of shape `{ vertical, horizontal }`. Having a field set to true, flips the image in specified axis. Only one flip per transformation is valid. If you want to flip according to both axes then provide two separate transformations.
- **crop (object)** -- An object of shape `{ originX, originY, width, height }`. Fields specify top-left corner and dimensions of a crop rectangle.
- **saveOptions (object)** -- A map defining how modified image should be saved:
 - **compress (number)** -- A value in range `0.0 - 1.0` specifying compression level of the result image. `1` means no compression (highest quality) and `0` the highest compression (lowest quality).
 - **format (string)** -- Either `'jpeg'` or `'png'`. Specifies what type of compression should be used and what is the result file extension. PNG compression is lossless but slower, JPEG is faster but the image has

visible artifacts. Defaults to 'jpeg' .

- o **base64 (boolean)** -- Whether to also include the image data in Base64 format.

Returns

Returns `{ uri, width, height }` where `uri` is a URI to the modified image (useable as the source for an `Image / Video` element), `width, height` specify the dimensions of the image. It can contain also `base64` - it is included if the `base64` saveOption was truthy, and is a string containing the JPEG/PNG (depending on `format`) data of the image in Base64--prepend that with `'data:image/xxx;base64,'` to get a data URI, which you can use as the source for an `Image` element for example (where `xxx` is 'jpeg' or 'png').

Basic Example

This will first rotate the image 90 degrees clockwise, then flip the rotated image vertically and save it as a PNG.

```
import React from 'react';
import { Button, TouchableOpacity, Text, View, Image } from 'react-native';
import { Asset, ImageManipulator } from 'expo';

import Colors from '../constants/Colors';

export default class ImageManipulatorSample extends React.Component {
  state = {
    ready: false,
    image: null,
  };

  componentDidMount() {
    (async () => {
      const image = Asset.fromModule(require('../path/to/image.jpg'));
      await image.downloadAsync();
      this.setState({
        ready: true,
        image,
      });
    })();
  }

  render() {
    return (
      <View style={{ flex: 1 }}>
        <View style={{ padding: 10 }}>
          <Button onPress={this._rotate90andFlip} />
          {this.state.ready && this._renderImage()}
        </View>
      </View>
    );
  }

  _rotate90andFlip = async () => {
    const manipResult = await ImageManipulator.manipulateAsync(
      this.state.image.localUri || this.state.image.uri,
      [{ rotate: 90}, { flip: { vertical: true }}],
      { format: 'png' }
    );
    this.setState({ image: manipResult });
  }

  _renderImage = () => {
    return (
      <View style={{marginVertical: 10, alignItems: 'center', justifyContent: 'center'}}>
```

```
        <Image
          source={{ uri: this.state.image.localUri || this.state.image.uri }}
          style={{ width: 300, height: 300, resizeMode: 'contain' }}
        />
      </View>
    );
}
}
```

AppAuth

This module provides access to the native OAuth library AppAuth by [OpenID](#).

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

```
// in managed apps:  
import { AppAuth } from 'expo';  
  
// in bare apps:  
import { AppAuth } from 'expo-app-auth';
```

Methods

authAsync

```
AppAuth.authAsync(props: OAuthProps): Promise<TokenResponse>
```

Starts an OAuth flow and returns authorization credentials.

Parameters

Name	Type	Description
props	OAuthProps	Configuration for the OAuth flow

Return

Name	Type	Description
tokenResponse	Promise<TokenResponse>	Authenticated response token

Example

```
const config = {  
  issuer: 'https://accounts.google.com',  
  clientId: '<CLIENT_ID>',  
  scopes: ['profile'],  
};  
  
const tokenResponse = await AppAuth.authAsync(config);
```

refreshAsync

```
AppAuth.refreshAsync(props: OAuthProps, refreshToken: string): Promise<TokenResponse>
```

Renew the authorization credentials (access token). Some providers may not return a new refresh token.

Parameters

Name	Type	Description
props	OAuthProps	Configuration for the OAuth flow
refreshToken	string	Refresh token to exchange for an Access Token

Return

Name	Type	Description
tokenResponse	Promise<TokenResponse>	Refreshed authentication response token

Example

```
const config = {
  issuer: 'https://accounts.google.com',
  clientId: '<CLIENT_ID>',
  scopes: ['profile'],
};

const tokenResponse = await AppAuth.refreshAsync(config, refreshToken);
```

revokeAsync

```
AppAuth.revokeAsync(props: OAuthBaseProps, options: OAuthRevokeOptions): Promise<any>
```

A fully JS function which revokes the provided access token or refresh token. Use this method for signing-out.
Returns a fetch request.

Parameters

Name	Type	Description
props	OAuthBaseProps	The same OAuth configuratiton used for the initial flow
options	OAuthRevokeOptions	Refresh token or access token to revoke

Example

```
const config = {
  issuer: 'https://accounts.google.com',
  clientId: '<CLIENT_ID>',
};


```

```

const options = {
  token: accessToken, // or a refreshToken
  isClientIdProvided: true,
};

// Sign out...
await AppAuth.revokeAsync(config, options);

```

Constants

AppAuth OAuthRedirect

Redirect scheme used to assemble the `redirectUrl` prop.

AppAuth URLSchemes

iOS only

A list of URL Schemes from the `info.plist`

Types

TokenResponse

Return value of the following `AppAuth` methods:

- `authAsync`
- `refreshAsync`

Name	Type	Description
accessToken	`string`	null`
accessTokenExpirationDate	`string`	null`
additionalParameters	`{ [string]: any }`	null`
idToken	`string`	null`
tokenType	`string`	null`
refreshToken	`string`	undefined`

OAuthBaseProps

Name	Type	Description
clientId	string	The client identifier
issuer	string	URL using the https scheme with no query or fragment component that the OP asserts as its Issuer Identifier
serviceConfiguration	OAuthServiceConfiguration	specifies how to connect to a particular OAuth provider

OAuthProps

extends `OAuthBaseProps`, is used to create OAuth flows.

Name	Type	Description	
clientId	string	The client identifier	
issuer	string	URL using the https scheme with no query or fragment component that the OP asserts as its Issuer Identifier	
serviceConfiguration	OAuthServiceConfiguration	specifies how to connect to a particular OAuth provider	
clientSecret	`string	undefined`	used to prove that identity of the client when exchanging an authorization code for an access token
scopes	`Array	undefined`	a list of space-delimited, case-sensitive strings define the scope of the access requested
redirectUrl	`string	undefined`	The client's redirect URI. Default: <code>AppAuth.OAuthRedirect + ':/oauthredirect'</code>
additionalParameters	OAuthParameters	Extra props passed to the OAuth server request	
canMakeInsecureRequests	`boolean	undefined`	Android: Only enables the use of HTTP requests

OAuthRevokeOptions

Name	Type	Description

token	string	The access token or refresh token to revoke
isClientIdProvided	boolean	Denotes the availability of the Client ID for the token revocation

OAuthServiceConfiguration

Name	Type	Description	
authorizationEndpoint	`string	undefined`	Optional URL of the OP's OAuth 2.0 Authorization Endpoint
registrationEndpoint	`string	undefined`	Optional URL of the OP's Dynamic Client Registration Endpoint
revocationEndpoint	`string	undefined`	Optional URL of the OAuth server used for revoking tokens
tokenEndpoint	string	URL of the OP's OAuth 2.0 Token Endpoint	

OAuthParameters

Learn more about OAuth Parameters on this exciting page: [openid-connect-core](#). To save time I've copied over some of the relevant information, which you can find below.

Name	Type	
nonce	`OAuthNonceParameter	undefined`
display	`OAuthParametersDisplay	undefined`
prompt	`OAuthPromptParameter	undefined`
max_age	`OAuthMaxAgeParameter	undefined`
ui_locales	`OAuthUILocalesParameter	undefined`
id_token_hint	`OAuthIDTokenHintParameter	undefined`
login_hint	`OAuthLoginHintParameter	undefined`
acr_values	`OAuthACRValuesParameter	undefined`

Other parameters MAY be sent. See Sections [3.2.2](#), [3.3.2](#), [5.2](#), [5.5](#), [6](#), and [7.2.1](#) for additional Authorization Request parameters and parameter values defined by this specification.

OAuthDisplayParameter

```
type OAuthDisplayParameter = 'page' | 'popup' | 'touch' | 'wap';
```

ASCII string value that specifies how the Authorization Server displays the authentication and consent user interface pages to the End-User.

Value	Description
page	The Authorization Server SHOULD display the authentication and consent UI consistent with a full User Agent page view. If the display parameter is not specified, this is the default display

	mode.
popup	The Authorization Server SHOULD display the authentication and consent UI consistent with a popup User Agent window. The popup User Agent window should be of an appropriate size for a login-focused dialog and should not obscure the entire window that it is popping up over.
touch	The Authorization Server SHOULD display the authentication and consent UI consistent with a device that leverages a touch interface.
wap	The Authorization Server SHOULD display the authentication and consent UI consistent with a "feature phone" type display.

The Authorization Server MAY also attempt to detect the capabilities of the User Agent and present an appropriate display.

OAuthPromptParameter

```
type OAuthPromptParameter = 'none' | 'login' | 'consent' | 'select_account';
```

Space delimited, case sensitive list of ASCII string values that specifies whether the Authorization Server prompts the End-User for reauthentication and consent.

Value	Description
none	The Authorization Server MUST NOT display any authentication or consent user interface pages. An error is returned if an End-User is not already authenticated or the Client does not have pre-configured consent for the requested Claims or does not fulfill other conditions for processing the request. The error code will typically be <code>login_required</code> , <code>interaction_required</code> , or another code defined in Section 3.1.2.6 . This can be used as a method to check for existing authentication and/or consent.
login	The Authorization Server SHOULD prompt the End-User for reauthentication. If it cannot reauthenticate the End-User, it MUST return an error, typically <code>login_required</code> .
consent	The Authorization Server SHOULD prompt the End-User for consent before returning information to the Client. If it cannot obtain consent, it MUST return an error, typically <code>consent_required</code> .
select_account	The Authorization Server SHOULD prompt the End-User to select a user account. This enables an End-User who has multiple accounts at the Authorization Server to select amongst the multiple accounts that they might have current sessions for. If it cannot obtain an account selection choice made by the End-User, it MUST return an error, typically <code>account_selection_required</code> .

The `prompt` parameter can be used by the Client to make sure that the End-User is still present for the current session or to bring attention to the request. If this parameter contains `none` with any other value, an error is returned.

OAuthNonceParameter

```
type OAuthNonceParameter = string;
```

String value used to associate a Client session with an ID Token, and to mitigate replay attacks. The value is passed through unmodified from the Authentication Request to the ID Token. Sufficient entropy MUST be present in the `nonce` values used to prevent attackers from guessing values. For implementation notes, see [Section 15.5.2](#).

OAuthNonceParameter

```
type OAuthNonceParameter = string;
```

String value used to associate a Client session with an ID Token, and to mitigate replay attacks. The value is passed through unmodified from the Authentication Request to the ID Token. Sufficient entropy MUST be present in the `nonce` values used to prevent attackers from guessing values. For implementation notes, see [Section 15.5.2](#).

OAuthUILocalesParameter

```
type OAuthUILocalesParameter = string;
```

End-User's preferred languages and scripts for the user interface, represented as a space-separated list of [BCP47](#) [RFC5646] language tag values, ordered by preference. For instance, the value "fr-CA fr en" represents a preference for French as spoken in Canada, then French (without a region designation), followed by English (without a region designation). An error SHOULD NOT result if some or all of the requested locales are not supported by the OpenID Provider.

OAuthIDTokenHintParameter

```
type OAuthIDTokenHintParameter = string;
```

ID Token previously issued by the Authorization Server being passed as a hint about the End-User's current or past authenticated session with the Client. If the End-User identified by the ID Token is logged in or is logged in by the request, then the Authorization Server returns a positive response; otherwise, it SHOULD return an error, such as `login_required`. When possible, an `id_token_hint` SHOULD be present when `prompt=none` is used and an `invalid_request` error MAY be returned if it is not; however, the server SHOULD respond successfully when possible, even if it is not present. The Authorization Server need not be listed as an audience of the ID Token when it is used as an `id_token_hint` value. If the ID Token received by the RP from the OP is encrypted, to use it as an `id_token_hint`, the Client MUST decrypt the signed ID Token contained within the encrypted ID Token. The Client MAY re-encrypt the signed ID token to the Authentication Server using a key that enables the server to decrypt the ID Token, and use the re-encrypted ID token as the `id_token_hint` value.

OAuthMaxAgeParameter

```
type OAuthMaxAgeParameter = string;
```

Maximum Authentication Age. Specifies the allowable elapsed time in seconds since the last time the End-User was actively authenticated by the OP. If the elapsed time is greater than this value, the OP MUST attempt to actively re-authenticate the End-User. (The `max_age` request parameter corresponds to the OpenID 2.0 [PAPE](#) [OpenID.PAPE] `max_auth_age` request parameter.) When `max_age` is used, the ID Token returned MUST include an `auth_time` Claim Value.

OAuthLoginHintParameter

```
type OAuthLoginHintParameter = string;
```

OPTIONAL. Hint to the Authorization Server about the login identifier the End-User might use to log in (if necessary). This hint can be used by an RP if it first asks the End-User for their e-mail address (or other identifier) and then wants to pass that value as a hint to the discovered authorization service. It is RECOMMENDED that the hint value match the value used for discovery. This value MAY also be a phone number in the format specified for the `phone_number` Claim. The use of this parameter is left to the OP's discretion.

OAuthACRValuesParameter

```
type OAuthACRValuesParameter = string;
```

Requested Authentication Context Class Reference values. Space-separated string that specifies the acr values that the Authorization Server is being requested to use for processing this Authentication Request, with the values appearing in order of preference. The Authentication Context Class satisfied by the authentication performed is returned as the acr Claim Value, as specified in Section 2. The acr Claim is requested as a Voluntary Claim by this parameter.

Usage

Below is a set of example functions that demonstrate how to use `expo-app-auth` with the Google OAuth Sign-In provider.

```
import { AsyncStorage } from 'react-native';
import { AppAuth } from 'expo-app-auth';
/*
// or from expo directly...
import { AppAuth } from 'expo';
*/

const config = {
  issuer: 'https://accounts.google.com',
  scopes: ['openid', 'profile'],
  /* This is the CLIENT_ID generated from a Firebase project */
  clientId: '603386649315-vp4revvrcgrcjme51ebuhbkbspl04819.apps.googleusercontent.com',
};

/*
 * StorageKey is used for caching the OAuth Key in your app so you can use it later.
 * This can be any string value, but usually it follows this format: @AppName:NameOfValue
 */
const StorageKey = '@PillarValley:GoogleOAuthKey';

/*
 * Notice that Sign-In / Sign-Out aren't operations provided by this module.
 * We emulate them by using authAsync / revokeAsync.
 * For instance if you wanted an "isAuthenticated" flag, you would observe your local tokens.
 * If the tokens exist then you are "Signed-In".
 * Likewise if you cannot refresh the tokens, or they don't exist, then you are "Signed-Out"
 */
async function signInAsync() {
```

```

const authState = await AppAuth.authAsync(config);
await cacheAuthAsync(authState);
console.log('signInAsync', authState);
return authState;
}

/* Let's save our user tokens so when the app resets we can try and get them later */
function cacheAuthAsync(authState) {
  return AsyncStorage.setItem(StorageKey, JSON.stringify(authState));
}

/* Before we start our app, we should check to see if a user is signed-in or not */
async function getCachedAuthAsync() {
  /* First we will try and get the cached auth */
  const value = await AsyncStorage.getItem(StorageKey);
  /* Async Storage stores data as strings, we should parse our data back into a JSON */
  const authState = JSON.parse(value);
  console.log('getCachedAuthAsync', authState);
  if (authState) {
    /* If our data exists, than we should see if it's expired */
    if (checkIfTokenExpired(authState)) {
      /*
        * The session has expired.
        * Let's try and refresh it using the refresh token that some
        * OAuth providers will return when we sign-in initially.
      */
      return refreshAuthAsync(authState);
    } else {
      return authState;
    }
  }
  return null;
}

/*
 * You might be familiar with the term "Session Expired", this method will check if our session has expired.
 * An expired session means that we should reauthenticate our user.
 * You can learn more about why on the internet: https://www.quora.com/Why-do-web-sessions-expire
 * > Fun Fact: Charlie Cheever the creator of Expo also made Quora :D
 */
function checkIfTokenExpired({ accessTokenExpirationDate }) {
  return new Date(accessTokenExpirationDate) < new Date();
}

/*
 * Some OAuth providers will return a "Refresh Token" when you sign-in initially.
 * When our session expires, we can exchange the refresh token to get new auth tokens.
 * > Auth tokens are not the same as a Refresh token
 *
 * Not every provider (very few actually) will return a new "Refresh Token".
 * This just means the user will have to Sign-In more often.
 */
async function refreshAuthAsync({ refreshToken }) {
  const authState = await AppAuth.refreshAsync(config, refreshToken);
  console.log('refreshAuthAsync', authState);
  await cacheAuthAsync(authState);
  return authState;
}

/*
 * To sign-out we want to revoke our tokens.
 * This is what high-level auth solutions like FBSDK are doing behind the scenes.
 */
async function signOutAsync({ accessToken }) {
  try {
    await AppAuth.revokeAsync(config, {

```

```
    token: accessToken,
    clientIdProvided: true,
});
/*
 * We are removing the cached tokens so we can check on our auth state later.
 * No tokens = Not Signed-In :)
 */
await AsyncStorage.removeItem(StorageKey);
return null;
} catch ({ message }) {
  alert(`Failed to revoke token: ${message}`);
}
}
```

AppLoading

A React component that tells Expo to keep the app loading screen open if it is the first and only component rendered in your app. Unless `autoHideSplash` prop is set to `false` the loading screen will disappear and your app will be visible when the component is removed.

This is incredibly useful to let you download and cache fonts, logo and icon images and other assets that you want to be sure the user has on their device for an optimal experience before rendering they start using the app.

Installation

This API is pre-installed in [managed](#) apps. It is not available for [bare](#) React Native apps.

Usage

```
import React from 'react';
import { Image, Text, View } from 'react-native';
import { Asset, AppLoading } from 'expo';

export default class App extends React.Component {
  state = {
    isReady: false,
  };

  render() {
    if (!this.state.isReady) {
      /* @info As long as AppLoading is the only leaf/native component that has been mounted, the loading screen will remain visible */
      return (
        <AppLoading
          startAsync={this._cacheResourcesAsync}
          onFinish={() => this.setState({ isReady: true })}
          onError={console.warn}
        />
      );/* @end */
    }
  }

  return (
    <View style={{ flex: 1 }}>
      <Image source={require('./assets/images/expo-icon.png')} />
      <Image source={require('./assets/images/slack-icon.png')} />
    </View>
  );
}

async _cacheResourcesAsync() {
  const images = [
    require('./assets/images/expo-icon.png'),
    require('./assets/images/slack-icon.png'),
  ];
  /* @info Read more about <a href='../guides/preloading-and-caching-assets.html'>Preloading and Caching Assets</a> */
  const cacheImages = images.map((image) => {
    return Asset.fromModule(image).downloadAsync();
  });
}
```

```
});/* @end */

return Promise.all(cacheImages)

}

}
```

API

```
import { AppLoading } from 'expo';
```

props

The following props are recommended, but optional for the sake of backwards compatibility (they were introduced in SDK21). If you do not provide any props, you are responsible for coordinating loading assets, handling errors, and updating state to unmount the `AppLoading` component.

- **startAsync (function)** -- A `function` that returns a `Promise`, and the `Promise` should resolve when the app is done loading required data and assets.
- **onError (function)** -- If `startAsync` throws an error, it is caught and passed into the function provided to `onError`.
- **onFinish (function)** -- (**Required if you provide `startAsync`**). Called when `startAsync` resolves or rejects. This should be used to set state and unmount the `AppLoading` component.
- **autoHideSplash (boolean)** -- Whether to hide the native splash screen as soon as you unmount the `AppLoading` component. See [SplashScreen module](#) for an example.

AR

ARCore is not yet supported. This lib is iOS only right now. Augmented Reality with ARKit for iOS This library is generally used with [expo-three](#) to generate a camera, and manage a 3D scene.

Installation

This API is pre-installed in [managed](#) apps. It is not available for [bare](#) React Native apps.

API

```
import { AR } from 'expo';
```

Examples can be found here

Getting Started

Here is an example of a 3D scene that is configured with `three.js` and `Expo.AR`

Availability

`isAvailable()`

This will check the following condition:

- Device year is greater than 2014
- Device is not a simulator
- Device is iOS, and not Android

`getVersion()`

Get the version of ARKit running on the device. iOS 11 devices come with `1.0`, and the newly released iOS 11.3 comes with ARKit `1.5` ;

Listeners

All listeners will return an object capable of removing itself as such:

```
const listener = AR.onFrameDidUpdate(() => {});
listener.remove();
```

Optionally you can also remove all listeners for a single event.

`removeAllListeners(eventType)`

```
AR.removeAllListeners(AR.EventTypes.FrameDidUpdate);
```

```
onFrameDidUpdate( () => {} )
```

This will update everytime the ARSession has updated it's frame (has new data)

```
onDidFailWithError( ({ error }) => {} )
```

This will be called with the localized description of any Error thrown by ARKit

```
onAnchorsDidUpdate( ({ anchors: Array, eventType: AnchorEventTypes }) => {} )
```

Invoked when an anchor is found, updated, or removed. This is the primary way to get data from Detection Images, Planes, Faces, and general Anchors.

Example

```
AR.onAnchorsDidUpdate(({ anchors, eventType }) => {
  for (let anchor of anchors) {
    if (anchor.type === AR.AnchorTypes.Anchor) {
      const { identifier, transform } = anchor;

      if (eventType === AR.AnchorEventTypes.Add) {
        // Something added!
      } else if (eventType === AR.AnchorEventTypes.Remove) {
        // Now it's changed
      } else if (eventType === AR.AnchorEventTypes.Update) {
        // Now it's gone...
      }
    }
  }
});
```

```
onCameraDidChangeTrackingState( { trackingState: TrackingState,
trackingStateReason: TrackingStateReason } ) => {} )
```

Called whenever the camera changes it's movement / tracking state. Useful for telling the user how to better hold and move the camera.

```
onSessionWasInterrupted( () => {} )
```

Simply this is called when the app backgerounds.

```
onSessionInterruptionEnded( () => {} )
```

This is called when the app returns to the foreground.

Hit Testing

Maybe the most powerful function, hit testing allows you to get real world info on a particular position in the screen.

```
performHitTest(point, types: HitTestResultType)
```

The `point` is a normalized value, meaning it is **between 0-1** this can be achieved by dividing the dimension position by size. Ex: `{ x: x / width, y: y / height }`

Example

```
const normalizedPoint = { x, y };
const hitTestResultTypes = AR.HitTestResultTypes.HorizontalPlane;
const { hitTest } = AR.performHitTest(normalizedPoint, hitTestResultTypes);
for (let hit of hitTest) {
  const { worldTransform, type, localTransform, distance, anchor: { identifier, transform } } = hit;
}
```

Constants

Possible `types` for specifying a hit-test search, or for the result of a hit-test search.

```
HitTestResultTypes = {
  /**
   * Result type from intersecting the nearest feature point.
   */
  FeaturePoint: 'featurePoint',
  /**
   * Result type from intersecting a horizontal plane estimate, determined for the current frame.
   */
  HorizontalPlane: 'horizontalPlane',
  /**
   * Result type from intersecting a vertical plane estimate, determined for the current frame.
   */
  VerticalPlane: 'verticalPlane',
  /**
   * Result type from intersecting with an existing plane anchor.
   */
  ExistingPlane: 'existingPlane',
  /**
   * Result type from intersecting with an existing plane anchor, taking into account the plane's extent.
   */
  ExistingPlaneUsingExtent: 'existingPlaneUsingExtent',
  /**
   * Result type from intersecting with an existing plane anchor, taking into account the plane's geometry.
   */
  ExistingPlaneUsingGeometry: 'existingPlaneUsingGeometry',
};
```

Detection Images

Given an image, ARKit will update you when it finds it in the real world.

Make sure that all reference images are greater than 100 pixels and have a positive physical size in meters.

```
setDetectionImagesAsync(images)
```

Example

```
const asset = Expo.Asset.fromModule(require('./image.png'))
await asset.downloadAsync();

await AR.setDetectionImagesAsync({
  myDopeImage: {
    /**
     * The local uri of the image, this can be obtained with Expo.Asset.fromModule()
     */
    uri: asset.localUri,
    /**
     */
```

```

    * Name used to identify the Image Anchor returned in a `onAnchorsDidUpdate` listener.
    */
  name: 'myDopeImage',
  /**
   * Real-world size in meters.
   */
  width: 0.1,
},
...
});

AR.onAnchorsDidUpdate(({anchors, eventType}) => {
  for (let anchor of anchors) {
    if (anchor.type === AR.AnchorTypes.Image) {
      const { identifier, image, transform } = anchor;

      if (eventType === AR.AnchorEventTypes.Add) {
        // Add some node
      } else if (eventType === AR.AnchorEventTypes.Remove) {
        // Remove that node
      } else if (eventType === AR.AnchorEventTypes.Update) {
        // Update whatever node
      }
    }
  }
})

```

Raw Data

This synchronous function can return anchors, raw feature points, light estimation, and captured depth data.

`getCurrentFrame(attributes: ?ARFrameRequest): ?ARFrame`

This method can be used to access frame data from the `ARSession`. Because not all frame data is needed for most tasks; you can request which props you wish to receive, with an `ARFrameRequest`.

```

type ARFrameRequest = {
  anchors?: ARframeAnchorRequest,
  rawFeaturePoints?: boolean,
  lightEstimation?: boolean,
  capturedDepthData?: boolean,
};

const FrameAttributes = {
  Anchors: 'anchors',
  RawFeaturePoints: 'rawFeaturePoints',
  LightEstimation: 'lightEstimation',
  CapturedDepthData: 'capturedDepthData',
};

```

An example of the input:

```

const {
  anchors,
  // An array of raw feature points: { x: number, y: number, z: number, id: string }
  rawFeaturePoints,
  // The basic light estimation data, this will return
  lightEstimation,
  // Unfortunately we had to remove this prop. You cannot access it at the moment
  capturedDepthData,
  // timestamp is included by default
}

```

```

    timestamp,
} = AR.getCurrentFrame({
  // We want to get the anchor data, and include the Face Anchor
  anchors: {
    [AR.AnchorTypes.Face]: {
      blendShapes: true,
      geometry: true,
    },
  },
  rawFeaturePoints: true,
  lightEstimation: true,
  // Not available
  capturedDepthData: true,
});

```

Depending on what you provided, you will receive an `ARFrame`.

```

type ARFrame = {
  // The timestamp of the frame will be passed back everytime
  timestamp: number,
  // Serialized array of anchors, by default each will have: type, transform, and id.
  // You can filter these by `type` if you wish.
  anchors?: ?Array<Anchor>,
  // A RawFeaturePoint will have {x,y,z,id}.
  // This can be visualized with `ExpoTHREE.AR.Points`.
  rawFeaturePoints?: ?Array<RawFeaturePoint>,
  // The light estimation will return `ambientIntensity` (Lumens) and `ambientColorTemperature` (Kelvin)
  // An example of how to use these values can be found in `ExpoTHREE.AR.Light`
  lightEstimation?: ?LightEstimation,
};

```

FrameAttributes

Here is a breakdown on the keys, and their return values.

FrameAttributes.Actors

The input to this value can be used to capture complex face data. Because there is a lot of face data, we don't want to get everything all the time.

```

type ARFrameAnchorRequest = {
  // You pass in the anchor's class name.
  // Currently only `ARFaceAnchor` is supported.
  ARFaceAnchor?: {
    // When the value is `true` all `BlendShapes` will be returned.
    // Optionally you can pass in an object that will only include some of the `BlendShapes`.
    // Ex: `{ [AR.BlendShapes.CheekPuff]: true }` will send back just the puffed cheek value.
    blendShapes?: boolean | { [BlendShape]: boolean },
    // [Experimental]: If included and true, this will return all the data required to create the face mesh.
    // This will freeze the the thread, as there is a lot of data.
    // Currently looking into a better way to return this.
    geometry?: boolean,
  },
};

// This will return just the amount your left and right eyebrows are down.
const { anchors } = AR.getCurrentFrame({
  [AR.FrameAttributes.Anchors]: {
    [AR.AnchorTypes.Face]: {
      blendShapes: [AR.BlendShapes.BrowDownL, AR.BlendShapes.BrowDownR],
    },
  },
});

```

```

});
```

```

const {
  [AR.AnchorTypes.Face]: {
    blendShapes: {
      [AR.BlendShapes.BrowDownL]: browDownLValue,
      [AR.BlendShapes.BrowDownR]: browDownRValue
    }
  }
} = anchors;
```

```

console.log(browDownLValue, browDownRValue);
```

The output value will be an array of `Anchors`

```

type Anchor = {
  // Use this value to determine if the anchor is a plane/image/face
  type: AnchorType,
  transform: Matrix,
  id: string,

  // ARPlaneAnchor only
  // This is the origin offset from the center of the plane.
  // { x: number, z: number }
  center?: Vector3,
  // The size of the plane
  extent?: { width: number, length: number },

  // ARImageAnchor only
  image?: {
    name: ?string,
    // Size in meters
    size: Size,
  },
  // ARFaceAnchor only
  geometry?: FaceGeometry,
  blendShapes?: { [BlendShape]: number },
};
```

FrameAttributes.RawFeaturePoints

When this key is provided an array of raw feature points will be returned. Examples on usage can be found in `expo-three`

```

type RawFeaturePoint = {
  x: number,
  y: number,
  z: number,
  id: string
};
```

FrameAttributes.LightEstimation

ARKit will try and estimate what the room lighting is. With this data you can render your scene with similar lighting. Checkout the Lighting demo in `expo-three` for a better idea of how to use this data.

```

export type LightEstimation = {
  // Lumens - brightness
  ambientIntensity: number,
  // Kelvin - color
```

```
    ambientColorTemperature: number,  
    // Not available yet - front facing props  
    primaryLightDirection?: Vector3,  
    primaryLightIntensity?: number,  
};
```

Anchors

This can return the following Anchor Types:

- ARAnchor
- ARPlaneAnchor
- ARImageAnchor
- ARFaceAnchor

ARAnchor

```
type: "ARAnchor", // AR.AnchorTypes.Anchor  
transform: anchor.transform,  
identifier: anchor.identifier
```

ARPlaneAnchor

```
type: "ARPlaneAnchor", // AR.AnchorTypes.Plane  
transform: anchor.transform,  
identifier: anchor.identifier,  
center: {  
    x: Float,  
    y: Float,  
    z: Float  
},  
extent: {  
    width: Float,  
    length: Float  
}
```

ARImageAnchor

```
type: "ARImageAnchor", // AR.AnchorTypes.Image  
transform: anchor.transform,  
identifier: anchor.identifier,  
image: {  
    name: anchor.referenceImage.name,  
    size: { // Physical size in meters  
        width: Float,  
        height: Float,  
    }  
}
```

ARFaceAnchor

```
type: "ARFaceAnchor", // AR.AnchorTypes.Face  
transform: anchor.transform,  
identifier: anchor.identifier,  
isTracked: Bool,  
geometry: {  
    vertexCount: Int, // ARFaceAnchor.geometry.vertexCount  
    textureCoordinateCount: Int, // ARFaceAnchor.geometry.textureCoordinateCount
```

```

triangleCount: Int, // ARFaceAnchor.geometry.triangleCount
vertices: [ { x: Float, y: Float, z: Float } ],
textureCoordinates: [ { u: Float, v: Float } ],
triangleIndices: [ Int ],
},
blendShapes: {
  browDown_L: Float, // AR.BlendShapes.BrowDownL
  browDown_R: Float, // AR.BlendShapes.BrowDownR
  browInnerUp: Float, // AR.BlendShapes.BrowInnerUp
  browOuterUp_L: Float, // AR.BlendShapes.BrowOuterUpL
  browOuterUp_R: Float, // AR.BlendShapes.BrowOuterUpR
  cheekPuff: Float, // AR.BlendShapes.CheekPuff
  cheekSquint_L: Float, // AR.BlendShapes.CheekSquintL
  cheekSquint_R: Float, // AR.BlendShapes.CheekSquintR
  eyeBlink_L: Float, // AR.BlendShapes.EyeBlinkL
  eyeBlink_R: Float, // AR.BlendShapes.EyeBlinkR
  eyeLookDown_L: Float, // AR.BlendShapes.EyeLookDownL
  eyeLookDown_R: Float, // AR.BlendShapes.EyeLookDownR
  eyeLookIn_L: Float, // AR.BlendShapes.EyeLookInL
  eyeLookIn_R: Float, // AR.BlendShapes.EyeLookInR
  eyeLookOut_L: Float, // AR.BlendShapes.EyeLookOutL
  eyeLookOut_R: Float, // AR.BlendShapes.EyeLookOutR
  eyeLookUp_L: Float, // AR.BlendShapes.EyeLookUpL
  eyeLookUp_R: Float, // AR.BlendShapes.EyeLookUpR
  eyeSquint_L: Float, // AR.BlendShapes.EyeSquintL
  eyeSquint_R: Float, // AR.BlendShapes.EyeSquintR
  eyeWide_L: Float, // AR.BlendShapes.EyeWideL
  eyeWide_R: Float, // AR.BlendShapes.EyeWideR
  jawForward: Float, // AR.BlendShapes.JawForward
  jawLeft: Float, // AR.BlendShapes.JawLeft
  jawOpen: Float, // AR.BlendShapes.JawOpen
  jawRight: Float, // AR.BlendShapes.JawRight
  mouthClose: Float, // AR.BlendShapes.MouthClose
  mouthDimple_L: Float, // AR.BlendShapes.MouthDimpleL
  mouthDimple_R: Float, // AR.BlendShapes.MouthDimpleR
  mouthFrown_L: Float, // AR.BlendShapes.MouthFrownL
  mouthFrown_R: Float, // AR.BlendShapes.MouthFrownR
  mouthFunnel: Float, // AR.BlendShapes.MouthFunnel
  mouthLeft: Float, // AR.BlendShapes.MouthLeft
  mouthLowerDown_L: Float, // AR.BlendShapes.MouthLowerDownL
  mouthLowerDown_R: Float, // AR.BlendShapes.MouthLowerDownR
  mouthPress_L: Float, // AR.BlendShapes.MouthPressL
  mouthPress_R: Float, // AR.BlendShapes.MouthPressR
  mouthPucker: Float, // AR.BlendShapes.MouthPucker
  mouthRight: Float, // AR.BlendShapes.MouthRight
  mouthRollLower: Float, // AR.BlendShapes.MouthRollLower
  mouthRollUpper: Float, // AR.BlendShapes.MouthRollUpper
  mouthShrugLower: Float, // AR.BlendShapes.MouthShrugLower
  mouthShrugUpper: Float, // AR.BlendShapes.MouthShrugUpper
  mouthSmile_L: Float, // AR.BlendShapes.MouthSmileL
  mouthSmile_R: Float, // AR.BlendShapes.MouthSmileR
  mouthStretch_L: Float, // AR.BlendShapes.MouthStretchL
  mouthStretch_R: Float, // AR.BlendShapes.MouthStretchR
  mouthUpperUp_L: Float, // AR.BlendShapes.MouthUpperUpL
  mouthUpperUp_R: Float, // AR.BlendShapes.MouthUpperUpR
  noseSneer_L: Float, // AR.BlendShapes.NoseSneerL
  noseSneer_R: Float, // AR.BlendShapes.NoseSneerR
}

```

Camera Data

Matrix data can be used in `three.js` (with [expo-three](#)) to generate a 3D camera.

getARMatrices(near: number, far: number)

```
getARMatrices(arsession, width, height, near, far) is now getARMatrices(near, far)
```

Build up / Tear down

Given reference to a `GLView` and a `ARTrackingConfiguration`, this will create an `ARSession` associated with the `GLContext` `startARSessionAsync(view, trackingConfiguration)`

```
startARSessionAsync is now startAsync
```

When invoked, this method will tear-down the `ARSession`, and `WebGLTexture` used for the camera stream. This is an end-of-lifecycle method. `stopAsync()`

```
stopARSessionAsync() is now stopAsync
```

Running State

Used to reset the anchors and current data in the `ARSession`. `reset()`

Used to pause the `ARSession`. `pause()`

Used to resume the `ARSession` after pausing it. `resume()`

Configuration

Check availability.

```
isConfigurationAvailable(configuration: TrackingConfiguration): Bool
```

A Configuration defines how ARKit constructs a scene based on real-world device motion.

```
setConfigurationAsync(configuration: TrackingConfiguration)
```

Alternatively you could also use:

- `isFrontCameraAvailable()`
- `isRearCameraAvailable()`

```
TrackingConfigurations = {
    /**
     * Provides high-quality AR experiences that use the rear-facing camera precisely track a device's position and orientation and allow plane detection and hit testing.
     */
    world: 'ARWorldTrackingConfiguration',
    /**
     * Provides basic AR experiences that use the rear-facing camera and track only a device's orientation.
     */
    orientation: 'AROrientationTrackingConfiguration',
    /**
     * Provides AR experiences that use the front-facing camera and track the movement and expressions of the user's face.
     */
    face: 'ARFaceTrackingConfiguration',
};
```

Plane Detection

Used to enable or disable plane detection (`ARPlaneAnchor` s will be able to return in `onAnchorsDidUpdate`).

As of iOS 11.3 (ARKit 1.5) you can now enable vertical plane detection.

- **Default:** `AR.PlaneDetectionTypes.None`
- **GET:** `planeDetection(): PlaneDetection`
- **SET:** `setPlaneDetection(planeDetection: PlaneDetection)`

Constants

Options for whether and how ARKit detects flat surfaces in captured images.

```
PlaneDetectionTypes = {  
    /**  
     * No plane detection is run.  
     */  
    None: 'none',  
    /**  
     * Plane detection determines horizontal planes in the scene.  
     */  
    Horizontal: 'horizontal',  
    /**  
     * Plane detection determines vertical planes in the scene.  
     */  
    Vertical: 'vertical',  
};
```

World Origin

The center of the 3D space used by ARKit

```
setWorldOriginAsync(matrix4x4)
```

A Matrix 4x4 is an array of 16 doubles

```
`setWorldOriginAsync([1,1,1,1,0,0,0,0,0,0,0,0,0])`;
```

Light Estimation

Estimated scene lighting information associated with a captured video frame in an AR session.

- **Default:** `true`
- **SET:** `setLightEstimationEnabled(value: Boolean)`
- **GET:** `getLightEstimationEnabled(): Boolean`

Light estimation can be retrieved through `getCurrentFrame` with the `lightEstimation` key added.

Provides Audio Data

A Boolean value that specifies whether to capture audio during the AR session. You cannot currently access the audio data as it is useless.

- **Default:** `false`
- **SET:** `setProvidesAudioData(value: Boolean)`
- **GET:** `getProvidesAudioData(): Boolean`

Auto Focus

iOS 11.3+ A Boolean value that determines whether the device camera uses fixed focus or autofocus behavior.

- **Default:** true
- **SET:** setAutoFocusEnabled(value: Boolean)
- **GET:** getAutoFocusEnabled(): Boolean

World Alignment

Options for how ARKit constructs a scene coordinate system based on real-world device motion.

- **Default:** AR.WorldAlignmentTypes.Gravity
- **SET:** setWorldAlignment(worldAlignment: WorldAlignment)
- **GET:** getWorldAlignment(): WorldAlignment

```
const WorldAlignmentTypes = {  
  Gravity: 'gravity',  
  GravityAndHeading: 'gravityAndHeading',  
  AlignmentCamera: 'alignmentCamera',  
};
```

Camera Texture

The internal ID used to render the camera texture.

- **GET:** getCameraTexture(): Number

Supported Video Formats

iOS 11.3+ The set of video capture formats available on the current device. The video format cannot be set yet.

- **Default:** The first element returned is the default value
- **GET:** AR.getSupportedVideoFormats(configuration: TrackingConfiguration): Array

```
{  
  imageResolution: {  
    width: 1920,  
    height: 1080  
  },  
  framesPerSecond: 60  
}
```

ART

React Native comes with a built in library for simple vector drawing called ART. It is barely documented and instead you likely want to use [Svg](#) which is more feature complete and better documented and more standard implementation of vector graphics.

But sometimes you'll find some code that uses ART such as [react-native-progress](#) and you'll want to be able to run it on Expo, and, since ART is built in to React Native, you can.

Again, you almost definitely want to use [Svg](#) instead of this unless you are really sure you want to use ART for some reason.

#

Asset

This module provides an interface to Expo's asset system. An asset is any file that lives alongside the source code of your app that the app needs at runtime. Examples include images, fonts and sounds. Expo's asset system integrates with React Native's, so that you can refer to files with `require('path/to/file')`. This is how you refer to static image files in React Native for use in an `Image` component, for example. Check out React Native's [documentation on static image resources](#) for more information. This method of referring to static image resources works out of the box with Expo.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Asset } from 'expo';  
  
// in bare apps:  
import { Asset } from 'expo-asset';
```

This class represents an asset in your app. It gives metadata about the asset (such as its name and type) and provides facilities to load the asset data.

- `name`

The name of the asset file without the extension. Also without the part from `@` onward in the filename (used to specify scale factor for images).

- `type`

The extension of the asset filename

- `hash`

The MD5 hash of the asset's data

- `uri`

A URI that points to the asset's data on the remote server. When running the published version of your app, this refers to the location on Expo's asset server where Expo has stored your asset. When running the app from Expo CLI during development, this URI points to Expo CLI's server running on your computer and the asset is served directly from your computer.

- `localUri`

If the asset has been downloaded (by calling `downloadAsync()`), the `file://` URI pointing to the local file on the device that contains the asset data.

- `width`

If the asset is an image, the width of the image data divided by the scale factor. The scale factor is the number after `@` in the filename, or `1` if not present.

- `height`

If the asset is an image, the height of the image data divided by the scale factor. The scale factor is the number after `@` in the filename, or `1` if not present.

- `downloadAsync()`

Downloads the asset data to a local file in the device's cache directory. Once the returned promise is fulfilled without error, the `localUri` field of this asset points to a local file containing the asset data. The asset is only downloaded if an up-to-date local file for the asset isn't already present due to an earlier download.

Asset.loadAsync(modules)

A helper that wraps `Asset.fromModule(module).downloadAsync` for convenience.

Arguments

- **modules (Array|number)** -- An array of `require('path/to/file')`. Can also be just one module without an Array.

Returns

Returns a Promise that resolves when the asset has been saved to disk.

Asset.fromModule(module)

Returns the `Asset` instance representing an asset given its module

Arguments

- **module (number)** -- The value of `require('path/to/file')` for the asset

Returns

The `Asset` instance for the asset

Example

```
const imageURI = Asset.fromModule(require('./images/hello.jpg')).uri;
```

On running this piece of code, `imageURI` gives the remote URI that the contents of `images/hello.jpg` can be read from. The path is resolved relative to the source file that this code is evaluated in.

```
#
```


Audio

Provides basic sample playback and recording.

Note that Expo does not yet support backgrounding, so audio is not available to play in the background of your experience. Audio also automatically stops if headphones / bluetooth audio devices are disconnected.

Try the [playlist example app](#) (source code is [on GitHub](#)) to see an example usage of the media playback API, and the [recording example app](#) (source code is [on GitHub](#)) to see an example usage of the recording API.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Audio } from 'expo';  
  
// in bare apps:  
import { Audio } from 'expo-av';
```

Enabling Audio and customizing Audio Mode

Audio.setIsEnabledAsync(value)

Audio is enabled by default, but if you want to write your own Audio API in an ExpoKit app, you might want to disable the Audio API.

Arguments

- **value (boolean)** -- `true` enables Audio, and `false` disables it.

Returns

A `Promise` that will reject if audio playback could not be enabled for the device.

Audio.setAudioModeAsync(mode)

We provide this API to customize the audio experience on iOS and Android.

Arguments

- **mode (object)** --

A dictionary with the following key-value pairs:

- `playsInSilentModeiOS` : a boolean selecting if your experience's audio should play in silent mode on iOS. This value defaults to `false`.
- `allowsRecordingiOS` : a boolean selecting if recording is enabled on iOS. This value defaults to `false`. NOTE: when this flag is set to `true`, playback may be routed to the phone receiver instead of to the speaker.
- `interruptionModeiOS` : an enum selecting how your experience's audio should interact with the audio from other apps on iOS:
 - `INTERRUPTION_MODE_IOS_MIX_WITH_OTHERS` : This is the default option. If this option is set, your experience's audio is mixed with audio playing in background apps.
 - `INTERRUPTION_MODE_IOS_DO_NOT_MIX` : If this option is set, your experience's audio interrupts audio from other apps.
 - `INTERRUPTION_MODE_IOS_DUCK_OTHERS` : If this option is set, your experience's audio lowers the volume ("ducks") of audio from other apps while your audio plays.
- `shouldDuckAndroid` : a boolean selecting if your experience's audio should automatically be lowered in volume ("duck") if audio from another app interrupts your experience. This value defaults to `true`. If `false`, audio from other apps will pause your audio.
- `interruptionModeAndroid` : an enum selecting how your experience's audio should interact with the audio from other apps on Android:
 - `INTERRUPTION_MODE_ANDROID_DO_NOT_MIX` : If this option is set, your experience's audio interrupts audio from other apps.
 - `INTERRUPTION_MODE_ANDROID_DUCK_OTHERS` : This is the default option. If this option is set, your experience's audio lowers the volume ("ducks") of audio from other apps while your audio plays.
- `playThroughEarpieceAndroid` : set to true to route audio to earpiece (on Android).

Returns

A `Promise` that will reject if the audio mode could not be enabled for the device. Note that these are the only legal AudioMode combinations of (`playsInSilentModeiOS`, `allowsRecordingiOS`, `interruptionModeiOS`), and any other will result in promise rejection:

- `false, false, INTERRUPTION_MODE_IOS_DO_NOT_MIX`
- `false, false, INTERRUPTION_MODE_IOS_MIX_WITH_OTHERS`
- `true, true, INTERRUPTION_MODE_IOS_DO_NOT_MIX`
- `true, true, INTERRUPTION_MODE_IOS_DUCK_OTHERS`
- `true, true, INTERRUPTION_MODE_IOS_MIX_WITH_OTHERS`
- `true, false, INTERRUPTION_MODE_IOS_DO_NOT_MIX`
- `true, false, INTERRUPTION_MODE_IOS_DUCK_OTHERS`
- `true, false, INTERRUPTION_MODE_IOS_MIX_WITH_OTHERS`

Playing sounds

Audio.Sound

This class represents a sound corresponding to an Asset or URL.

Returns

A newly constructed instance of `Audio.Sound`.

Example

```
const soundObject = new Audio.Sound();
try {
  await soundObject.loadAsync(require('./assets/sounds/hello.mp3'));
  await soundObject.playAsync();
  // Your sound is playing!
} catch (error) {
  // An error occurred!
}
```

A static convenience method to construct and load a sound is also provided:

- `Audio.Sound.createAsync(source, initialStatus = {}, onPlaybackStatusUpdate = null, downloadFirst = true)`

Creates and loads a sound from source, with optional `initialStatus`, `onPlaybackStatusUpdate`, and `downloadFirst`.

This is equivalent to the following:

```
const soundObject = new Audio.Sound();
soundObject.setOnPlaybackStatusUpdate(onPlaybackStatusUpdate);
await soundObject.loadAsync(source, initialStatus, downloadFirst);
```

Parameters

- **source (*object / number / Asset*)** -- The source of the sound. The following forms are supported:
 - A dictionary of the form `{ uri: string, headers?: { [string]: string }, overrideFileExtensionAndroid?: string }` with a network URL pointing to a media file on the web, an optional headers object passed in a network request to the `uri` and an optional Android-specific `overrideFileExtensionAndroid` string overriding extension inferred from the URL. The `overrideFileExtensionAndroid` property may come in handy if the player receives an URL like `example.com/play` which redirects to `example.com/player.m3u8`. Setting this property to `m3u8` would allow the Android player to properly infer the content type of the media and use proper media file reader.
 - `require('path/to/file')` for an audio file asset in the source code directory.
 - An `Expo.Asset` object for an audio file asset.
- **initialStatus (*PlaybackStatusToSet*)** -- The initial intended `PlaybackStatusToSet` of the sound, whose values will override the default initial playback status. This value defaults to `{}` if no parameter is passed. See the [AV documentation](#) for details on `PlaybackStatusToSet` and the default initial playback status.
- **onPlaybackStatusUpdate (*function*)** -- A function taking a single parameter `PlaybackStatus`. This value defaults to `null` if no parameter is passed. See the [AV documentation](#) for details on the functionality provided by `onPlaybackStatusUpdate`.
- **downloadFirst (*boolean*)** -- If set to true, the system will attempt to download the resource to the device before loading. This value defaults to `true`. Note that at the moment, this will only work for `source`s of the form `require('path/to/file')` or `Asset` objects.

Returns

A `Promise` that is rejected if creation failed, or fulfilled with the following dictionary if creation succeeded:

- `sound` : the newly created and loaded `Sound` object.
- `status` : the `PlaybackStatus` of the `Sound` object. See the [AV documentation](#) for further information.

Example

```
try {
  const { sound: soundObject, status } = await Audio.Sound.createAsync(
    require('../assets/sounds/hello.mp3'),
    { shouldPlay: true }
  );
  // Your sound is playing!
} catch (error) {
  // An error occurred!
}
```

The rest of the API for `Audio.Sound` is the same as the imperative playback API for `Expo.Video` -- see the [AV documentation](#) for further information:

- `soundObject.loadAsync(source, initialStatus = {}, downloadFirst = true)`
- `soundObject.unloadAsync()`
- `soundObject.getStatusAsync()`
- `soundObject.setOnPlaybackStatusUpdate(onPlaybackStatusUpdate)`
- `soundObject.setStatusAsync(statusToSet)`
- `soundObject.playAsync()`
- `soundObject.replayAsync()`
- `soundObject.pauseAsync()`
- `soundObject.stopAsync()`
- `soundObject.setPositionAsync(millis)`
- `soundObject.setRateAsync(value, shouldCorrectPitch)`
- `soundObject.setVolumeAsync(value)`
- `soundObject.setIsMutedAsync(value)`
- `soundObject.setIsLoopingAsync(value)`
- `soundObject.setProgressUpdateIntervalAsync(millis)`

Recording sounds

Audio.Recording

This class represents an audio recording. After creating an instance of this class, `prepareToRecordAsync` must be called in order to record audio. Once recording is finished, call `stopAndUnloadAsync`. Note that only one recorder is allowed to exist in the state between `prepareToRecordAsync` and `stopAndUnloadAsync` at any given time.

Note that your experience must request audio recording permissions in order for recording to function. See the [Permissions module](#) for more details.

Returns

A newly constructed instance of `Audio.Recording`.

Example

```
const recording = new Audio.Recording();
try {
  await recording.prepareToRecordAsync(Audio.RECORDING_OPTIONS_PRESET_HIGH_QUALITY);
  await recording.startAsync();
  // You are now recording!
} catch (error) {
  // An error occurred!
}
```

- `recordingInstance.getStatusAsync()`

Gets the `status` of the `Recording`.

Returns

A `Promise` that is resolved with the `status` of the `Recording`: a dictionary with the following key-value pairs.

Before `prepareToRecordAsync` is called, the `status` will be as follows:

- `canRecord` : a boolean set to `false`.
- `isDoneRecording` : a boolean set to `false`.

After `prepareToRecordAsync()` is called, but before `stopAndUnloadAsync()` is called, the `status` will be as follows:

- `canRecord` : a boolean set to `true`.
- `isRecording` : a boolean describing if the `Recording` is currently recording.
- `durationMillis` : the current duration of the recorded audio.

After `stopAndUnloadAsync()` is called, the `status` will be as follows:

- `canRecord` : a boolean set to `false`.
- `isDoneRecording` : a boolean set to `true`.
- `durationMillis` : the final duration of the recorded audio.

- `recordingInstance.setOnRecordingStatusUpdate(onRecordingStatusUpdate)`

Sets a function to be called regularly with the `status` of the `Recording`. See `getStatusAsync()` for details on `status`.

`onRecordingStatusUpdate` will be called when another call to the API for this recording completes (such as `prepareToRecordAsync()`, `startAsync()`, `getStatusAsync()`, or `stopAndUnloadAsync()`), and will also be called at regular intervals while the recording can record. Call `setProgressUpdateInterval()` to modify the interval with which `onRecordingStatusUpdate` is called while the recording can record.

Parameters

- **onRecordingStatusUpdate (function)** -- A function taking a single parameter `status` (a dictionary, described in `getStatusAsync`).
- `recordingInstance.setProgressUpdateInterval(millis)`

Sets the interval with which `onRecordingStatusUpdate` is called while the recording can record. See `setOnRecordingStatusUpdate` for details. This value defaults to 500 milliseconds.

Parameters

- **millis (number)** -- The new interval between calls of `onRecordingStatusUpdate`.
- `recordingInstance.prepareToRecordAsync(options)`

Loads the recorder into memory and prepares it for recording. This must be called before calling `startAsync()`. This method can only be called if the `Recording` instance has never yet been prepared.

Parameters

- **options (RecordingOptions)** -- Options for the recording, including sample rate, bitrate, channels, format, encoder, and extension. If no options are passed to `prepareToRecordAsync()`, the recorder will be created with options `Audio.RECORDING_OPTIONS_PRESET_LOW_QUALITY`. See below for details on `RecordingOptions`.

Returns

A `Promise` that is fulfilled when the recorder is loaded and prepared, or rejects if this failed. If another `Recording` exists in your experience that is currently prepared to record, the `Promise` will reject. If the `RecordingOptions` provided are invalid, the `Promise` will also reject. The promise is resolved with the `status` of the recording (see `getStatusAsync()` for details).

- `recordingInstance.isPreparedToRecord()`

Returns

A `boolean` that is true if and only if the `Recording` is prepared to record.

- `recordingInstance.startAsync()`

Begins recording. This method can only be called if the `Recording` has been prepared.

Returns

A `Promise` that is fulfilled when recording has begun, or rejects if recording could not start. The promise is resolved with the `status` of the recording (see `getStatusAsync()` for details).

- `recordingInstance.pauseAsync()`

Pauses recording. This method can only be called if the `Recording` has been prepared.

NOTE: This is only available on Android API version 24 and later.

Returns

A `Promise` that is fulfilled when recording has paused, or rejects if recording could not be paused. If the Android API version is less than 24, the `Promise` will reject. The promise is resolved with the `status` of the recording (see `getStatusAsync()` for details).

- `recordingInstance.stopAndUnloadAsync()`

Stops the recording and deallocates the recorder from memory. This reverts the `Recording` instance to an unprepared state, and another `Recording` instance must be created in order to record again. This method can only be called if the `Recording` has been prepared.

Returns

A `Promise` that is fulfilled when recording has stopped, or rejects if recording could not be stopped. The promise is resolved with the `status` of the recording (see `getStatusAsync()` for details).

- `recordingInstance.getURI()`

Gets the local URI of the `Recording`. Note that this will only succeed once the `Recording` is prepared to record.

Returns

A `string` with the local URI of the `Recording`, or `null` if the `Recording` is not prepared to record.

- `recordingInstance.createNewLoadedSound()`

Creates and loads a new `Sound` object to play back the `Recording`. Note that this will only succeed once the `Recording` is done recording (once `stopAndUnloadAsync()` has been called).

Parameters

- **initialStatus (`PlaybackStatusToSet`)** -- The initial intended `PlaybackStatusToSet` of the sound, whose values will override the default initial playback status. This value defaults to `{}` if no parameter is passed. See the [AV documentation](#) for details on `PlaybackStatusToSet` and the default initial playback status.
- **onPlaybackStatusUpdate (`function`)** -- A function taking a single parameter `PlaybackStatus`. This value defaults to `null` if no parameter is passed. See the [AV documentation](#) for details on the functionality provided by `onPlaybackStatusUpdate`

Returns

A `Promise` that is rejected if creation failed, or fulfilled with the following dictionary if creation succeeded:

- `sound` : the newly created and loaded `Sound` object.
- `status` : the `PlaybackStatus` of the `Sound` object. See the [AV documentation](#) for further information.

RecordingOptions

The recording extension, sample rate, bitrate, channels, format, encoder, etc can be customized by passing a dictionary of options to `prepareToRecordAsync()`.

We provide the following preset options for convenience, as used in the example above. See below for the definitions of these presets.

- `Audio.RECORDING_OPTIONS_PRESET_HIGH_QUALITY`
- `Audio.RECORDING_OPTIONS_PRESET_LOW_QUALITY`

We also provide the ability to define your own custom recording options, but **we recommend you use the presets, as not all combinations of options will allow you to successfully `prepareToRecordAsync()`**. You will have to test your custom options on iOS and Android to make sure it's working. In the future, we will enumerate all possible valid combinations, but at this time, our goal is to make the basic use-case easy (with presets) and the advanced use-case possible (by exposing all the functionality available in native). As always, feel free to ping us on the forums or Slack with any questions.

In order to define your own custom recording options, you must provide a dictionary of the following key value pairs.

- `android` : a dictionary of key-value pairs for the Android platform. This key is required.
 - `extension` : the desired file extension. This key is required. Example valid values are `.3gp` and `.m4a`.
For more information, see the [Android docs for supported output formats](#).
 - `outputFormat` : the desired file format. This key is required. See the next section for an enumeration of all valid values of `outputFormat`.
 - `audioEncoder` : the desired audio encoder. This key is required. See the next section for an enumeration of all valid values of `audioEncoder`.
 - `sampleRate` : the desired sample rate. This key is optional. An example valid value is `44100`.
Note that the sampling rate depends on the format for the audio recording, as well as the capabilities of the platform. For instance, the sampling rate supported by AAC audio coding standard ranges from 8 to 96 kHz, the sampling rate supported by AMRNB is 8kHz, and the sampling rate supported by AMRWB is 16kHz. Please consult with the related audio coding standard for the supported audio sampling rate.
 - `numberOfChannels` : the desired number of channels. This key is optional. Example valid values are `1` and `2`.
Note that `prepareToRecordAsync()` may perform additional checks on the parameter to make sure whether the specified number of audio channels are applicable.
 - `bitRate` : the desired bit rate. This key is optional. An example valid value is `128000`.
Note that `prepareToRecordAsync()` may perform additional checks on the parameter to make sure whether the specified bit rate is applicable, and sometimes the passed bitRate will be clipped internally to ensure the audio recording can proceed smoothly based on the capabilities of the platform.
 - `maxFileSize` : the desired maximum file size in bytes, after which the recording will stop (but `stopAndUnloadAsync()` must still be called after this point). This key is optional. An example valid value is `65536`.
- `ios` : a dictionary of key-value pairs for the iOS platform
 - `extension` : the desired file extension. This key is required. An example valid value is `.caf`.
 - `outputFormat` : the desired file format. This key is optional. See the next section for an enumeration of all valid values of `outputFormat`.
 - `audioQuality` : the desired audio quality. This key is required. See the next section for an enumeration of all valid values of `audioQuality`.
 - `sampleRate` : the desired sample rate. This key is required. An example valid value is `44100`.

- `numberOfChannels` : the desired number of channels. This key is required. Example valid values are `1` and `2`.
- `bitRate` : the desired bit rate. This key is required. An example valid value is `128000`.
- `bitRateStrategy` : the desired bit rate strategy. This key is optional. See the next section for an enumeration of all valid values of `bitRateStrategy`.
- `bitDepthHint` : the desired bit depth hint. This key is optional. An example valid value is `16`.
- `linearPCMBitDepth` : the desired PCM bit depth. This key is optional. An example valid value is `16`.
- `linearPCMIsBigEndian` : a boolean describing if the PCM data should be formatted in big endian. This key is optional.
- `linearPCMIsFloat` : a boolean describing if the PCM data should be encoded in floating point or integral values. This key is optional.

Following is an enumeration of all of the valid values for certain `RecordingOptions` keys.

Note Not all of the iOS formats included in this list of constants are currently supported by iOS, in spite of appearing in the Apple source code. For an accurate list of formats supported by iOS, see [Core Audio Codecs](#) and [iPhone Audio File Formats](#).

- `android` :
 - `outputFormat` :
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_DEFAULT`
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_THREE_GPP`
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_MPEG_4`
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_AMR_NB`
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_AMR_WB`
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_AAC_ADIF`
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_AAC_ADTS`
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_RTP_AVP`
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_MPEG2TS`
 - `Audio.RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_WEBM`
 - `audioEncoder` :
 - `Audio.RECORDING_OPTION_ANDROID_AUDIO_ENCODER_DEFAULT`
 - `Audio.RECORDING_OPTION_ANDROID_AUDIO_ENCODER_AMR_NB`
 - `Audio.RECORDING_OPTION_ANDROID_AUDIO_ENCODER_AMR_WB`
 - `Audio.RECORDING_OPTION_ANDROID_AUDIO_ENCODER_AAC`
 - `Audio.RECORDING_OPTION_ANDROID_AUDIO_ENCODER_HE_AAC`
 - `Audio.RECORDING_OPTION_ANDROID_AUDIO_ENCODER_AAC_ELD`

- `ios` :
 - `outputFormat` :
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_LINEARPCM`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_AC3`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_60958AC3`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_APPLEIMA4`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4AAC`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4CELP`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4HVXC`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4TWINVQ`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MACE3`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MACE6`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_ULAW`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_ALAW`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_QDESIGN`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_QDESIGN2`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_QUALCOMM`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEGLAYER1`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEGLAYER2`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEGLAYER3`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_APPLELOSSLESS`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4AAC_HE`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4AAC_LD`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4AAC_ELD`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4AAC_ELD_SBR`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4AAC_ELD_V2`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4AAC_HE_V2`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MPEG4AAC_SPATIAL`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_AMR`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_AMR_WB`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_AUDIBLE`
 - `Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_ILBC`

- Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_DVIINTELIMA
- Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_MICROSOFTGSM
- Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_AES3
- Audio.RECORDING_OPTION_IOS_OUTPUT_FORMAT_ENHANCEDAC3
- audioQuality :
 - Audio.RECORDING_OPTION_IOS_AUDIO_QUALITY_MIN
 - Audio.RECORDING_OPTION_IOS_AUDIO_QUALITY_LOW
 - Audio.RECORDING_OPTION_IOS_AUDIO_QUALITY_MEDIUM
 - Audio.RECORDING_OPTION_IOS_AUDIO_QUALITY_HIGH
 - Audio.RECORDING_OPTION_IOS_AUDIO_QUALITY_MAX
- bitRateStrategy :
 - Audio.RECORDING_OPTION_IOS_BIT_RATE_STRATEGY_CONSTANT
 - Audio.RECORDING_OPTION_IOS_BIT_RATE_STRATEGY_LONG_TERM_AVERAGE
 - Audio.RECORDING_OPTION_IOS_BIT_RATE_STRATEGY_VARIABLE_CONSTRAINED
 - Audio.RECORDING_OPTION_IOS_BIT_RATE_STRATEGY_VARIABLE

For reference, following are the definitions of the two preset examples of `RecordingOptions`, as implemented in the Audio SDK:

```
export const RECORDING_OPTIONS_PRESET_HIGH_QUALITY: RecordingOptions = {
  android: {
    extension: '.m4a',
    outputFormat: RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_MPEG_4,
    audioEncoder: RECORDING_OPTION_ANDROID_AUDIO_ENCODER_AAC,
    sampleRate: 44100,
    numberOfChannels: 2,
    bitRate: 128000,
  },
  ios: {
    extension: '.caf',
    audioQuality: RECORDING_OPTION_IOS_AUDIO_QUALITY_MAX,
    sampleRate: 44100,
    numberOfChannels: 2,
    bitRate: 128000,
    linearPCMBitDepth: 16,
    linearPCMIsBigEndian: false,
    linearPCMIsFloat: false,
  },
};

export const RECORDING_OPTIONS_PRESET_LOW_QUALITY: RecordingOptions = {
  android: {
    extension: '.3gp',
    outputFormat: RECORDING_OPTION_ANDROID_OUTPUT_FORMAT_THREE_GPP,
    audioEncoder: RECORDING_OPTION_ANDROID_AUDIO_ENCODER_AMR_NB,
    sampleRate: 44100,
    numberOfChannels: 2,
    bitRate: 128000,
  },
  ios: {
```

```
        extension: '.caf',
        audioQuality: RECORDING_OPTION_IOS_AUDIO_QUALITY_MIN,
        sampleRate: 44100,
        numberOfChannels: 2,
        bitRate: 128000,
        linearPCMBitDepth: 16,
        linearPCMIsBigEndian: false,
        linearPCMIsFloat: false,
    },
};
```

#

AuthSession

`AuthSession` is the easiest way to add web browser based authentication (for example, browser-based OAuth flows) to your app, built on top of [WebBrowser](#). If you would like to understand how it does this, read this document from top to bottom. If you just want to use it, jump to the [Example](#).

Installation

This API is pre-installed in [managed](#) apps. It is not available for [bare](#) React Native apps.

How web browser based authentication flows work

The typical flow for browser-based authentication in mobile apps is as follows:

- **Initiation:** the user presses a sign in button
- **Open web browser:** the app opens up a web browser to the authentication provider sign in page. The url that is opened for the sign in page usually includes information to identify the app, and a URL to redirect to on success. *Note: the web browser should share cookies with your system web browser so that users do not need to sign in again if they are already authenticated on the system browser -- Expo's [WebBrowser API](#) takes care of this.*
- **Authentication provider redirects:** upon successful authentication, the authentication provider should redirect back to the application by redirecting to URL provided by the app in the query parameters on the sign in page ([read more about how linking works in mobile apps](#)), *provided that the URL is in the whitelist of allowed redirect URLs*. Whitelisting redirect URLs is important to prevent malicious actors from pretending to be your application. The redirect includes data in the URL (such as user id and token), either in the location hash, query parameters, or both.
- **App handles redirect:** the redirect is handled by the app and data is parsed from the redirect URL.

What `AuthSession` does for you

It reduces boilerplate

`AuthSession` handles most of the app-side responsibilities for you:

- It opens the sign in URL for your authentication provider (`authUrl`, you must provide it) in a web browser that shares cookies with your system browser.
- It handles success redirects and extracts all of the data encoded in the URL.
- It handles failures and provides information to you about what went wrong.

It makes redirect URL whitelists easier to manage for development and working in teams

Additionally, `AuthSession` **simplifies setting up authorized redirect URLs** by using an Expo service that sits between you and your authentication provider ([read Security Concerns for caveats](#)). This is particularly valuable with Expo because your app can live at various URLs. In development, you can have a tunnel URL, a lan URL, and a localhost URL. The tunnel URL on your machine for the same app will be different from a co-worker's

machine. When you publish your app, that will be another URL that you need to whitelist. If you have multiple environments that you publish to, each of those will also need to be whitelisted. `AuthSession` gets around this by only having you whitelist one URL with your authentication provider: `https://auth.expo.io/@your-username/your-app-slug`. When authentication is successful, your authentication provider will redirect to that Expo Auth URL, and then the Expo Auth service will redirect back to your application. If the URL that the auth service is redirecting back to does not match the published URL for the app or the standalone app scheme (eg: `exp://expo.io/@your-username/your-app-slug`, or `yourscheme://`), then it will show a warning page before asking the user to sign in. This means that in development you will see this warning page when you sign in, a small price to pay for the convenience.

How does this work? When you open an authentication session with `AuthSession`, it first visits `https://auth.expo.io/@your-username/your-app-slug/start` and passes in the `authUrl` and `returnUrl` (the URL to redirect back to your application) in the query parameters. The Expo Auth service saves away the `returnUrl` (and if it is not a published URL or your registered custom theme, shows a warning page) and then sends the user off to the `authUrl`. When the authentication provider redirects back to `https://auth.expo.io/@your-username/your-app-slug` on success, the Expo Auth services redirects back to the `returnUrl` that was provided on initiating the authentication flow.

Security considerations

If you are authenticating with a popular social provider, when you are ready to ship to production you should be sure that you do not directly request the access token for the user. Instead, most providers give an option to request a one-time code that can be combined with a secret key to request an access token. For an example of this flow, [see the *Confirming Identity* section in the Facebook Login documentation](#).

Never put any secret keys inside of your app, there is no secure way to do this! Instead, you should store your secret key(s) on a server and expose an endpoint that makes API calls for your client and passes the data back.

Usage in standalone apps

In order to be able to deep link back into your app, you will need to set a `scheme` in your project `app.json`, and then build your standalone app (it can't be updated with an OTA update). If you do not include a scheme, the authentication flow will complete but it will be unable to pass the information back into your application and the user will have to manually exit the authentication modal.

Example

```
import React from 'react';
import { Button, StyleSheet, Text, View } from 'react-native';
import { AuthSession } from 'expo';

/* @info Replace <strong>'YOUR_APP_ID'</strong> with your application id from <a href='https://developers.facebook.com' target='_blank'>developers.facebook.com</a> */
const FB_APP_ID = 'YOUR_APP_ID';
/* @end */

export default class App extends React.Component {
  state = {
    result: null,
  };
}
```

```

render() {
  return (
    <View style={styles.container}>
      <Button title="Open FB Auth" onPress={this._handlePressAsync} />
      /* @info In this example, show the authentication result after success. In a real application, this would be
      a weird thing to do, instead you would use this data to match the user with a user in your application and sign them
      in. */
      {this.state.result ? (
        <Text>{JSON.stringify(this.state.result)}</Text>
      ) : null}
      /* @end */
    </View>
  );
}

_handlePressAsync = async () => {
  let redirectUrl = /* @info <strong>AuthSession.getRedirectUrl()</strong> gets the appropriate URL on <em>https://
  auth.expo.io</em> to redirect back to your application. Read more about it below. */ AuthSession.getRedirectUrl();/*
  @end */

  let result = /* @info <strong>AuthSession.startAsync</strong> returns a Promise that resolves to an object with t
  he information that was passed back from your authentication provider, for example the user id. */ await AuthSession.
  startAsync/* @end */({

    /* @info authUrl is a required parameter -- it is the URL that points to the sign in page for your chosen authe
    ntication service (in this case, we are using Facebook sign in) */ authUrl:/* @end */

    /* @info The particular URL and the format you need to use for this depend on your authentication service. Fo
    r Facebook, information was found <a href='https://developers.facebook.com/docs/facebook-login/manually-build-a-login
    -flow/' target='_blank'>here</a>.*/`https://www.facebook.com/v2.8/dialog/oauth?response_type=token`/* @end */ +
    `&client_id=${FB_APP_ID}` +
    `&redirect_uri=${/* @info Be sure to call <a href='https://developer.mozilla.org/en-US/docs/Web/JavaScript/Re
    ference/Global_Objects/encodeURIComponent'>encodeURIComponent</a> on any query parameters, or use a library such as <a
    href='https://github.com/ljharb/qs'>qs</a>. */encodeURIComponent(redirectUrl)/* @end */}`,
  });

  this.setState({ result });
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
});

```

API

```
import { AuthSession } from 'expo';
```

AuthSession.startAsync(options)

Initiate an authentication session with the given options. Only one `AuthSession` can be active at any given time in your application; if you attempt to open a second session while one is still in progress, the second session will return a value to indicate that `AuthSession` is locked.

Arguments

- **options (object)** --

A map of options:

- **authUrl (string)** -- **Required**. The URL that points to the sign in page that you would like to open the user to.
- **returnUrl (string)** -- The URL to return to the application. Defaults to `${Constants.linkUrl}expo-auth-session`, for example `exp://expo.io/@yourname/your-app-slug+expo-auth-session`.

Returns

Returns a Promise that resolves to a result object of the following form:

- If the user cancelled the authentication session by closing the browser, the result is `{ type: 'cancel' }` .
- If the authentication is dismissed manually with `AuthSession.dismiss()`, the result is `{ type: 'dismissed' }` .
- If the authentication flow is successful, the result is `{type: 'success', params: Object, event: Object }`
- If the authentication flow returns an error, the result is `{type: 'error', params: Object, errorCode: string, event: Object }`
- If you call `AuthSession.startAsync` more than once before the first call has returned, the result is `{type: 'locked'}`, because only one `AuthSession` can be in progress at any time.

AuthSession.dismiss()

Cancels an active `AuthSession` if there is one. No return value, but if there is an active `AuthSession` then the Promise returned by the `AuthSession.startAsync` that initiated it resolves to `{ type: 'dismissed' }` .

AuthSession.getRedirectUrl()

Get the URL that your authentication provider needs to redirect to. For example: `https://auth.expo.io/@your-username/your-app-slug` .

Advanced usage

Filtering out AuthSession events in Linking handlers

There are many reasons why you might want to handle inbound links into your app, such as push notifications or just regular deep linking (you can read more about this in the [Linking guide](#)); authentication redirects are only one type of deep link, and `AuthSession` handles these particular links for you. In your own `Linking.addEventListener` handlers, you can filter out deep links that are handled by `AuthSession` by checking if the URL includes the `+expo-auth-session` String -- if it does, you can ignore it. This works because `AuthSession` adds `+expo-auth-session` to the default `returnUrl`; however, if you provide your own `returnUrl`, you may want to consider adding a similar identifier to enable you to filter out `AuthSession` events from other handlers.

```
#
```

AV

The `Audio.Sound` objects and `Video` components share a unified imperative API for media playback.

Note that for `Video`, all of these operations are also available via props on the component, but we recommend using this imperative playback API for most applications where finer control over the state of the video playback is needed.

Try the [playlist example app](#) (source code is [on GitHub](#)) to see an example usage of the playback API for both `Audio.Sound` and `Video`.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Audio, Video } from 'expo';  
  
// in bare apps:  
import { Audio, Video } from 'expo-av';
```

Usage

Construction and obtaining a reference

In this page, we reference operations on `playbackObject`s. Here is an example of obtaining access to the reference for both sound and video:

Example: `Audio.Sound`

```
const playbackObject = new Audio.Sound();  
// OR  
const playbackObject = await Audio.Sound.createAsync(  
  { uri: 'http://foo/bar.mp3' },  
  { shouldPlay: true }  
);  
...
```

See the [audio documentation](#) for further information on `Audio.Sound.createAsync()`.

Example: `Video`

```
...  
_handleVideoRef = component => {  
  const playbackObject = component;  
  ...
```

```

}

...

render() {
  return (
    ...
    <Video
      ref={this._handleVideoRef}
      ...
    />
    ...
  )
}
...

```

Playback API

On the `playbackObject` reference, the following API is provided:

- `playbackObject.loadAsync(source, initialStatus = {}, downloadFirst = true)`

Loads the media from `source` into memory and prepares it for playing. This must be called before calling `setStatusAsync()` or any of the convenience set status methods. This method can only be called if the `playbackObject` is in an unloaded state.

Parameters

- **source (object / number / Asset)** -- The source of the media. The following forms are supported:

- A dictionary of the form `{ uri: string, headers?: { [string]: string }, overrideFileExtensionAndroid?: string }` with a network URL pointing to a media file on the web, an optional `headers` object passed in a network request to the `uri` and an optional Android-specific `overrideFileExtensionAndroid` `String` overriding extension inferred from the URL.

The `overrideFileExtensionAndroid` property may come in handy if the player receives an URL like `example.com/play` which redirects to `example.com/player.m3u8`. Setting this property to `m3u8` would allow the Android player to properly infer the content type of the media and use proper media file reader.

- `require('path/to/file')` for a media file asset in the source code directory.
- An `Asset` object for a media file asset.

The [iOS developer documentation](#) lists the audio and video formats supported on iOS.

There are two sets of audio and video formats supported on Android: [formats supported by ExoPlayer](#) and [formats supported by Android's MediaPlayer](#). Expo uses ExoPlayer implementation by default; to use `MediaPlayer`, add `androidImplementation: 'MediaPlayer'` to the initial status of the AV object.

- **initialStatus (`PlaybackStatusToSet`)** -- The initial intended `PlaybackStatusToSet` of the `playbackObject`, whose values will override the default initial playback status. This value defaults to `{}` if no parameter is passed. See below for details on `PlaybackStatusToSet` and the default initial playback status.
- **downloadFirst (boolean)** -- If set to true, the system will attempt to download the resource to the device before loading. This value defaults to `true`. Note that at the moment, this will only work for `source`s of the form `require('path/to/file')` or `Asset` objects.

Returns

A `Promise` that is fulfilled with the `PlaybackStatus` of the `playbackObject` once it is loaded, or rejects if loading failed. The `Promise` will also reject if the `playbackObject` was already loaded. See below for details on `PlaybackStatus`.

- `playbackObject.unloadAsync()`

Unloads the media from memory. `loadAsync()` must be called again in order to be able to play the media.

Returns

A `Promise` that is fulfilled with the `PlaybackStatus` of the `playbackObject` once it is unloaded, or rejects if unloading failed. See below for details on `PlaybackStatus`.

- `playbackObject.getStatusAsync()`

Gets the `PlaybackStatus` of the `playbackObject`.

Returns

A `Promise` that is fulfilled with the `PlaybackStatus` of the `playbackObject`. See below for details on `PlaybackStatus`.

- `playbackObject.setOnPlaybackStatusUpdate(onPlaybackStatusUpdate)`

Sets a function to be called regularly with the `PlaybackStatus` of the `playbackObject`. See below for details on `PlaybackStatus` and an example use case of this function.

`onPlaybackStatusUpdate` will be called whenever a call to the API for this `playbackObject` completes (such as `setStatusAsync()`, `getStatusAsync()`, or `unloadAsync()`), and will also be called at regular intervals while the media is in the loaded state. Set `progressUpdateIntervalMillis` via `setStatusAsync()` or `setProgressUpdateIntervalAsync()` to modify the interval with which `onPlaybackStatusUpdate` is called while loaded.

Parameters

- **onPlaybackStatusUpdate (function)** -- A function taking a single parameter `PlaybackStatus` (a dictionary, described below).
 - `playbackObject.replayAsync(statusToSet)`
- Replays the item. When using `playFromPositionAsync(0)` the item is seeked to the position at `0 ms`. On iOS this method uses internal implementation of the player and is able to play the item from the beginning immediately.

Parameters

- **statusToSet (PlaybackStatusToSet)** -- The new `PlaybackStatusToSet` of the `playbackObject`, whose values will override the current playback status. See below for details on `PlaybackStatusToSet`. `positionMillis` and `shouldPlay` properties will be overriden with respectively `0` and `true`.

Returns

A `Promise` that is fulfilled with the `PlaybackStatus` of the `playbackObject` once the new status has been set successfully, or rejects if setting the new status failed. See below for details on `PlaybackStatus`.

- `playbackObject.setStatusAsync(statusToSet)`

Sets a new `PlaybackStatusToSet` on the `playbackObject`. This method can only be called if the media has been loaded.

Parameters

- `statusToSet (PlaybackStatusToSet)` -- The new `PlaybackStatusToSet` of the `playbackObject`, whose values will override the current playback status. See below for details on `PlaybackStatusToSet`.

Returns

A `Promise` that is fulfilled with the `PlaybackStatus` of the `playbackObject` once the new status has been set successfully, or rejects if setting the new status failed. See below for details on `PlaybackStatus`.

The following convenience methods built on top of `setStatusAsync()` are also provided. Each has the same return value as `setStatusAsync()`.

- `playbackObject.playAsync()`

This is equivalent to `playbackObject.setStatusAsync({ shouldPlay: true })`.

Playback may not start immediately after calling this function for reasons such as buffering. Make sure to update your UI based on the `isPlaying` and `isBuffering` properties of the `PlaybackStatus` (described below).

- `playbackObject.playFromPositionAsync(millis)`

This is equivalent to `playbackObject.setStatusAsync({ shouldPlay: true, positionMillis: millis })`.

Playback may not start immediately after calling this function for reasons such as buffering. Make sure to update your UI based on the `isPlaying` and `isBuffering` properties of the `PlaybackStatus` (described below).

- `playbackObject.playFromPositionAsync(millis, { toleranceMillisBefore, toleranceMillisAfter })`

This is equivalent to `playbackObject.setStatusAsync({ shouldPlay: true, positionMillis: millis, seekMillisToleranceBefore: toleranceMillisBefore, seekMillisToleranceAfter: toleranceMillisAfter })`. The tolerances are used only on iOS ([more details](#)).

Playback may not start immediately after calling this function for reasons such as buffering. Make sure to update your UI based on the `isPlaying` and `isBuffering` properties of the `PlaybackStatus` (described below).

Parameters

- `millis (number)` -- The desired position of playback in milliseconds.

- `playbackObject.pauseAsync()`

This is equivalent to `playbackObject.setStatusAsync({ shouldPlay: false })`.

- `playbackObject.stopAsync()`

This is equivalent to `playbackObject.setStatusAsync({ shouldPlay: false, positionMillis: 0 })`.

- `playbackObject.setPositionAsync(millis)`

This is equivalent to `playbackObject.setStatusAsync({ positionMillis: millis })`.

- `playbackObject.setPositionAsync(millis, { toleranceMillisBefore, toleranceMillisAfter })`

This is equivalent to `playbackObject.setStatusAsync({ positionMillis: millis, seekMillisToleranceBefore: toleranceMillisBefore, seekMillisToleranceAfter: toleranceMillisAfter })`. The tolerances are used only on iOS ([more details](#)).

Parameters

- **millis (number)** -- The desired position of playback in milliseconds.
- `playbackObject.setRateAsync(value, shouldCorrectPitch, pitchCorrectionQuality)`

This is equivalent to `playbackObject.setStatusAsync({ rate: value, shouldCorrectPitch: shouldCorrectPitch, pitchCorrectionQuality: pitchCorrectionQuality })`.

Parameters

- **value (number)** -- The desired playback rate of the media. This value must be between `0.0` and `32.0`. Only available on Android API version 23 and later and iOS.
- `playbackObject.setVolumeAsync(value)`

This is equivalent to `playbackObject.setStatusAsync({ volume: value })`.

Parameters

- **value (number)** -- A number between `0.0` (silence) and `1.0` (maximum volume).
- `playbackObject.setIsMutedAsync(value)`

This is equivalent to `playbackObject.setStatusAsync({ isMuted: value })`.

Parameters

- **value (boolean)** -- A boolean describing if the audio of this media should be muted.
- `playbackObject.setIsLoopingAsync(value)`

This is equivalent to `playbackObject.setStatusAsync({ isLooping: value })`.

Parameters

- **value (boolean)** -- A boolean describing if the media should play once (`false`) or loop indefinitely (`true`).
- `playbackObject.setProgressUpdateIntervalAsync(millis)`

This is equivalent to `playbackObject.setStatusAsync({ progressUpdateIntervalMillis: millis })`.

Parameters

- **millis (number)** -- The new minimum interval in milliseconds between calls of `onPlaybackStatusUpdate`. See `setOnPlaybackStatusUpdate()` for details.

Playback Status

Most of the preceding API calls revolve around passing or returning the `status` of the `playbackObject`.

- `PlaybackStatus`

This is the structure returned from all playback API calls and describes the state of the `playbackObject` at that point in time. It is a dictionary with the following key-value pairs.

If the `playbackObject` is not loaded, it will contain the following key-value pairs:

- `isLoading` : a boolean set to `false`.
- `error` : a string only present if the `playbackObject` just encountered a fatal error and forced unload.

Otherwise, it contains all of the following key-value pairs:

- `isLoading` : a boolean set to `true`.
- `uri` : the location of the media source.
- `progressUpdateIntervalMillis` : the minimum interval in milliseconds between calls of `onPlaybackStatusUpdate`. See `setOnPlaybackStatusUpdate()` for details.
- `durationMillis` : the duration of the media in milliseconds. This is only present if the media has a duration (note that in some cases, a media file's duration is readable on Android, but not on iOS).
- `positionMillis` : the current position of playback in milliseconds.
- `playableDurationMillis` : the position until which the media has been buffered into memory. Like `durationMillis`, this is only present in some cases.
- `shouldPlay` : a boolean describing if the media is supposed to play.
- `isPlaying` : a boolean describing if the media is currently playing.
- `isBuffering` : a boolean describing if the media is currently buffering.
- `rate` : the current rate of the media.
- `shouldCorrectPitch` : a boolean describing if we are correcting the pitch for a changed rate.
- `volume` : the current volume of the audio for this media.
- `isMuted` : a boolean describing if the audio of this media is currently muted.
- `isLooping` : a boolean describing if the media is currently looping.
- `didJustFinish` : a boolean describing if the media just played to completion at the time that this status was received. When the media plays to completion, the function passed in `setOnPlaybackStatusUpdate()` is called exactly once with `didJustFinish` set to `true`. `didJustFinish` is never `true` in any other case.

- `PlaybackStatusToSet`

This is the structure passed to `setStatusAsync()` to modify the state of the `playbackObject`. It is a dictionary with the following key-value pairs, all of which are optional.

- `progressUpdateIntervalMillis` : the new minimum interval in milliseconds between calls of `onPlaybackStatusUpdate`. See `setOnPlaybackStatusUpdate()` for details.
- `positionMillis` : the desired position of playback in milliseconds.
- `seekMillisToleranceBefore` : the tolerance in milliseconds with which seek will be applied to the player.
[iOS only, details]
- `seekMillisToleranceAfter` : the tolerance in milliseconds with which seek will be applied to the player.
[iOS only, details]
- `shouldPlay` : a boolean describing if the media is supposed to play. Playback may not start immediately after setting this value for reasons such as buffering. Make sure to update your UI based on the `isPlaying` and `isBuffering` properties of the `PlaybackStatus`.
- `rate` : the desired playback rate of the media. This value must be between `0.0` and `32.0`. Only available on Android API version 23 and later and iOS.
- `shouldCorrectPitch` : a boolean describing if we should correct the pitch for a changed rate. If set to `true`, the pitch of the audio will be corrected (so a rate different than `1.0` will timestretch the audio).

- `volume` : the desired volume of the audio for this media. This value must be between `0.0` (silence) and `1.0` (maximum volume).
- `isMuted` : a boolean describing if the audio of this media should be muted.
- `isLooping` : a boolean describing if the media should play once (`false`) or loop indefinitely (`true`).
- `androidImplementation` : underlying implementation to use (when set to `MediaPlayer` it uses Android's `MediaPlayer`, uses `ExoPlayer` otherwise). You may need to use this property if you're trying to play an item unsupported by ExoPlayer ([formats supported by ExoPlayer](#), [formats supported by Android's MediaPlayer](#)). Note that setting this property takes effect only when the AV object is just being created (toggling its value later will do nothing). *[Android only]*

Note that a `rate` different than `1.0` is currently only available on Android API version 23 and later and iOS.

Note that `volume` and `isMuted` only affect the audio of this `playbackObject` and do NOT affect the system volume.

Default initial `PlaybackStatusToSet`

The default initial `PlaybackStatusToSet` of all `Audio.Sound` objects and `Video` components is as follows:

```
{
  progressUpdateIntervalMillis: 500,
  positionMillis: 0,
  shouldPlay: false,
  rate: 1.0,
  shouldCorrectPitch: false,
  volume: 1.0,
  isMuted: false,
  isLooping: false,
}
```

This default initial status can be overwritten by setting the optional `initialStatus` in `loadAsync()` or `Audio.Sound.createAsync()`.

What is seek tolerance and why would I want to use it [iOS only]

When asked to seek an A/V item, native player in iOS sometimes may seek to a slightly different time. This technique, mentioned in [Apple documentation](#), is used to shorten the time of the `seekTo` call (the player may decide to play immediately from a different time than requested, instead of decoding the exact requested part and playing it with the decoding delay).

If you matter about the precision more than about the delay, you can specify the tolerance with which the player will seek according to your needs.

Example usage

Example: `setOnPlaybackStatusUpdate()`

```
_onPlaybackStatusUpdate = playbackStatus => {
  if (!playbackStatus.isLoaded) {
    // Update your UI for the unloaded state
    if (playbackStatus.error) {
      console.log(`Encountered a fatal error during playback: ${playbackStatus.error}`);
      // Send Expo team the error on Slack or the forums so we can help you debug!
    }
  }
}
```

```

} else {
    // Update your UI for the loaded state

    if (playbackStatus.isPlaying) {
        // Update your UI for the playing state
    } else {
        // Update your UI for the paused state
    }

    if (playbackStatus.isBuffering) {
        // Update your UI for the buffering state
    }

    if (playbackStatus.didJustFinish && !playbackStatus.isLooping) {
        // The player has just finished playing and will stop. Maybe you want to play something else?
    }

    ...
}

};

... // Load the playbackObject and obtain the reference.
playbackObject.setOnPlaybackStatusUpdate(this._onPlaybackStatusUpdate);
...

```

Example: Loop media exactly 20 times

```

const N = 20;
...

_onPlaybackStatusUpdate = (playbackStatus) => {
    if (playbackStatus.didJustFinish) {
        if (this.state.numberOfLoops == N - 1) {
            playbackObject.setIsLooping(false);
        }
        this.setState({ numberOfLoops: this.state.numberOfLoops + 1 });
    }
};

...

this.setState({ numberOfLoops: 0 });
... // Load the playbackObject and obtain the reference.
playbackObject.setOnPlaybackStatusUpdate(this._onPlaybackStatusUpdate);
playbackObject.setIsLooping(true);
...

```

#

BackgroundFetch

Provides API to perform [background fetch](#) tasks. This module uses [TaskManager](#) Native API under the hood. In order to use `BackgroundFetch` API in standalone and detached apps on iOS, your app has to include background mode in the `Info.plist` file. See [background tasks configuration guide](#) for more details.

Installation

This API is pre-installed in [managed](#) apps. It is not yet available for [bare](#) React Native apps.

API

```
import { BackgroundFetch } from 'expo';
```

BackgroundFetch.getStatusAsync()

Gets a status of background fetch.

Returns

Returns a promise resolving to one of these values:

- `BackgroundFetch.Status.Restricted` — Background updates are unavailable and the user cannot enable them again. This status can occur when, for example, parental controls are in effect for the current user.
- `BackgroundFetch.Status.Denied` - The user explicitly disabled background behavior for this app or for the whole system.
- `BackgroundFetch.Status.Available` - Background updates are available for the app.

BackgroundFetch.registerTaskAsync(taskName)

Registers background fetch task with given name. Registered tasks are saved in persistent storage and restored once the app is initialized.

Arguments

- **taskName (string)** -- Name of the task to register. The task needs to be defined first - see [TaskManager.defineTask](#) for more details.

Returns

Returns a promise that resolves once the task is registered and rejects in case of any errors.

Task parameters

Background fetch task receives no data, but your task should return a value that best describes the results of your background fetch work.

- `BackgroundFetch.Result.NoData` - There was no new data to download.
- `BackgroundFetch.Result.NewData` - New data was successfully downloaded.
- `BackgroundFetch.Result.Failed` - An attempt to download data was made but that attempt failed.

This return value is to let iOS know what the result of your background fetch was, so the platform can better schedule future background fetches. Also, your app has up to 30 seconds to perform the task, otherwise your app will be terminated and future background fetches may be delayed.

```
import { BackgroundFetch, TaskManager } from 'expo';

TaskManager.defineTask(YOUR_TASK_NAME, () => {
  try {
    const receivedNewData = // do your background fetch here
    return receivedNewData ? BackgroundFetch.Result.NewData : BackgroundFetch.Result.NoData;
  } catch (error) {
    return BackgroundFetch.Result.Failed;
  }
});
```

BackgroundFetch.unregisterTaskAsync(taskName)

Unregisters background fetch task, so the application will no longer be executing this task.

Arguments

- **taskName (string)** -- Name of the task to unregister.

Returns

A promise resolving when the task is fully unregistered.

BackgroundFetch.setMinimumIntervalAsync(minimumInterval)

Sets the minimum number of seconds that must elapse before another background fetch can be initiated. This value is advisory only and does not indicate the exact amount of time expected between fetch operations.

It is a global value which means that it can overwrite settings from another application opened through Expo Client.

Arguments

- **minimumInterval (number)** -- Number of seconds that must elapse before another background fetch can be called.

Returns

A promise resolving once the minimum interval is set.

BarCodeScanner

A React component that renders a viewfinder for the device's either front or back camera viewfinder and will scan bar codes that show up in the frame.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Supported formats

Bar code format	iOS	Android
aztec	Yes	Yes
codabar	No	Yes
code39	Yes	Yes
code93	Yes	Yes
code128	Yes	Yes
code138	Yes	No
code39mod43	Yes	No
datamatrix	Yes	Yes
ean13	Yes	Yes
ean8	Yes	Yes
interleaved2of5	Yes	No
itf14	Yes*	Yes
maxicode	No	Yes
pdf417	Yes	Yes
rss14	No	Yes
rssexpanded	No	Yes
upc_a	No	Yes
upc_e	Yes	Yes
upc_ean	No	Yes
qr	Yes	Yes

- sometimes when an ITF-14 barcode is recognized it's type is set to `interleaved2of5` .

Usage

You must request permission to access the user's camera before attempting to get it. To do this, you will want to use the [Permissions API](#). You can see this in practice in the following example.

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { BarCodeScanner, Permissions } from 'expo';

export default class BarcodeScannerExample extends React.Component {
  state = {
    hasCameraPermission: null,
  }

  async componentDidMount() {
    /* @info Before we can use the BarCodeScanner we need to ask the user for permission to access their camera. <a href='permissions.html'>Read more about Permissions.</a> */
    const { status } = await Permissions.askAsync(Permissions.CAMERA);
    this.setState({ hasCameraPermission: status === 'granted' });
    /* @end */
  }

  render() {
    const { hasCameraPermission } = this.state;

    if (hasCameraPermission === null) {
      return <Text>Requesting for camera permission</Text>;
    }
    if (hasCameraPermission === false) {
      return <Text>No access to camera</Text>;
    }
    return (
      <View style={{ flex: 1 }}>
        <BarCodeScanner
          onBarcodeScanned={this.handleBarcodeScanned}
          style={StyleSheet.absoluteFill}
        />
      </View>
    );
  }

  handleBarcodeScanned = ({ type, data }) => {
    alert(`Bar code with type ${type} and data ${data} has been scanned!`);
  }
}
```

Try this example on Snack.

API

```
// in managed apps:
import { BarCodeScanner } from 'expo';

// in bare apps:
import { BarCodeScanner } from 'expo-barcode-scanner';
```

Props

- **type (string)** -- Camera facing. Use one of `BarCodeScanner.Constants.Type`. Use either `Type.front` or `Type.back`. Same as `Camera.Constants.Type`. Default: `Type.back`.

- **barCodeTypes (Array!)** -- An array of bar code types. Usage: `BarCodeScanner.Constants.BarCodeType.<codeType>` where `codeType` is one of the listed above. Default: all supported bar code types. For example: `barCodeTypes=[[BarCodeScanner.Constants.BarCodeType.qr]]`
- **onBarcodeScanned (function)** -- A callback that is invoked when a bar code has been successfully scanned. The callback is provided with an object of the shape `{ type: BarCodeScanner.Constants.BarCodeType, data: string }`, where the type refers to the bar code type that was scanned and the data is the information encoded in the bar code (in this case of QR codes, this is often a URL).

Methods

BarCodeScanner.scanFromURLAsync(url, barCodeTypes)

Scan bar codes from the image given by the URL.

Arguments

- **url (string)** -- URL to get the image from.
- **barCodeTypes (Array!)** -- (as in prop) An array of bar code types. Default: all supported bar code types.
 - Note: Only QR codes are supported on iOS.

Returns

A possibly empty array of objects of the shape `{ type: BarCodeScanner.Constants.BarCodeType, data: string }`, where the type refers to the bar code type that was scanned and the data is the information encoded in the bar code.

Barometer

Access the device barometer sensor to respond to changes in air pressure. `pressure` is measured in `hectopascals` or `hPa`.

Installation

Warning: This API is not yet available in the [managed](#) apps.

To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Barometer } from 'expo';  
  
// in bare apps:  
import { Barometer } from 'expo-sensors';
```

OS	Units	Provider	Description
iOS	<code>hPa</code>	<code>CMAltimeter</code>	Altitude events reflect the change in the current altitude, not the absolute altitude.
Android	<code>hPa</code>	<code>Sensor.TYPE_PRESSURE</code>	Monitoring air pressure changes.
Web	N/A	N/A	This sensor is not available on the web and cannot be accessed. An <code>UnavailabilityError</code> will be thrown if you attempt to get data.

`Barometer.isAvailableAsync()`

Returns a promise which resolves into a boolean denoting the availability of the device barometer.

OS	Availability
iOS	iOS 8+
Android	Android 2.3+ (API Level 9+)
Web	N/A

Returns

- A promise that resolves to a `boolean` denoting the availability of the sensor.

`Barometer.addListener((data: BarometerMeasurement) => void)`

Subscribe for updates to the barometer.

```
const subscription = Barometer.addListener(({ pressure, relativeAltitude }) => {
  console.log({ pressure, relativeAltitude });
});
```

Arguments

- **listener (function)** -- A callback that is invoked when an barometer update is available. When invoked, the listener is provided a single argument that is an object containing: `pressure: number` (`hPa`). On **iOS** the `relativeAltitude: number` (`meters`) value will also be available.

Returns

- A subscription that you can call `remove()` on when you would like to unsubscribe the listener.

Barometer.removeAllListeners()

Remove all listeners.

Types

BarometerMeasurement

The altitude data returned from the native sensors.

```
type BarometerMeasurement = {
  pressure: number,
  /* iOS Only */
  relativeAltitude?: number,
};
```

Name	Type	Format	iOS	Android	Web
pressure	number	hPa			
relativeAltitude	`number`	undefined`	meters		

Example: basic subscription

```
import React from 'react';
import { Barometer } from 'expo-sensors';
import { StyleSheet, Text, TouchableOpacity, View } from 'react-native';

export default class BarometerSensor extends React.Component {
  state = {
    data: {},
  };

  componentDidMount() {
    this._toggle();
  }

  componentWillUnmount() {
```

```

    this._unsubscribe();
}

_toggle = () => {
  if (this._subscription) {
    this._unsubscribe();
  } else {
    this._subscribe();
  }
};

_subscribe = () => {
  this._subscription = Barometer.addListener(data => {
    this.setState({ data });
  });
};

_unsubscribe = () => {
  this._subscription && this._subscription.remove();
  this._subscription = null;
};

render() {
  const { pressure = 0 } = this.state.data;

  return (
    <View style={styles.sensor}>
      <Text>Barometer:</Text>
      <Text>{pressure * 100} Pa</Text>

      <View style={styles.buttonContainer}>
        <TouchableOpacity onPress={this._toggle} style={styles.button}>
          <Text>Toggle</Text>
        </TouchableOpacity>
      </View>
    </View>
  );
}
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  buttonContainer: {
    flexDirection: 'row',
    alignItems: 'stretch',
    marginTop: 15,
  },
  button: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#eee',
    padding: 10,
  },
  sensor: {
    marginTop: 15,
    paddingHorizontal: 10,
  },
});

```


BlurView

A React component that renders a native blur view on iOS and falls back to a semi-transparent view on Android. A common usage of this is for navigation bars, tab bars, and modals.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

API

```
// in managed apps:  
import { BlurView } from 'expo';  
  
// in bare apps:  
import { BlurView } from 'expo-blur';
```

props

```
tint A string: light , default , or dark .  
  
intensity A number from 1 to 100 to control the intensity of the blur effect.  
  
#
```

Branch

Expo includes alpha support for [Branch](#) attribution services.

Note: This API only works with standalone builds created with [expo build](#).

Installation

This API is pre-installed in [managed](#) apps. In a [bare](#) React Native app, you should use [react-native-branch-deep-linking](#) instead.

Usage

Importing Branch

The Branch SDK currently lives under Expo's **DangerZone** namespace because it's in a place where it might change significantly. You can import it like this:

```
import { DangerZone } from 'expo';
let { Branch } = DangerZone;
```

Configuration (standalone apps only)

- Add the **Branch Key** to your `app.json` in the section `android.config.branch.apiKey` and `ios.config.branch.apiKey`. You can find your key on [this page](#) of the Branch Dashboard.
- Add a **linking scheme** to your `app.json` in the `scheme` section if you don't already have one.

Enable Branch support for universal links (iOS only)

Branch can track universal links from domains you associate with your app. **Note:** Expo won't forward these to your JS via the normal Linking API.

- Enable associated domains on [Apple's Developer Portal](#) for your app id. To do so go in the `App IDs` section and click on your app id. Select `Edit`, check the `Associated Domains` checkbox and click `Done`.
- Enable Universal Links in the [Link Settings](#) section of the Branch Dashboard and fill in your Bundle Identifier and Apple App Prefix.
- Add an associated domain to support universal links to your `app.json` in the `ios.associatedDomains` section. This should be in the form of `applinks:<link-domain>` where `link-domain` can be found in the Link Domain section of the [Link Settings](#) page on the Branch Dashboard.

Using the Branch API

We pull in the API from [react-native-branch](#), so the documentation there is the best resource to follow. Make sure you import Branch using the above instructions (from `DangerZone.Branch`).

Example

Listen for links:

```
DangerZone.Branch.subscribe((bundle) => {
  if (bundle && bundle.params && !bundle.error) {
    // `bundle.params` contains all the info about the link.
  }
});
```

Open a share dialog:

```
class ArticleScreen extends Component {
  componentDidMount() {
    this.createBranchUniversalObject();
  }

  async createBranchUniversalObject() {
    const { article } = this.props;

    this._branchUniversalObject = await DangerZone.Branch.createBranchUniversalObject(
      `article_${article.id}`,
      {
        title: article.title,
        contentImageUrl: article.thumbnail,
        contentDescription: article.description,
        // This metadata can be used to easily navigate back to this screen
        // when implementing deep linking with `Branch.subscribe`.
        metadata: {
          screen: 'articleScreen',
          params: JSON.stringify({ articleId: article.id }),
        },
      },
    );
  }

  onShareLinkPress = async () => {
    const shareOptions = {
      messageHeader: this.props.article.title,
      messageBody: `Checkout my new article!`,
    };
    await this._branchUniversalObject.showShareSheet(shareOptions);
  };
}
```

#

Brightness

An API to get and set screen brightness.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Brightness } from 'expo';  
  
// in bare apps:  
import * as Brightness from 'expo-brightness';
```

Brightness.setBrightness(brightnessValue)

Sets screen brightness.

Arguments

- **brightnessValue (number)** -- A number between 0 and 1, representing the desired screen brightness.

Brightness.getBrightnessAsync()

Gets screen brightness.

Returns

A `Promise` that is resolved with a number between 0 and 1, representing the current screen brightness.

Brightness.setSystemBrightness(brightnessValue)

WARNING: this method is experimental.

Sets global system screen brightness, requires `WRITE_SETTINGS` permissions on Android.

Arguments

- **brightnessValue (number)** -- A number between 0 and 1, representing the desired screen brightness.

Example

```
await Permissions.askAsync(Permissions.SYSTEM_BRIGHTNESS);  
  
const { status } = await Permissions.getAsync(Permissions.SYSTEM_BRIGHTNESS);  
if (status === 'granted') {
```

```
Brightness.setSystemBrightness(100);
}
...
```

Brightness.getSystemBrightnessAsync()

WARNING: this method is experimental.

Gets global system screen brightness.

Returns

A `Promise` that is resolved with a number between 0 and 1, representing the current system screen brightness.

Calendar

Provides an API for interacting with the device's system calendars, events, reminders, and associated records.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Configuration

In managed apps, `Calendar` requires `Permissions.CALENDAR`. Interacting with reminders on iOS requires `Permissions.REMINDERS`.

API

```
// in managed apps:  
import { Calendar } from 'expo';  
  
// in bare apps:  
import * as Calendar from 'expo-calendar';
```

See the bottom of this page for a complete list of all possible fields for the objects used in this API.

Calendar.getCalendarsAsync(entityType)

Gets an array of calendar objects with details about the different calendars stored on the device.

Arguments

- **entityType (string)** -- (iOS only) Not required, but if defined, filters the returned calendars to a specific entity type. Possible values are `Calendar.EntityTypes.EVENT` (for calendars shown in the Calendar app) and `Calendar.EntityTypes.REMINDER` (for the Reminders app).

Returns

An array of [calendar objects](#) matching the provided entity type (if provided).

Calendar.requestRemindersPermissionsAsync()

iOS only. Requests the user for reminders permissions, same as `Permissions.askAsync(Permissions.REMINDERS)`.

Returns

Returns a promise resolving to an object with a key `granted` which value indicates whether the permission is granted or not.

Calendar.createCalendarAsync(details)

Creates a new calendar on the device, allowing events to be added later and displayed in the OS Calendar app.

Arguments

- **details (object)** --

A map of details for the calendar to be created (see below for a description of these fields):

- **title (string)** -- Required
- **color (string)** -- Required
- **entityType (string)** -- Required (iOS only)
- **sourceld (string)** -- Required (iOS only). ID of the source to be used for the calendar. Likely the same as the source for any other locally stored calendars.
- **source (object)** -- Required (Android only). Object representing the source to be used for the calendar.
 - **isLocalAccount (boolean)** -- Whether this source is the local phone account. Must be `true` if `type` is undefined.
 - **name (string)** -- Required. Name for the account that owns this calendar and was used to sync the calendar to the device.
 - **type (string)** -- Type of the account that owns this calendar and was used to sync it to the device. If `isLocalAccount` is falsy then this must be defined, and must match an account on the device along with `name`, or the OS will delete the calendar.
- **name (string)** -- Required (Android only)
- **ownerAccount (string)** -- Required (Android only)
- **timeZone (string)** -- (Android only)
- **allowedAvailabilities (array)** -- (Android only)
- **allowedReminders (array)** -- (Android only)
- **allowedAttendeeTypes (array)** -- (Android only)
- **isVisible (boolean)** -- (Android only)
- **isSynced (boolean)** -- (Android only)
- **accessLevel (string)** -- (Android only)

Returns

A string representing the ID of the newly created calendar.

Calendar.updateCalendarAsync(id, details)

Updates the provided details of an existing calendar stored on the device. To remove a property, explicitly set it to `null` in `details`.

Arguments

- **id (string)** -- ID of the calendar to update. Required.
- **details (object)** --

A map of properties to be updated (see below for a description of these fields):

- **title (string)**

- **sourceld (string)** -- (iOS only)
- **color (string)** -- (iOS only)
- **name (string)** -- (Android only)
- **isVisible (boolean)** -- (Android only)
- **isSynced (boolean)** -- (Android only)

Calendar.deleteCalendarAsync(id)

Deletes an existing calendar and all associated events/reminders/attendees from the device. Use with caution.

Arguments

- **id (string)** -- ID of the calendar to delete.

Calendar.getEventsAsync(calendarIds, startDate, endDate)

Returns all events in a given set of calendars over a specified time period. The filtering has slightly different behavior per-platform -- on iOS, all events that overlap at all with the `[startDate, endDate]` interval are returned, whereas on Android, only events that begin on or after the `startDate` and end on or before the `endDate` will be returned.

Arguments

- **calendarIds (array)** -- Array of IDs of calendars to search for events in. Required.
- **startDate (Date)** -- Beginning of time period to search for events in. Required.
- **endDate (Date)** -- End of time period to search for events in. Required.

Returns

An array of [event objects](#) matching the search criteria.

Calendar.getEventAsync(id, recurringEventOptions)

Returns a specific event selected by ID. If a specific instance of a recurring event is desired, the start date of this instance must also be provided, as instances of recurring events do not have their own unique and stable IDs on either iOS or Android.

Arguments

- **id (string)** -- ID of the event to return. Required.
- **recurringEventOptions (object)** --

A map of options for recurring events:

- **instanceStartDate (Date)** -- Date object representing the start time of the desired instance, if looking for a single instance of a recurring event. If this is not provided and `id` represents a recurring event, the first instance of that event will be returned by default.

Returns

An [event object](#) matching the provided criteria, if one exists.

Calendar.createEventAsync(calendarId, details)

Creates a new event on the specified calendar.

Arguments

- **calendarId (string)** -- ID of the calendar to create this event in (or `Calendar.DEFAULT` to add the calendar to the OS-specified default calendar for events). Required.
- **details (object)** --

A map of details for the event to be created (see below for a description of these fields):

- **title (string)**
- **startDate (Date)** -- Required.
- **endDate (Date)** -- Required on Android.
- **allDay (boolean)**
- **location (string)**
- **notes (string)**
- **alarms (Array)**
- **recurrenceRule (RecurrenceRule)**
- **availability (string)**
- **timeZone (string)** -- Required on Android.
- **endTimeZone (string)** -- (Android only)
- **url (string)** -- (iOS only)
- **organizerEmail (string)** -- (Android only)
- **accessLevel (string)** -- (Android only)
- **guestsCanModify (boolean)** -- (Android only)
- **guestsCanInviteOthers (boolean)** -- (Android only)
- **guestsCanSeeGuests (boolean)** -- (Android only)

Returns

A string representing the ID of the newly created event.

Calendar.updateEventAsync(id, details, recurringEventOptions)

Updates the provided details of an existing calendar stored on the device. To remove a property, explicitly set it to `null` in `details`.

Arguments

- **id (string)** -- ID of the event to be updated. Required.
- **details (object)** --

A map of properties to be updated (see below for a description of these fields):

- **title (string)**
- **startDate (Date)**
- **endDate (Date)**
- **allDay (boolean)**
- **location (string)**

- **notes (string)**
- **alarms (Array)**
- **recurrenceRule (RecurrenceRule)**
- **availability (string)**
- **timeZone (string)**
- **endTimeZone (string)** -- (Android only)
- **url (string)** -- (iOS only)
- **organizerEmail (string)** -- (Android only)
- **accessLevel (string)** -- (Android only)
- **guestsCanModify (boolean)** -- (Android only)
- **guestsCanInviteOthers (boolean)** -- (Android only)
- **guestsCanSeeGuests (boolean)** -- (Android only)
- **recurringEventOptions (object)** --

A map of options for recurring events:

- **instanceStartDate (Date)** -- Date object representing the start time of the desired instance, if wanting to update a single instance of a recurring event. If this is not provided and **id** represents a recurring event, the first instance of that event will be updated by default.
- **futureEvents (boolean)** -- Whether or not future events in the recurring series should also be updated. If `true`, will apply the given changes to the recurring instance specified by `instanceStartDate` and all future events in the series. If `false`, will only apply the given changes to the instance specified by `instanceStartDate`.

Calendar.deleteEventAsync(id, recurringEventOptions)

Deletes an existing event from the device. Use with caution.

Arguments

- **id (string)** -- ID of the event to be deleted. Required.
- **recurringEventOptions (object)** --

A map of options for recurring events:

- **instanceStartDate (Date)** -- Date object representing the start time of the desired instance, if wanting to delete a single instance of a recurring event. If this is not provided and **id** represents a recurring event, the first instance of that event will be deleted by default.
- **futureEvents (boolean)** -- Whether or not future events in the recurring series should also be deleted. If `true`, will delete the instance specified by `instanceStartDate` and all future events in the series. If `false`, will only delete the instance specified by `instanceStartDate`.

Calendar.getAttendeesForEventAsync(eventId, recurringEventOptions)

Gets all attendees for a given event (or instance of a recurring event).

Arguments

- **eventId (string)** -- ID of the event to return attendees for. Required.
- **recurringEventOptions (object)** --

A map of options for recurring events:

- **instanceStartDate (*Date*)** -- Date object representing the start time of the desired instance, if looking for a single instance of a recurring event. If this is not provided and **eventId** represents a recurring event, the attendees of the first instance of that event will be returned by default.

Returns

An array of [attendee objects](#) associated with the specified event.

Calendar.createAttendeeAsync(*eventId*, *details*)

Available on Android only. Creates a new attendee record and adds it to the specified event. Note that if *eventId* specifies a recurring event, this will add the attendee to every instance of the event.

Arguments

- **eventId (*string*)** -- ID of the event to add this attendee to. Required.
- **details (*object*)** --

A map of details for the attendee to be created (see below for a description of these fields):

- **id (*string*)** Required.
- **email (*string*)** Required.
- **name (*string*)**
- **role (*string*)** Required.
- **status (*string*)** Required.
- **type (*string*)** Required.

Returns

A string representing the ID of the newly created attendee record.

Calendar.updateAttendeeAsync(*id*, *details*)

Available on Android only. Updates an existing attendee record. To remove a property, explicitly set it to `null` in *details*.

Arguments

- **id (*string*)** -- ID of the attendee record to be updated. Required.
- **details (*object*)** --

A map of properties to be updated (see below for a description of these fields):

- **id (*string*)**
- **email (*string*)**
- **name (*string*)**
- **role (*string*)**
- **status (*string*)**
- **type (*string*)**

`Calendar.deleteAttendeeAsync(id)`

Available on Android only. Deletes an existing attendee record from the device. Use with caution.

Arguments

- **id (string)** -- ID of the attendee to delete.

`Calendar.getRemindersAsync(calendarIds, status, startDate, endDate)`

Available on iOS only. Returns a list of reminders matching the provided criteria. If `startDate` and `endDate` are defined, returns all reminders that overlap at all with the [start, end] interval -- i.e. all reminders that end after the `startDate` or begin before the `endDate`.

Arguments

- **calendarIds (array)** -- Array of IDs of calendars to search for reminders in. Required.
- **status (string)** -- One of `Calendar.ReminderStatus.COMPLETED` OR `Calendar.ReminderStatus.INCOMPLETE`.
- **startDate (Date)** -- Beginning of time period to search for reminders in. Required if `status` is defined.
- **endDate (Date)** -- End of time period to search for reminders in. Required if `status` is defined.

Returns

An array of [reminder objects](#) matching the search criteria.

`Calendar.getReminderAsync(id)`

Available on iOS only. Returns a specific reminder selected by ID.

Arguments

- **id (string)** -- ID of the reminder to return. Required.

Returns

An [reminder object](#) matching the provided ID, if one exists.

`Calendar.createReminderAsync(calendarId, details)`

Available on iOS only. Creates a new reminder on the specified calendar.

Arguments

- **calendarId (string)** -- ID of the calendar to create this reminder in (or `Calendar.DEFAULT` to add the calendar to the OS-specified default calendar for reminders). Required.
- **details (object)** --

A map of details for the reminder to be created: (see below for a description of these fields)

- **title (string)**

- **startDate** (*Date*)
- **dueDate** (*Date*)
- **completed** (*boolean*)
- **completionDate** (*Date*)
- **location** (*string*)
- **notes** (*string*)
- **alarms** (*array*)
- **recurrenceRule** (*RecurrenceRule*)
- **timeZone** (*string*)
- **url** (*string*)

Returns

A string representing the ID of the newly created reminder.

`Calendar.updateReminderAsync(id, details)`

Available on iOS only. Updates the provided details of an existing reminder stored on the device. To remove a property, explicitly set it to `null` in `details`.

Arguments

- **id** (*string*) -- ID of the reminder to be updated. Required.
- **details** (*object*) --

A map of properties to be updated (see below for a description of these fields):

- **title** (*string*)
- **startDate** (*Date*)
- **dueDate** (*Date*)
- **completionDate** (*Date*) -- Setting this property of a nonnull Date will automatically set the reminder's `completed` value to `true`.
- **location** (*string*)
- **notes** (*string*)
- **alarms** (*array*)
- **recurrenceRule** (*RecurrenceRule*)
- **timeZone** (*string*)
- **url** (*string*)

`Calendar.deleteReminderAsync(id)`

Available on iOS only. Deletes an existing reminder from the device. Use with caution.

Arguments

- **id** (*string*) -- ID of the reminder to be deleted. Required.

`Calendar.getSourcesAsync()`

Available on iOS only.

Returns

An array of [source objects](#) all sources for calendars stored on the device.

Calendar.getSourceAsync(id)

Available on iOS only. Returns a specific source selected by ID.

Arguments

- **id (*string*)** -- ID of the source to return. Required.

Returns

A [source object](#) matching the provided ID, if one exists.

Calendar.openEventInCalendar(id)

Available on Android only. Sends an intent to open the specified event in the OS Calendar app.

Arguments

- **id (*string*)** -- ID of the event to open. Required.

List of object properties

Calendar

A calendar record upon which events (or, on iOS, reminders) can be stored. Settings here apply to the calendar as a whole and how its events are displayed in the OS calendar app.

Field name	Type	Platforms	Description	Possible values
id	<i>string</i>	both	Internal ID that represents this calendar on the device	
title	<i>string</i>	both	Visible name of the calendar	
entityType	<i>string</i>	iOS	Whether the calendar is used in the Calendar or Reminders OS app	<code>Calendar.EntityTypes.EVENT</code> , <code>Calendar.EntityTypes.REMINDER</code>
source	<i>Source</i>	both	Object representing the source to be used for the	

			calendar	
color	<i>string</i>	both	Color used to display this calendar's events	
allowsModifications	<i>boolean</i>	both	Boolean value that determines whether this calendar can be modified	
type	<i>string</i>	iOS	Type of calendar this object represents	<code>Calendar.CalendarType.LOCAL</code> , <code>Calendar.CalendarType.CALDAV</code> , <code>Calendar.CalendarType.EXCHANGE</code> , <code>Calendar.CalendarType.SUBSCRIBED</code> , <code>Calendar.CalendarType.BIRTHDAYS</code>
isPrimary	<i>boolean</i>	Android	Boolean value indicating whether this is the device's primary calendar	
name	<i>string</i>	Android	Internal system name of the calendar	
ownerAccount	<i>string</i>	Android	Name for the account that owns this calendar	
timeZone	<i>string</i>	Android	Time zone for the calendar	
allowedAvailabilities	<i>array</i>	both	Availability types that this calendar supports	<code>array of Calendar.Availability.BUSY</code> , <code>Calendar.Availability.FREE</code> , <code>Calendar.Availability.TENTATIVE</code> , <code>Calendar.Availability.UNAVAILABLE</code> (iOS only), <code>Calendar.Availability.NOT_SUPPORTED</code> (iOS only)
allowedReminders	<i>array</i>	Android	Alarm methods that this calendar supports	<code>array of Calendar.AlarmMethod.ALARM</code> , <code>Calendar.AlarmMethod.ALERT</code> , <code>Calendar.AlarmMethod.EMAIL</code> , <code>Calendar.AlarmMethod.SMS</code> , <code>Calendar.AlarmMethod.DEFAULT</code>
allowedAttendeeTypes	<i>array</i>	Android	Attendee types that this calendar	<code>array of Calendar.AttendeeType.UNKNOWN</code> (iOS only), <code>Calendar.AttendeeType.PERSON</code> (iOS only), <code>Calendar.AttendeeType.ROOM</code> (iOS only), <code>Calendar.AttendeeType.GROUP</code> (iOS only), <code>Calendar.AttendeeType.RESOURCE</code> , <code>Calendar.AttendeeType.OPTIONAL</code> (Android only)

			supports	only), <code>Calendar.AttendeeType.REQUIRED</code> (Android only), <code>Calendar.AttendeeType.NONE</code> (Android only)
isVisible	<code>boolean</code>	Android	Indicates whether the OS displays events on this calendar	
isSynced	<code>boolean</code>	Android	Indicates whether this calendar is synced and its events stored on the device	
accessLevel	<code>string</code>	Android	Level of access that the user has for the calendar	<code>Calendar.CalendarAccessLevel.CONTRIBUTOR</code> , <code>Calendar.CalendarAccessLevel.EDITOR</code> , <code>Calendar.CalendarAccessLevel.FREEBUSY</code> , <code>Calendar.CalendarAccessLevel.OVERRIDE</code> , <code>Calendar.CalendarAccessLevel.OWNER</code> , <code>Calendar.CalendarAccessLevel.READ</code> , <code>Calendar.CalendarAccessLevel.RESPOND</code> , <code>Calendar.CalendarAccessLevel.ROOT</code> , <code>Calendar.CalendarAccessLevel.NONE</code>

Event

An event record, or a single instance of a recurring event. On iOS, used in the Calendar app.

Field name	Type	Platforms	Description	Possible values
id	<code>string</code>	both	Internal ID that represents this event on the device	
calendarId	<code>string</code>	both	ID of the calendar that contains this event	
title	<code>string</code>	both	Visible name of the event	
startDate	<code>Date</code>	both	Date object or string representing the time when the event starts	
endDate	<code>Date</code>	both	Date object or string representing the time when the	

			event ends	
allDay	boolean	both	Whether the event is displayed as an all-day event on the calendar	
location	string	both	Location field of the event	
notes	string	both	Description or notes saved with the event	
alarms	array	both	Array of Alarm objects which control automated reminders to the user	
recurrenceRule	RecurrenceRule	both	Object representing rules for recurring or repeating events. Null for one-time events.	
availability	string	both	The availability setting for the event	Calendar.Availability.BUSY , Calendar.Availability.FREE , Calendar.Availability.TENTATIVE , Calendar.Availability.UNAVAILABLE (iOS only), Calendar.Availability.NOT_SUPPORTED (iOS only)
timeZone	string	both	Time zone the event is scheduled in	
endTimeZone	string	Android	Time zone for the event end time	
url	string	iOS	URL for the event	
creationDate	string	iOS	Date when the event record was created	
lastModifiedDate	string	iOS	Date when the event record was last modified	

originalStartDate	<i>string</i>	iOS	For recurring events, the start date for the first (original) instance of the event	
isDetached	<i>boolean</i>	iOS	Boolean value indicating whether or not the event is a detached (modified) instance of a recurring event	
status	<i>string</i>	iOS	Status of the event	<code>Calendar.EventStatus.NONE</code> , <code>Calendar.EventStatus.CONFIRMED</code> , <code>Calendar.EventStatus.TENTATIVE</code> , <code>Calendar.EventStatus.CANCELED</code>
organizer	<i>Attendee</i>	iOS	Organizer of the event, as an Attendee object	
organizerEmail	<i>string</i>	Android	Email address of the organizer of the event	
accessLevel	<i>string</i>	Android	User's access level for the event	<code>Calendar.EventAccessLevel.CONFIDENTIAL</code> , <code>Calendar.EventAccessLevel.PRIVATE</code> , <code>Calendar.EventAccessLevel.PUBLIC</code> , <code>Calendar.EventAccessLevel.DEPERSONALIZED</code>
guestsCanModify	<i>boolean</i>	Android	Whether invited guests can modify the details of the event	
guestsCanInviteOthers	<i>boolean</i>	Android	Whether invited guests can invite other guests	
guestsCanSeeGuests	<i>boolean</i>	Android	Whether invited guests can see other guests	
originalId	<i>string</i>	Android	For detached (modified) instances of recurring	

originalId	<i>string</i>	Android	events, the ID of the original recurring event	
instanceId	<i>string</i>	Android	For instances of recurring events, volatile ID representing this instance; not guaranteed to always refer to the same instance	

Reminder (iOS only)

A reminder record, used in the iOS Reminders app. No direct analog on Android.

Field name	Type	Description
id	<i>string</i>	Internal ID that represents this reminder on the device
calendarId	<i>string</i>	ID of the calendar that contains this reminder
title	<i>string</i>	Visible name of the reminder
startDate	<i>Date</i>	Date object or string representing the start date of the reminder task
dueDate	<i>Date</i>	Date object or string representing the time when the reminder task is due
completed	<i>boolean</i>	Indicates whether or not the task has been completed
completionDate	<i>Date</i>	Date object or string representing the date of completion, if <code>completed</code> is <code>true</code>
location	<i>string</i>	Location field of the reminder
notes	<i>string</i>	Description or notes saved with the reminder
alarms	<i>array</i>	Array of Alarm objects which control automated alarms to the user about the task
recurrenceRule	<i>RecurrenceRule</i>	Object representing rules for recurring or repeated reminders. Null for one-time tasks.
timeZone	<i>string</i>	Time zone the reminder is scheduled in
url	<i>string</i>	URL for the reminder
creationDate	<i>string</i>	Date when the reminder record was created
lastModifiedDate	<i>string</i>	Date when the reminder record was last modified

Attendee

A person or entity that is associated with an event by being invited or fulfilling some other role.

Field name	Type	Platforms	Description	Possible values
id	<i>string</i>	Android	Internal ID that represents this attendee on the device	
email	<i>string</i>	Android	Email address of the attendee	
name	<i>string</i>	both	Displayed name of the attendee	
role	<i>string</i>	both	Role of the attendee at the event	Calendar.AttendeeRole.UNKNOWN (iOS only), Calendar.AttendeeRole.REQUIRED (iOS only), Calendar.AttendeeRole.OPTIONAL (iOS only), Calendar.AttendeeRole.CHAIR (iOS only), Calendar.AttendeeRole.NON_PARTICIPANT (iOS only), Calendar.AttendeeRole.ATTENDEE (Android only), Calendar.AttendeeRole.ORGANIZER (Android only), Calendar.AttendeeRole.PERFORMER (Android only), Calendar.AttendeeRole.SPEAKER (Android only), Calendar.AttendeeRole.NONE (Android only)
status	<i>string</i>	both	Status of the attendee in relation to the event	Calendar.AttendeeStatus.ACCEPTED , Calendar.AttendeeStatus.DECLINED , Calendar.AttendeeStatus.TENTATIVE , Calendar.AttendeeStatus.DELEGATED (iOS only), Calendar.AttendeeStatus.COMPLETED (iOS only), Calendar.AttendeeStatus.IN_PROCESS (iOS only), Calendar.AttendeeStatus.UNKNOWN (iOS only), Calendar.AttendeeStatus.PENDING (iOS only), Calendar.AttendeeStatus.INVITED (Android only), Calendar.AttendeeStatus.NONE (Android only)
type	<i>string</i>	both	Type of the attendee	Calendar.AttendeeType.UNKNOWN (iOS only), Calendar.AttendeeType.PERSON (iOS only), Calendar.AttendeeType.ROOM (iOS only), Calendar.AttendeeType.GROUP (iOS only), Calendar.AttendeeType.RESOURCE , Calendar.AttendeeType.OPTIONAL (Android only), Calendar.AttendeeType.REQUIRED (Android only), Calendar.AttendeeType.NONE (Android only)
url	<i>string</i>	iOS	URL for the attendee	
isCurrentUser	<i>boolean</i>	iOS	Indicates whether or not this attendee is the current OS user	

RecurrenceRule

A recurrence rule for events or reminders, allowing the same calendar item to recur multiple times.

Field name	Type	Description	Possible values
frequency	string	How often the calendar item should recur	<code>Calendar.Frequency.DAILY</code> , <code>Calendar.Frequency.WEEKLY</code> , <code>Calendar.Frequency.MONTHLY</code> , <code>Calendar.Frequency.YEARLY</code>
interval	number	Interval at which the calendar item should recur. For example, an <code>interval: 2</code> with <code>frequency: DAILY</code> would yield an event that recurs every other day. Defaults to <code>1</code> .	
endDate	Date	Date on which the calendar item should stop recurring; overrides <code>occurrence</code> if both are specified	
occurrence	number	Number of times the calendar item should recur before stopping	

Alarm

A method for having the OS automatically remind the user about an calendar item

Field name	Type	Platforms	Description	Possible values
absoluteDate	Date	iOS	Date object or string representing an absolute time the alarm should occur; overrides <code>relativeOffset</code> and <code>structuredLocation</code> if specified alongside either	
relativeOffset	number	both	Number of minutes from the <code>startDate</code> of the calendar item that the alarm should occur; use negative values to have the alarm occur before the <code>startDate</code>	
method	string	Android	Method of alerting the user that this alarm should use; on iOS this is always a notification	<code>Calendar.AlarmMethod.ALARM</code> , <code>Calendar.AlarmMethod.ALERT</code> , <code>Calendar.AlarmMethod.EMAIL</code> , <code>Calendar.AlarmMethod.SMS</code> , <code>Calendar.AlarmMethod.DEFAULT</code>

Source

A source account that owns a particular calendar. Expo apps will typically not need to interact with Source objects.

Field name	Type	Platforms	Description	Possible values
id	string	iOS	Internal ID that represents this source	

			on the device	
name	<i>string</i>	both	Name for the account that owns this calendar	
type	<i>string</i>	both	Type of account that owns this calendar	on iOS, one of <code>Calendar.SourceType.LOCAL</code> , <code>Calendar.SourceType.EXCHANGE</code> , <code>Calendar.SourceType.CALDAV</code> , <code>Calendar.SourceType.MOBILEME</code> , <code>Calendar.SourceType.SUBSCRIBED</code> , or <code>Calendar.SourceType.BIRTHDAYS</code>
isLocalAccount	<i>boolean</i>	Android	Whether this source is the local phone account	

#

Camera

A React component that renders a preview for the device's either front or back camera. Camera's parameters like zoom, auto focus, white balance and flash mode are adjustable. With use of `Camera` one can also take photos and record videos that are saved to the app's cache. Moreover, the component is also capable of detecting faces and bar codes appearing on the preview.

Note: Only one active Camera preview is supported currently. When using navigation, the best practice is to unmount previously rendered `Camera` component so next screens can use camera without issues.

Note: Android devices can use one of two available Camera apis underneath. This was previously chosen automatically, based on the device's Android system version and camera hardware capabilities. As we experienced some issues with Android's Camera2 API, we decided to choose the older API as a default. However, using the newer one is still possible through setting `useCamera2Api` prop to true. The change we made should be barely visible - the only thing that is not supported using the old Android's API is setting focus depth.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Configuration

In managed apps, `Camera` requires `Permissions.CAMERA`. Video recording requires `Permissions.AUDIO_RECORDING`.

Usage

Basic Example

```
import React from 'react';
import { Text, View, TouchableOpacity } from 'react-native';
import { Camera, Permissions } from 'expo';

export default class CameraExample extends React.Component {
  state = {
    hasCameraPermission: null,
    type: Camera.Constants.Type.back,
  };

  async componentDidMount() {
    const { status } = await Permissions.askAsync(Permissions.CAMERA);
    this.setState({ hasCameraPermission: status === 'granted' });
  }

  render() {
    const { hasCameraPermission } = this.state;
    if (hasCameraPermission === null) {
      return <View />;
    } else if (hasCameraPermission === false) {
      return <Text>No access to camera</Text>;
    } else {
  
```

```

        return (
          <View style={{ flex: 1 }}>
            <Camera style={{ flex: 1 }} type={this.state.type}>
              <View
                style={{
                  flex: 1,
                  backgroundColor: 'transparent',
                  flexDirection: 'row',
                }}>
                <TouchableOpacity
                  style={{
                    flex: 0.1,
                    alignSelf: 'flex-end',
                    alignItems: 'center',
                  }}
                  onPress={() => {
                    this.setState({
                      type: this.state.type === Camera.Constants.Type.back
                        ? Camera.Constants.Type.front
                        : Camera.Constants.Type.back,
                    });
                  }}>
                  <Text
                    style={{ fontSize: 18, marginBottom: 10, color: 'white' }}>
                    {' '}Flip{' '}
                  </Text>
                </TouchableOpacity>
              </View>
            </Camera>
          </View>
        );
      }
    }
  }
}

```

Comprehensive Example

Check out a full example at [expo/camerja](#). You can try it with Expo at [@community/camerja](#).

API

```

// in managed apps:
import { Camera } from 'expo';

// in bare apps:
import { Camera } from 'expo-camera';

```

props

- **type**

Camera facing. Use one of `Camera.Constants.Type`. When `Type.front`, use the front-facing camera. When `Type.back`, use the back-facing camera. Default: `Type.back`.

- **flashMode**

Camera flash mode. Use one of `Camera.Constants.FlashMode`. When `on`, the flash on your device will turn on when taking a picture, when `off`, it won't. Setting to `auto` will fire flash if required, `torch` turns on flash during the preview. Default: `off`.

- **autoFocus**

State of camera auto focus. Use one of `Camera.Constants.AutoFocus`. When `on`, auto focus will be enabled, when `off`, it won't and focus will lock as it was in the moment of change but it can be adjusted on some devices via `focusDepth` prop.

- **zoom (float)**

A value between 0 and 1 being a percentage of device's max zoom. 0 - not zoomed, 1 - maximum zoom. Default: 0.

- **whiteBalance**

Camera white balance. Use one of `Camera.Constants.WhiteBalance`: `auto`, `sunny`, `cloudy`, `shadow`, `fluorescent`, `incandescent`. If a device does not support any of these values previous one is used.

- **focusDepth (float)**

Distance to plane of sharpest focus. A value between 0 and 1: 0 - infinity focus, 1 - focus as close as possible. Default: 0. For Android this is available only for some devices and when `useCamera2Api` is set to true.

- **ratio (string)**

Android only. A string representing aspect ratio of the preview, eg. `4:3`, `16:9`, `1:1`. To check if a ratio is supported by the device use `getSupportedRatiosAsync`. Default: `4:3`.

- **pictureSize (string)**

A string representing the size of pictures `takePictureAsync` will take. Available sizes can be fetched with `getAvailablePictureSizesAsync`.

- **onCameraReady (function)**

Callback invoked when camera preview has been set.

- **onFacesDetected (function)**

Callback invoked with results of face detection on the preview. See [FaceDetector documentation](#) for details.

- **faceDetectorSettings (Object)**

A settings object passed directly to an underlying module providing face detection features. See [FaceDetector documentation](#) for details.

- **onMountError (function)**

Callback invoked when camera preview could not be started. It is provided with an error object that contains a `message`.

- **onBarcodeRead (function)**

Deprecated. Use `onBarcodeScanned` instead.

- **onBarcodeScanned (function)**

Callback that is invoked when a bar code has been successfully scanned. The callback is provided with an object of the shape `{ type: BarCodeScanner.Constants.BarCodeType, data: string }`, where the type refers to the bar code type that was scanned and the data is the information encoded in the bar code (in this case of QR codes, this is often a URL). See `BarCodeScanner.Constants.BarCodeType` for supported values.

- **barCodeTypes (Array)**

Deprecated. Use `barCodeScannerSettings` instead.

- **barCodeScannerSettings (object)**

Settings exposed by `BarCodeScanner` module. Supported settings: `[barCodeTypes]`.

```
<Camera
  barCodeScannerSettings={{
    barCodeTypes: [BarCodeScanner.Constants.BarCodeType.qr]
  }}
/>
```

- **useCamera2Api (boolean)**

Android only. Whether to use Android's Camera2 API. See `Note` at the top of this page.

- **videoStabilizationMode (Camera.Constants.VideoStabilization)**

iOS only. The video stabilization mode used for a video recording. Use one of `Camera.Constants.VideoStabilization.{off, standard, cinematic, auto}`.

You can read more about each stabilization type [here](#).

Methods

To use methods that Camera exposes one has to create a components `ref` and invoke them using it.

```
// ...
<Camera ref={ref => { this.camera = ref; }} />
// ...
snap = async () => {
  if (this.camera) {
    let photo = await this.camera.takePictureAsync();
  }
};
```

takePictureAsync

Takes a picture and saves it to app's cache directory. Photos are rotated to match device's orientation (if `options.skipProcessing` flag is not enabled) and scaled to match the preview. Anyway on Android it is essential to set `ratio` prop to get a picture with correct dimensions.

Arguments

- **options (object) --**

A map of options:

- **quality (number) --** Specify the quality of compression, from 0 to 1.0 means compress for small size, 1

- means compress for maximum quality.
- **base64 (boolean)** -- Whether to also include the image data in Base64 format.
- **exif (boolean)** -- Whether to also include the EXIF data for the image.
- **onPictureSaved (function)** -- A callback invoked when picture is saved. If set, the promise of this method will resolve immediately with no data after picture is captured. The data that it should contain will be passed to this callback. If displaying or processing a captured photo right after taking it is not your case, this callback lets you skip waiting for it to be saved.
- **skipProcessing (boolean)** - Android only. If set to `true`, camera skips orientation adjustment and returns an image straight from the device's camera. If enabled, `quality` option is discarded (processing pipeline is skipped as a whole). Although enabling this option reduces image delivery time significantly, it may cause the image to appear in a wrong orientation in the `Image` component (at the time of writing, it does not respect EXIF orientation of the images).

Note: Enabling `skipProcessing` would cause orientation uncertainty. `Image` component does not respect EXIF stored orientation information, that means obtained image would be displayed wrongly (rotated by 90°, 180° or 270°). Different devices provide different orientations. For example some SonyExperia or Samosung devices don't provide correctly oriented images by default. To always obtain correctly oriented image disable `skipProcessing` option.

Returns

Returns a Promise that resolves to an object: `{ uri, width, height, exif, base64 }` where `uri` is a URI to the local image file (useable as the source for an `Image` element) and `width, height` specify the dimensions of the image. `base64` is included if the `base64` option was truthy, and is a string containing the JPEG data of the image in Base64--prepend that with `'data:image/jpg;base64,'` to get a data URI, which you can use as the source for an `Image` element for example. `exif` is included if the `exif` option was truthy, and is an object containing EXIF data for the image--the names of its properties are EXIF tags and their values are the values for those tags.

The local image URI is temporary. Use `Expo.FileSystem.copyAsync` to make a permanent copy of the image.

recordAsync

Starts recording a video that will be saved to cache directory. Videos are rotated to match device's orientation. Flipping camera during a recording results in stopping it.

Arguments

- **options (object)** --

A map of options:

- **quality (VideoQuality)** -- Specify the quality of recorded video. Usage:
`Camera.Constants.VideoQuality['<value>']`, possible values: for 16:9 resolution `2160p`, `1080p`, `720p`, `480p`: Android only and for 4:3 `4:3` (the size is 640x480). If the chosen quality is not available for a device, the highest available is chosen.
- **maxDuration (number)** -- Maximum video duration in seconds.
- **maxFileSize (number)** -- Maximum video file size in bytes.
- **mute (boolean)** -- If present, video will be recorded with no sound.

Returns

Returns a Promise that resolves to an object containing video file `uri` property. The Promise is returned if `stopRecording` was invoked, one of `maxDuration` and `maxFileSize` is reached or camera preview is stopped.

stopRecording

Stops recording if any is in progress.

getSupportedRatiosAsync

Android only. Get aspect ratios that are supported by the device and can be passed via `ratio` prop.

Returns

Returns a Promise that resolves to an array of strings representing ratios, eg. `['4:3', '1:1']`.

getAvailablePictureSizesAsync

Get picture sizes that are supported by the device for given `ratio`.

Arguments

- `ratio (string)` -- A string representing aspect ratio of sizes to be returned.

Returns

Returns a Promise that resolves to an array of strings representing picture sizes that can be passed to `pictureSize` prop. The list varies across Android devices but is the same for every iOS.

pausePreview

Pauses the camera preview. It is not recommended to use `takePictureAsync` when preview is paused.

resumePreview

Resumes the camera preview.

Constants

System information that remains constant throughout the lifetime of your app.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Constants } from 'expo';  
  
// in bare apps:  
import Constants from 'expo-constants';
```

Constants.appOwnership

Returns `expo`, `standalone`, or `guest`. If `expo`, the experience is running inside of the Expo client. If `standalone`, it is a [standalone app](#). If `guest`, it has been opened through a link from a standalone app.

Constants.expoVersion

The version string of the Expo client currently running.

Constants.installationId

An identifier that is unique to this particular device and installation of the Expo client.

Constants.deviceName

A human-readable name for the device type.

Constants.deviceYearClass

The [device year class](#) of this device.

Constants.getWebViewUserAgentAsync()

Gets the user agent string which would be included in requests sent by a web view running on this device. This is probably not the same user agent you might be providing in your JS `fetch` requests.

Constants.isDevice

`true` if the app is running on a device, `false` if running in a simulator or emulator.

Constants.platform

- ios

- buildNumber

The build number specified in the embedded `Info.plist` value for `CFBundleVersion` in this app. In a standalone app, you can set this with the `ios.buildNumber` value in `app.json`. This may differ from the value in `Constants.manifest.ios.buildNumber` because the manifest can be updated over the air, whereas this value will never change for a given native binary.

- platform

The Apple internal model identifier for this device, e.g. `iPhone1,1`.

- model

The human-readable model name of this device, e.g. `iPhone 7 Plus`.

- userInterfaceIdiom

The user interface idiom of this device, i.e. whether the app is running on an iPhone or an iPad. Current supported values are `handset` and `tablet`. Apple TV and CarPlay will show up as `unsupported`.

- systemVersion

The version of iOS running on this device, e.g. `10.3`.

- android

- versionCode

The version code set by `android.versionCode` in `app.json`.

Constants.sessionId

A string that is unique to the current session of your app. It is different across apps and across multiple launches of the same app.

Constants.statusBarHeight

The default status bar height for the device. Does not factor in changes when location tracking is in use or a phone call is active.

Constants.systemFonts

A list of the system font names available on the current device.

Constants.manifest

The `manifest` object for the app.

Contacts

Provides access to the phone's system contacts.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Contacts } from 'expo';  
  
// in bare apps:  
import * as Contacts from 'expo-contacts';
```

getContactsAsync

```
getContactsAsync(contactQuery: ContactQuery): Promise<ContactResponse>
```

Return a list of contacts that fit a given criteria. You can get all of the contacts by passing no criteria.

Parameters

Name	Type	Description
contactQuery	ContactQuery	Used to query contacts.

Returns

Name	Type	Description
contactResponse	ContactResponse	Contacts returned from the query.

Example

```
const { data } = await Contacts.getContactsAsync({  
  fields: [Contacts.Fields.Emails],  
});  
  
if (data.length > 0) {  
  const contact = data[0];  
  console.log(contact);  
}
```

getContactByIdAsync

```
getContactByIdAsync(contactId: string, fields: FieldType[]): Promise<Contact>
```

Returns a contact matching the input id. Used for gathering precise data about a contact.

Parameters

Name	Type	Description
contactId	string	The ID of a system contact.
fields	FieldType[]	If available the fields defined will be returned. If <code>nil</code> then all fields will be returned.

Returns

Name	Type	Description
contact	Contact	Contact with an ID matching the input ID.

Example

```
const contact = await Contacts.getContactByIdAsync("161A368D-D614-4A15-8DC6-665FDBCFAE55");
if (contact) {
    console.log(contact);
}
```

addContactAsync

iOS Only - temporary

```
addContactAsync(contact: Contact, containerId: string): Promise<string>
```

Creates a new contact and adds it to the system.

Parameters

Name	Type	Description
contact	Contact	A contact with the changes you wish to persist. The <code>id</code> parameter will not be used.
containerId	string	IOS ONLY: The container that will parent the contact

Returns

Name	Type	Description
contactId	string	ID of the new system contact.

Example

```
const contact = {
    [Contacts.Fields.FirstName]: "Bird",
    [Contacts.Fields.LastName]: "Man",
    [Contacts.Fields.Company]: "Young Money",
}
const contactId = await Contacts.addContactAsync(contact);
```

updateContactAsync

iOS Only - temporary

```
updateContactAsync(contact: Contact): Promise<string>
```

Mutate the information of an existing contact.

On Android, you can use `presentFormAsync` to make edits to contacts. Do to an error with the Apple API, `nonGregorianBirthday` cannot be modified.

Parameters

Name	Type	Description
contact	Contact	A contact with the changes you wish to persist. The contact must contain a valid id

Returns

Name	Type	Description
contactId	string	The ID of a system contact.

Example

```
const contact = {
  id: "161A368D-D614-4A15-8DC6-665FDBCFAE55",
  [Contacts.Fields.FirstName]: "Drake",
  [Contacts.Fields.Company]: "Young Money",
}
await Contacts.updateContactAsync(contact);
```

removeContactAsync

iOS Only - temporary

```
removeContactAsync(contactId: string): Promise<any>
```

Delete a contact from the system.

Parameters

Name	Type	Description
contactId	string	ID of the contact you want to delete.

Example

```
await Contacts.removeContactAsync("161A368D-D614-4A15-8DC6-665FDBCFAE55");
```

writeContactToFileAsync

```
writeContactToFileAsync(contactQuery: ContactQuery): Promise<string>
```

Query a set of contacts and write them to a local uri that can be used for sharing with `ReactNative.Share`.

Parameters

Name	Type	Description
contactQuery	ContactQuery	Used to query contacts you want to write.

Returns

Name	Type	Description
localUri	string	Shareable local uri

Example

```
const localUri = await Contacts.writeContactToFileAsync({ id: "161A368D-D614-4A15-8DC6-665FDBCFAE55" });
Share.share({ url: localUri, message: "Call me!" });
```

IOS Only Functions

iOS contacts have a multi-layered grouping system that you can access through this API.

presentFormAsync

```
presentFormAsync(contactId: string, contact: Contact, formOptions: FormOptions): Promise<any>
```

Present a native form for manipulating contacts

Parameters

Name	Type	Description
contactId	string	The ID of a system contact.
contact	Contact	A contact with the changes you wish to persist.
formOptions	FormOptions	Options for the native editor

Example

```
// Edit contact
await Contacts.presentFormAsync("161A368D-D614-4A15-8DC6-665FDBCFAE55");
```

addExistingGroupToContainerAsync

```
addExistingGroupToContainerAsync(groupId: string, containerId: string): Promise<any>
```

Add a group to a container.

Parameters

Name	Type	Description

groupId	string	The group you wish to target.
containerId	string	The container you to add membership to.

Example

```
await Contacts.addExistingGroupToContainerAsync("161A368D-D614-4A15-8DC6-665FDBCFAE55", "665FDBCFAE55-D614-4A15-8DC6-161A368D");
```

createGroupAsync

```
createGroupAsync(groupName: string, containerId?: string): Promise<string>
```

Create a group with a name, and add it to a container. If the container is undefined, the default container will be targeted.

Parameters

Name	Type	Description
name	string	Name of the new group.
containerId	string	The container you to add membership to.

Returns

Name	Type	Description
groupId	string	ID of the new group.

Example

```
const groupId = await Contacts.createGroupAsync("Sailor Moon");
```

updateGroupNameAsync

```
updateGroupNameAsync(groupName: string, groupId: string): Promise<any>
```

Change the name of an existing group.

Parameters

Name	Type	Description
groupName	string	New name for an existing group.
groupId	string	ID for the group you want to edit.

Example

```
await Contacts.updateGroupName("Sailor Moon", "161A368D-D614-4A15-8DC6-665FDBCFAE55");
```

removeGroupAsync

```
removeGroupAsync(groupId: string): Promise<any>
```

Delete a group from the device.

Parameters

Name	Type	Description
groupId	string	ID of the group.

Example

```
await Contacts.removeGroupAsync("161A368D-D614-4A15-8DC6-665FDBCFAE55");
```

addExistingContactToGroupAsync

```
addExistingContactToGroupAsync(contactId: string, groupId: string): Promise<any>
```

Add a contact as a member to a group. A contact can be a member of multiple groups.

Parameters

Name	Type	Description
contactId	string	ID of the contact you want to edit.
groupId	string	ID for the group you want to add membership to.

Example

```
await Contacts.addExistingContactToGroupAsync("665FDBCFAE55-D614-4A15-8DC6-161A368D", "161A368D-D614-4A15-8DC6-665FDBCFAE55");
```

removeContactFromGroupAsync

```
removeContactFromGroupAsync(contactId: string, groupId: string): Promise<any>
```

Remove a contact's membership from a given group. This will not delete the contact.

Parameters

Name	Type	Description
contactId	string	ID of the contact you want to remove.
groupId	string	ID for the group you want to remove membership of.

Example

```
await Contacts.removeContactFromGroupAsync("665FDBCFAE55-D614-4A15-8DC6-161A368D", "161A368D-D614-4A15-8DC6-665FDBCFAE55");
```

getGroupsAsync

```
getGroupsAsync(query: GroupQuery): Promise<Group[]>
```

Query and return a list of system groups.

Parameters

Name	Type	Description
query	GroupQuery	Information regarding which groups you want to get.

Returns

Name	Type	Description
groups	Group[]	Collection of groups that fit query.

Example

```
const groups = await Contacts.getGroupsAsync({ groupName: "sailor moon" });
const allGroups = await Contacts.getGroupsAsync({});
```

getDefaultContainerIdAsync

```
getDefaultContainerIdAsync(): Promise<string>
```

Get the default container's ID.

Returns

Name	Type	Description
containerId	string	Default container ID.

Example

```
const containerId = await Contacts.getDefaultContainerIdAsync();
```

getContainersAsync

```
getContainersAsync(containerQuery: ContainerQuery): Promise<Container[]>
```

Query a list of system containers.

Parameters

Name	Type	Description
containerQuery	ContainerQuery	Information used to gather containers.

Returns

Name	Type	Description
containerId	string	Collection of containers that fit query.

Example

```
const allContainers = await getContainersAsync({ contactId: "665FDDBCFAE55-D614-4A15-8DC6-161A368D" });
```

Types

Contact

A set of fields that define information about a single entity.

Name	Type	Description	iOS	Android
id	string	Immutable identifier used for querying and indexing.		
name	string	Full name with proper format.		
firstName	string	Given name.		
middleName	string	Middle name.		
lastName	string	Family name.		
maidenName	string	Maiden name.		
namePrefix	string	Dr. Mr. Mrs. Ect...		
nameSuffix	string	Jr. Sr. Ect...		
nickname	string	An alias to the proper name.		
phoneticFirstName	string	Pronunciation of the first name.		
phoneticMiddleName	string	Pronunciation of the middle name.		
phoneticLastName	string	Pronunciation of the last name.		
company	string	Organization the entity belongs to.		
jobTitle	string	Job description.		
department	string	Job department.		
note	string	Additional information.		
imageAvailable	boolean	Used for efficient retrieval of images.		
image	Image	Thumbnail image (ios: 320x320)		
rawImage	Image	Raw image without cropping, usually large.		
		Denoting a person or		

contactType		company.		
birthday	Date	Birthday information in JS format.		
dates	Date[]	A list of other relevant user dates.		
relationships	Relationship[]	Names of other relevant user connections		
emails	Email[]	Email addresses		
phoneNumbers	PhoneNumber[]	Phone numbers		
addresses	Address[]	Locations		
instantMessageAddresses	InstantMessageAddress[]	IM connections		
urlAddresses	UrlAddress[]	Web URLs		
nonGregorianBirthday	Date	Birthday that doesn't conform to the Gregorian calendar format		
socialProfiles	SocialProfile[]	Social networks		
thumbnail	Image	Deprecated: Use <code>image</code>		
previousLastName	string	Deprecated: Use <code>maidenName</code>		

Group

iOS Only

A parent to contacts. A contact can belong to multiple groups. To get a group's children you can query with

```
getContactsAsync({ groupId })
```

Here are some different query operations:

- Child Contacts: `getContactsAsync({ groupId })`
- Groups From Container: `getGroupsAsync({ containerId })`
- Groups Named: `getContainersAsync({ groupName })`

Name	Type	Description
id	string	Immutable id representing the group
name	string	The editable name of a group

Container

iOS Only

A parent to contacts and groups. You can query the default container with `getDefaultContainerIdAsync()`. Here are some different query operations:

- Child Contacts: `getContactsAsync({ containerId })`
- Child Groups: `getGroupsAsync({ containerId })`
- Container from Contact: `getContainersAsync({ contactId })`

- Container from Group: `getContainersAsync({ groupId })`
- Container from ID: `getContainersAsync({ containerId })`

Name	Type	Description
id	string	Immutable id representing the group
name	string	The editable name of a group

Date

Name	Type	Description
day	number	Day.
month	number	Month - adjusted for JS <code>Date</code> which starts at 0.
year	number	Year.
format	CalendarFormatType	Format for input date.
id	string	Unique ID.
label	string	Localized display name.

Relationship

Name	Type	Description
name	string	Name of related contact.
id	string	Unique ID.
label	string	Localized display name.

Email

Name	Type	Description
email	string	email address.
isPrimary	boolean	Primary email address.
id	string	Unique ID.
label	string	Localized display name.

PhoneNumber

Name	Type	Description
number	string	Phone number.
isPrimary	boolean	Primary phone number.
digits	string	Phone number without format, ex: 8674305.
countryCode	string	Country code, ex: +1.

<code>id</code>	<code>string</code>	Unique ID.
<code>label</code>	<code>string</code>	Localized display name.

Address

Name	Type	Description
<code>street</code>	<code>string</code>	Street name.
<code>city</code>	<code>string</code>	City name.
<code>country</code>	<code>string</code>	Country name.
<code>region</code>	<code>string</code>	Region or state name.
<code>neighborhood</code>	<code>string</code>	Neighborhood name.
<code>postalCode</code>	<code>string</code>	Local post code.
<code>poBox</code>	<code>string</code>	P.O. Box.
<code>isoCountryCode</code>	<code>string</code>	Standard code.
<code>id</code>	<code>string</code>	Unique ID.
<code>label</code>	<code>string</code>	Localized display name.

SocialProfile

iOS Only

Name	Type	Description
<code>service</code>	<code>string</code>	Name of social app.
<code>username</code>	<code>string</code>	Username in social app.
<code>localizedProfile</code>	<code>string</code>	Localized name.
<code>url</code>	<code>string</code>	Web URL.
<code>userId</code>	<code>string</code>	UID for social app.
<code>id</code>	<code>string</code>	Unique ID.
<code>label</code>	<code>string</code>	Localized display name.

InstantMessageAddress

Name	Type	Description
<code>service</code>	<code>string</code>	Name of social app.
<code>username</code>	<code>string</code>	Username in IM app.
<code>localizedService</code>	<code>string</code>	Localized name of app.
<code>id</code>	<code>string</code>	Unique ID.
<code>label</code>	<code>string</code>	Localized display name.

UrlAddress

Name	Type	Description
url	string	Web URL
id	string	Unique ID.
label	string	Localized display name.

Image

Information regarding thumbnail images.

Name	Type	iOS	Android
uri	string		
width	number		
height	number		
base64	string		

Android: You can get dimensions using `ReactNative.Image.getSize`. Avoid using Base 64 in React Native

FormOptions

Denotes the functionality of a native contact form.

Name	Type	Description
displayedPropertyKeys	FieldType[]	The properties that will be displayed. iOS: Does nothing in editing mode.
message	string	Controller title.
alternateName	string	Used if contact doesn't have a name defined.
cancelButtonTitle	string	The name of the left bar button.
groupId	string	The parent group for a new contact.
allowsEditing	boolean	Allows for contact mutation.
allowsActions	boolean	Actions like share, add, create.
shouldShowLinkedContacts	boolean	Shows similar contacts.
isNew	boolean	Present the new contact controller - if false the unknown controller will be shown.
preventAnimation	boolean	Prevents the controller from animating in.

ContactQuery

Used to query contacts from the user's device.

Name	Type	Description	iOS	Android
		If available the fields defined will be returned. If		

fields	<code>FieldType[]</code>	<code>nil</code> then all fields will be returned.		
pageSize	<code>number</code>	The max number of contacts to return. If <code>nil</code> or <code>0</code> then all contacts will be returned.		
pageOffset	<code>number</code>	The number of contacts to skip before gathering contacts.		
id	<code>string</code>	Get contacts with a matching ID .		
sort	<code>SortType</code>	Sort method used when gathering contacts.		
name	<code>string</code>	Query contacts matching this name.		
groupId	<code>string</code>	Get all contacts that belong to the group matching this ID.		
containerId	<code>string</code>	Get all contacts that belong to the container matching this ID.		
rawContacts	<code>boolean</code>	Prevent unification of contacts when gathering. Default: <code>false</code> .		

GroupQuery

iOS Only

Used to query native contact groups.

Name	Type	Description
groupName	<code>string</code>	Query all groups matching a name.
groupId	<code>string</code>	Query the group with a matching ID.
containerId	<code>string</code>	Query all groups that belong to a certain container.

ContainerQuery

iOS Only

Used to query native contact containers.

Name	Type	Description
contactId	<code>string</code>	Query all the containers that parent a contact.
groupId	<code>string</code>	Query all the containers that parent a group.
containerId	<code>string</code>	Query a container from it's ID.

ContactResponse

The return value for queried contact operations like `getContactsAsync` .

Name	Type	Description
data	<code>Contact[]</code>	An array of contacts that match a particular query.
hasNextPage	<code>boolean</code>	This will be true if there are more contacts to retrieve beyond what is returned.

hasPreviousPage	boolean	true if there are previous contacts that weren't retrieved due to <code>pageOffset</code> .
total	number	Deprecated: use <code>data.length</code> to get the number of contacts returned.

Constants

Field

```
const contactField = Contact.Fields.FirstName;
```

Name	Value	iOS	Android
ID	'id'		
Name	'name'		
FirstName	'firstName'		
MiddleName	'middleName'		
LastName	'lastName'		
NamePrefix	'namePrefix'		
NameSuffix	'nameSuffix'		
PhoneticFirstName	'phoneticFirstName'		
PhoneticMiddleName	'phoneticMiddleName'		
PhoneticLastName	'phoneticLastName'		
Birthday	'birthday'		
Emails	'emails'		
PhoneNumbers	'phoneNumbers'		
Addresses	'addresses'		
InstantMessageAddresses	'instantMessageAddresses'		
UrlAddresses	'urlAddresses'		
Company	'company'		
JobTitle	'jobTitle'		
Department	'department'		
ImageAvailable	'imageAvailable'		
Image	'image'		
Note	'note'		
Dates	'dates'		
Relationships	'relationships'		
Nickname	'nickname'		

RawImage	'rawImage'		
MaidenName	'maidenName'		
ContactType	'contactType'		
SocialProfiles	'socialProfiles'		
NonGregorianBirthday	'nonGregorianBirthday'		
Thumbnail	Deprecated: use <code>Image</code>		
PreviousLastName	Deprecated: use <code>MaidenName</code>		

FormType

```
const formType = Contacts.FormTypes.New;
```

Name	Value	Description
New	'new'	Creating a contact
Unknown	'unknown'	Displaying a contact with actions
Default	'default'	Information regarding a contact

ContactType

iOS Only

```
const contactType = Contacts.ContactTypes.Person;
```

Name	Value	Description
Person	'person'	Contact is a human
Company	'company'	Contact is group or company

SortType

```
const sortType = Contacts.SortTypes.FirstName;
```

Name	Value	Description	iOS	Android
FirstName	'firstName'	Sort by first name in ascending order		
LastName	'lastName'	Sort by last name in ascending order		
UserDefault	'userDefault'	The user default method of sorting		

ContainerType

iOS Only

```
const containerType = Contacts.ContainerTypes.CardDAV;
```

Name	Value	Description
Local	'local'	A local non-iCloud container
Exchange	'exchange'	In association with Email
CardDAV	'cardDAV'	cardDAV protocol used for sharing
Unassigned	'unassigned'	Unknown

CalendarFormat

```
const calendarFormat = Contacts.CalendarFormats.Coptic;
```

This format denotes the common calendar format used to specify how a date is calculated in `nonGregorianBirthday` fields.

Constant	value	iOS	Android
Gregorian	'gregorian'		
Chinese	'chinese'		
Hebrew	'hebrew'		
Islamic	'islamic'		

Contact Fields

Deprecated: Use `Contacts.Fields`

This table illustrates what fields will be added on demand to every contact.

Constant	value	iOS	Android
PHONE_NUMBERS	'phoneNumbers'		
EMAILS	'emails'		
ADDRESSES	'addresses'		
IMAGE	'image'		
NOTE	'note'		
NAME_PREFIX	'namePrefix'		
NAME_SUFFIX	'nameSuffix'		
PHONETIC_FIRST_NAME	'phoneticFirstName'		
PHONETIC_MIDDLE_NAME	'phoneticMiddleName'		
PHONETIC_LAST_NAME	'phoneticLastName'		
IM_ADDRESSES	'instantMessageAddresses'		
URLS	'urlAddresses'		
DATES	'dates'		

NON_GREGORIAN_BIRTHDAY	'nonGregorianBirthday'		
SOCIAL_PROFILES	'socialProfiles'		
RAW_IMAGE	'rawImage'		
THUMBNAIL	'thumbnail'	Deprecated	Deprecated
PREVIOUS_LAST_NAME	'previousLastName'	Deprecated	Deprecated

Breaking Changes

SDK 29

- The `thumnail` field has been deprecated, use `image` on both platforms instead.
- On iOS `image` is now `rawImage`. There is no Android version of `rawImage`.
- Images now return a localUri instead of Base64 string.
- Base64 string is now returned in a encodable format.
- Empty contact fields will no longer be returned as empty strings on iOS.
- Passing no fields will now return all contact information.

DeviceMotion

Access the device motion and orientation sensors. All data is presented in terms of three axes that run through a device. According to portrait orientation: X runs from left to right, Y from bottom to top and Z perpendicularly through the screen from back to front.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { DangerZone } from 'expo';  
const { DeviceMotion } = DangerZone;  
  
// in bare apps:  
import { DeviceMotion } from 'expo-sensors';
```

DeviceMotion.isAvailableAsync()

Returns whether device motion is enabled on the device.

Returns

- A promise that resolves to a `boolean` denoting the availability of the sensor.

DeviceMotion.addListener(listener)

Subscribe for updates to DeviceMotion.

Arguments

- listener (function)** -- A callback that is invoked when a DeviceMotion update is available. When invoked, the listener is provided a single argument that is an object containing following fields:
 - acceleration (object)** -- Device acceleration on the three axis as an object with x, y, z keys. Expressed in m/s².
 - accelerationIncludingGravity (object)** -- Device acceleration with the effect of gravity on the three axis as an object with x, y, z keys. Expressed in m/s².
 - rotation (object)** -- Device's orientation in space as an object with alpha, beta, gamma keys where alpha is for rotation around Z axis, beta for X axis rotation and gamma for Y axis rotation.
 - rotationRate (object)** -- Rotation rates of the device around each of its axes as an object with alpha, beta, gamma keys where alpha is around Z axis, beta for X axis and gamma for Y axis.

- **orientation (number)** -- Device orientation based on screen rotation. Value is one of `0` (portrait), `90` (right landscape), `180` (upside down), `-90` (left landscape).

Returns

- A subscription that you can call `remove()` on when you would like to unsubscribe the listener.

DeviceMotion.removeAllListeners()

Remove all listeners.

DeviceMotion.setUpdateInterval(intervalMs)

Subscribe for updates to DeviceMotion.

Arguments

- **intervalMs (number)** Desired interval in milliseconds between DeviceMotion updates.

DocumentPicker

Provides access to the system's UI for selecting documents from the available providers on the user's device.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { DocumentPicker } from 'expo';  
  
// in bare apps:  
import * as DocumentPicker from 'expo-document-picker';
```

DocumentPicker.getDocumentAsync(options)

Display the system UI for choosing a document. By default, the chosen file is copied to the app's internal cache directory.

Arguments

- **options (object)** --

A map of options:

- **type (string)** -- The [MIME type](#) of the documents that are available to be picked. Is also supports wildcards like `image/*` to choose any image. To allow any type of document you can use `/*`. Defaults to `/*`.
- **copyToCacheDirectory (boolean)** -- If `true`, the picked file is copied to `FileSystem.CacheDirectory`, which allows other Expo APIs to read the file immediately. Defaults to `true`. This may impact performance for large files, so you should consider setting this to `false` if you expect users to pick particularly large files and your app does not need immediate read access.

Returns

If the user cancelled the document picking, returns `{ type: 'cancel' }`.

Otherwise, returns `{ type: 'success', uri, name, size }` where `uri` is a URI to the local document file, `name` is its original name and `size` is its size in bytes.

iOS configuration

On iOS, for [standalone apps](#) and [ExpoKit](#) projects, the DocumentPicker module requires the iCloud entitlement to work properly. You need to set the `useIcloudStorage` key to `true` in your `app.json` file as specified [here](#).

iCloud Application Service

In addition, you'll also need to enable the iCloud Application Service in your App identifier. This can be done in the detail of your [App ID in the Apple developer interface](#).

Enable iCloud service with CloudKit support, create one iCloud Container, and name it `icloud`.

`<your_bundle_identifier>` .

And finally, to apply those changes, you'll need to revoke your existing provisioning profile and run `expo build:ios -c`

For ExpoKit apps, you need to open the project in Xcode and follow the [Using DocumentPicker instructions](#) in the Advanced ExpoKit Topics guide.

ErrorRecovery

Utilities for helping you gracefully handle crashes due to fatal JavaScript errors.

Installation

This API is pre-installed in [managed](#) apps. It is not available to [bare](#) React Native apps.

API

```
import { ErrorRecovery } from 'expo';
```

ErrorRecovery.setRecoveryProps(props)

Set arbitrary error recovery props. If your project crashes in production as a result of a fatal JS error, Expo will reload your project. If you've set these props, they'll be passed to your reloaded project's initial props under `exp.errorRecovery`. [Read more about error handling with Expo](#).

Arguments

- **props (object)** -- An object which will be passed to your reloaded project's initial props if the project was reloaded as a result of a fatal JS error.

```
#
```

Facebook

Provides Facebook integration for Expo apps. Expo exposes a minimal native API since you can access Facebook's [Graph API](#) directly through HTTP (using `fetch`, for example).

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Configuration

Registering your app with Facebook

Follow [Facebook's developer documentation](#) to register an application with Facebook's API and get an application ID. Take note of this application ID because it will be used as the `appId` option in your `Facebook.logInWithReadPermissionsAsync` call. Then follow these steps based on the platforms you're targetting. This will need to be done from the [Facebook developer site](#):

- **The Expo client app**

- Add `host.exp.Exponent` as an iOS *Bundle ID*. Add `rRW++LUjmZZ+58EbN5DVhGAnkX4=` as an Android *key hash*. Your app's settings should end up including the following under "Settings > Basic":

The screenshot shows the Facebook Developer Settings interface. On the left, a sidebar lists options like Dashboard, Settings (selected), Basic (highlighted), Advanced, Roles, Alerts, App Review, PRODUCTS, Facebook Login, and + Add Product. The main area is divided into two sections: iOS and Android.

iOS Configuration:

- Bundle ID: host.exp.Exponent
- iPhone Store ID: The ID to identify your app in the iOS Store
- URL Scheme Suffix (Optional):
- iPad Store ID: The ID to identify your app in the iPad Store
- Single Sign On: No (Will launch from iOS Notifications)
- iOS Only: Log In-App Purchase Events Automatically (Recommended): A note explaining the feature and a link to learn more.

Android Configuration:

- Google Play Package Name: Unique app identifier used to open your app
- Class Name: The Main Activity you want Facebook to launch
- Key Hashes: rRW++LUjmZZ+58EbN5DVhGAnkX4=
- Amazon Appstore URL (Optional): Ex. http://www.amazon.com/dp/B004GJDQT8
- Single Sign On: No (Will launch from Android Notifications)

- **iOS standalone app**

- Add your app's Bundle ID as a *Bundle ID* in app's settings page pictured above.
- In your `app.json`, add a field `facebookScheme` with your Facebook login redirect URL scheme found [here](#) under 4. *Configure Your info.plist*. It should look like `"fb123456"`.
- Also in your `app.json`, add your [Facebook App ID](#) and [Facebook Display Name](#) under the `facebookAppId` and `facebookDisplayName` keys.

- **Android standalone app**

- [Build your standalone app](#) for Android.
- Run `expo fetch:android:hashes`.
- Copy `Facebook Key Hash` and paste it as an additional key hash in your Facebook developer page pictured above.

You may have to switch the app from 'development mode' to 'public mode' on the Facebook developer page before other users can log in.

API

```
// in managed apps:  
import { Facebook } from 'expo';  
  
// in bare apps:  
import * as Facebook from 'expo-facebook';
```

Facebook.logInWithReadPermissionsAsync(appId, options)

Prompts the user to log into Facebook and grants your app permission to access their Facebook data.

param string appId

Your Facebook application ID. [Facebook's developer documentation](#) describes how to get one.

param object options

A map of options:

- **permissions (array)** -- An array specifying the permissions to ask for from Facebook for this login. The permissions are strings as specified in the [Facebook API documentation](#). The default permissions are `['public_profile', 'email']`.
- **behavior (string)** -- The type of login prompt to show. Currently this is only supported on iOS, and must be one of the following values:
 - `'system'` (default) -- Attempts to log in through the Facebook account currently signed in through the device Settings. This will fallback to `native` behavior on iOS 11+ as Facebook has been removed from iOS's Settings.
 - `'web'` -- Attempts to log in through a modal `UIWebView` pop up.
 - `'browser'` -- Attempts to log in through Safari or `SFSafariViewController`.
 - `'native'` -- Attempts to log in through the native Facebook app, but the Facebook SDK may fallback to `browser` instead.

Returns

If the user or Facebook cancelled the login, returns `{ type: 'cancel' }`.

Otherwise, returns `{ type: 'success', token, expires, permissions, declinedPermissions }`. `token` is a string giving the access token to use with Facebook HTTP API requests. `expires` is the time at which this token will expire, as seconds since epoch. You can save the access token using, say, `AsyncStorage`, and use it till the expiration time. `permissions` is a list of all the approved permissions, whereas `declinedPermissions` is a list of the permissions that the user has rejected.

Example

```
async function logIn() {
  try {
    const {
      type,
      token,
      expires,
      permissions,
      declinedPermissions,
    } = await Facebook.logInWithReadPermissionsAsync('<APP_ID>', {
      permissions: ['public_profile'],
    });
    if (type === 'success') {
      // Get the user's name using Facebook's Graph API
      const response = await fetch(`https://graph.facebook.com/me?access_token=${token}`);
      Alert.alert('Logged in!', `Hi ${await response.json().name}`);
    } else {
      // type === 'cancel'
    }
  } catch ({ message }) {
    alert(`Facebook Login Error: ${message}`);
  }
}
```

Given a valid Facebook application ID in place of `<APP_ID>`, the code above will prompt the user to log into Facebook then display the user's name. This uses React Native's `fetch` to query Facebook's [Graph API](#).

#

FacebookAds

Facebook Audience SDK integration.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Configuration

Creating the placement ID

You need to create a placement ID to display ads. Follow steps 1 and 3 from the [Getting Started Guide for Facebook Audience](#) to create the placement ID.

Configuring app.json

In your project's `app.json`, add your [Facebook App ID](#) and [Facebook Display Name](#) under the `facebookAppId` and `facebookDisplayName` keys.

Development vs Production

When using Facebook Ads in development, you'll need to register your device to be able to show ads. You can add the following at the top of your file to register your device:

```
import { FacebookAds } from 'expo';

FacebookAds.AdSettings.addTestDevice(FacebookAds.AdSettings.currentDeviceHash);
```

You should see fake ads after you add this snippet.

To use Facebook Ads in production with real ads, you need to publish your app on Play Store or App Store and add your app in the Facebook console. Refer the [Submit Your App for Review](#) section in the [Getting Started Guide](#) for more details.

Usage

Interstitial Ads

Interstitial Ad is a type of ad that displays a full-screen modal dialog with media content. It has a dismiss button as well as a touchable area that takes the user outside of your app to the advertised content.

Example:

```
import { FacebookAds } from 'expo';

FacebookAds.InterstitialAdManager.showAd(placementId)
```

```
.then(didClick => {})
.catch(error => {})
```

The method returns a promise that will be rejected when an error occurs during a call (e.g. no fill from ad server or network error) and resolved when the user either dismisses or interacts with the displayed ad.

Native Ads

Native ads can be customized to match the design of your app. To display a native ad, you need to:

1. Create NativeAdsManager instance

The `NativeAdManager` is responsible for fetching and caching ads as you request them.

```
import { FacebookAds } from 'expo';

const adsManager = new FacebookAds.NativeAdsManager(placementId, numberOfAdsToRequest);
```

The constructor accepts two parameters:

- `placementId` - which is a unique identifier describing your ad units
- `numberOfAdsToRequest` - which is a number of ads to request by ads manager at a time

2. Wrap your component with `withNativeAd` HOC

Next, you need to wrap the component you want to use to show your add with the `withNativeAd` higher-order component. The wrapped component will receive a prop named `nativeAd`, which you can use to render an ad.

```
import { FacebookAds } from 'expo';

class AdComponent extends React.Component {
  render() {
    return (
      <View>
        <Text>{this.props.nativeAd.bodyText}</Text>
      </View>
    );
  }
}

export default FacebookAds.withNativeAd(AdComponent);
```

The `nativeAd` object can contain the following properties:

- `advertiserName` - The name of the Facebook Page or mobile app that represents the business running each ad.
- `headline` - The headline that the advertiser entered when they created their ad. This is usually the ad's main title.
- `linkDescription` - Additional information that the advertiser may have entered.
- `adTranslation` - The word 'ad', translated into the language based upon Facebook app language setting.
- `promotedTranslation` - The word 'promoted', translated into the language based upon Facebook app language setting.
- `sponsoredTranslation` - The word 'sponsored', translated into the language based upon Facebook app language setting.

- `bodyText` - Ad body
- `callToActionText` - Call to action phrase, e.g. - "Install Now"
- `socialContext` - social context for the Ad, for example "Over half a million users"

More information on how the properties correspond to an exemplary ad can be found in the official Facebook documentation for [Android](#) and for [iOS](#).

3. Add `AdMediaView` and `AdIconView` components

`AdMediaView` displays native ad media content whereas `AdIconView` is responsible for displaying an ad icon.

Note: Don't use more than one `AdMediaView` and `AdIconView` component (each) within one native ad. If you use more, only the last mounted one will be populated with ad content.

```
import { FacebookAds } from 'expo';
const { AdIconView, AdMediaView } = FacebookAds;

class AdComponent extends React.Component {
  render() {
    return (
      <View>
        <AdMediaView />
        <AdIconView />
      </View>
    );
  }
}

export default FacebookAds.withNativeAd(AdComponent);
```

4. Mark which components trigger the ad

Note: In order for elements wrapped with `AdTriggerView` to trigger the ad, you also must include `AdMediaView` in the children tree.

```
import { FacebookAds } from 'expo';
const { AdTriggerView, AdMediaView } = FacebookAds;

class AdComponent extends React.Component {
  render() {
    return (
      <View>
        <AdMediaView />
        <AdTriggerView>
          <Text>{this.props.nativeAd.bodyText}</Text>
        </AdTriggerView>
      </View>
    );
  }
}

export default FacebookAds.withNativeAd(AdComponent);
```

5. Render the ad component

Now you can render the wrapped component and pass the `adsManager` instance you created earlier.

```

class MyApp extends React.Component {
  render() {
    return (
      <View>
        <AdComponent adsManager={adsManager} />
      </View>
    );
  }
}

```

BannerAd

The `BannerAd` component allows you to display native ads as banners (known as *AdView*).

Banners are available in 3 sizes:

- `standard` (`BANNER_HEIGHT_50`)
- `large` (`BANNER_HEIGHT_90`)
- `rectangle` (`RECTANGLE_HEIGHT_250`)

1. Showing ad

In order to show an ad, you first have to import `BannerAd` from the package:

```

import { FacebookAds } from 'expo';

function ViewWithBanner(props) {
  return (
    <View>
      <FacebookAds.BannerAd
        placementId="YOUR_BANNER_PLACEMENT_ID"
        type="standard"
        onPress={() => console.log('click')}
        onError={(error) => console.log('error', error)}
      />
    </View>
  );
}

```

API

```

// in managed apps:
import { FacebookAds } from 'expo';

// in bare apps:
import * as FacebookAds from 'expo-ads-facebook';

```

NativeAdsManager

A wrapper for `FBNativeAdsManager`. It provides a mechanism to fetch a set of ads and use them.

disableAutoRefresh

By default the native ads manager will refresh its ads periodically. This does not mean that any ads which are shown in the application's UI will be refreshed, but requesting next native ads to render may return new ads at different times.

```
adsManager.disableAutoRefresh();
```

setMediaCachePolicy (iOS)

This controls which media from the native ads are cached before being displayed. The default is to not block on caching.

```
adsManager.setMediaCachePolicy('none' | 'icon' | 'image' | 'all');
```

Note: This method is a no-op on Android

InterstitialAdManager

InterstitialAdManager is a manager that allows you to display interstitial ads within your app.

showAd

Shows a fullscreen interstitial ad asynchronously.

```
InterstitialAdManager.showAd('placementId')
  .then(...)
  .catch(...);
```

Promise will be rejected when there's an error loading ads from Facebook Audience network. It will resolve with a `boolean` indicating whether user didClick an ad or not.

Note: There can be only one `showAd` call being performed at a time. Otherwise, an error will be thrown.

AdSettings

AdSettings contains global settings for all ad controls.

currentDeviceHash

Constant which contains current device's hash.

addTestDevice

Registers given device to receive test ads. When you run app on simulator, it should automatically get added. Use it to receive test ads in development mode on a standalone phone.

All devices should be specified before any other action takes place, like `AdsManager` gets created.

```
FacebookAds.AdSettings.addTestDevice('hash');
```

clearTestDevices

Clears all previously set test devices. If you want your ads to respect newly set config, you'll have to destroy and create an instance of AdsManager once again.

```
FacebookAds.AdSettings.clearTestDevices();
```

setLogLevel (iOS)

Sets current SDK log level.

```
FacebookAds.AdSettings.setLogLevel('none' | 'debug' | 'verbose' | 'warning' | 'error' | 'notification');
```

Note: This method is a no-op on Android.

setIsChildDirected

Configures the ad control for treatment as child-directed.

```
FacebookAds.AdSettings.setIsChildDirected(true | false);
```

setMediationService

If an ad provided service is mediating Audience Network in their SDK, it is required to set the name of the mediation service

```
FacebookAds.AdSettings.setMediationService('foobar');
```

setUrlPrefix

Sets the URL prefix to use when making ad requests.

```
FacebookAds.AdSettings.setUrlPrefix('...');
```

Note: This method should never be used in production

```
#
```

FaceDetector

`FaceDetector` lets you use the power of [Google Mobile Vision](#) framework to detect faces on images.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

Known issues

- Android does not recognize faces that aren't aligned with the interface (top of the interface matches top of the head).

Comprehensive Example

Check out a full example at [expo/camerja](#). You can try it with Expo at [@community/camerja](#).

`FaceDetector` is used in Gallery screen — it should detect faces on saved photos and show the probability that the face is smiling.

Intermodule interface

Other modules, like eg. [Camera](#) are able to use this `FaceDetector`.

API

```
// in managed apps:  
import { FaceDetector } from 'expo';  
  
// in bare apps:  
import * as FaceDetector from 'expo-face-detector';
```

Settings

In order to configure detector's behavior modules pass a settings object which is then interpreted by this module. The shape of the object should be as follows:

- **mode?** ([FaceDetector.Constants.Mode](#)) -- Whether to detect faces in fast or accurate mode. Use `FaceDetector.Constants.Mode.{fast, accurate}`.
- **detectLandmarks?** ([FaceDetector.Constants.Landmarks](#)) -- Whether to detect and return landmarks positions on the face (ears, eyes, mouth, cheeks, nose). Use `FaceDetector.Constants.Landmarks.{all, none}`.
- **runClassifications?** ([FaceDetector.Constants.Classifications](#)) -- Whether to run additional classifications on detected faces (smiling probability, open eye probabilities). Use `FaceDetector.Constants.Classifications.{all, none}`.

Eg. you could use the following snippet to detect faces in fast mode without detecting landmarks or whether face is smiling:

```
import { FaceDetector } from 'expo';

<Camera
  // ... other props
  onFacesDetected={this.handleFacesDetected}
  faceDetectorSettings={{
    mode: FaceDetector.Constants.Mode.fast,
    detectLandmarks: FaceDetector.Constants.Landmarks.none,
    runClassifications: FaceDetector.Constants.Classifications.none,
  }}
/>
```

Event shape

While detecting faces, `FaceDetector` will emit object events of the following shape:

- **faces (array)** - array of faces objects:
 - **faceID (number)** -- a face identifier (used for tracking, if the same face appears on consecutive frames it will have the same `faceID`).
 - **bounds (object)** -- an object containing:
 - **origin ({ x: number, y: number })** -- position of the top left corner of a square containing the face in view coordinates,
 - **size ({ width: number, height: number })** -- size of the square containing the face in view coordinates,
 - **rollAngle (number)** -- roll angle of the face (bank),
 - **yawAngle (number)** -- yaw angle of the face (heading, turning head left or right),
 - **smilingProbability (number)** -- probability that the face is smiling,
 - **leftEarPosition ({ x: number, y: number })** -- position of the left ear in view coordinates,
 - **rightEarPosition ({ x: number, y: number })** -- position of the right ear in view coordinates,
 - **leftEyePosition ({ x: number, y: number })** -- position of the left eye in view coordinates,
 - **leftEyeOpenProbability (number)** -- probability that the left eye is open,
 - **rightEyePosition ({ x: number, y: number })** -- position of the right eye in view coordinates,
 - **rightEyeOpenProbability (number)** -- probability that the right eye is open,
 - **leftCheekPosition ({ x: number, y: number })** -- position of the left cheek in view coordinates,
 - **rightCheekPosition ({ x: number, y: number })** -- position of the right cheek in view coordinates,
 - **mouthPosition ({ x: number, y: number })** -- position of the center of the mouth in view coordinates,
 - **leftMouthPosition ({ x: number, y: number })** -- position of the left edge of the mouth in view coordinates,
 - **rightMouthPosition ({ x: number, y: number })** -- position of the right edge of the mouth in view coordinates,
 - **noseBasePosition ({ x: number, y: number })** -- position of the nose base in view coordinates.

`smilingProbability` , `leftEyeOpenProbability` and `rightEyeOpenProbability` are returned only if `faceDetectionClassifications` property is set to `.all` .

Positions of face landmarks are returned only if `faceDetectionLandmarks` property is set to `.all` .

Methods

To use methods that `FaceDetector` exposes one just has to import the module. (In ejected apps on iOS face detection will be supported only if you add the `FaceDetector` subspec to your project. Refer to [Adding the Payments Module on iOS](#) for an example of adding a subspec to your ejected project.)

```
import { FaceDetector } from 'expo';

// ...
detectFaces = async (imageUri) => {
  const options = { mode: FaceDetector.Constants.Mode.fast };
  return await FaceDetector.detectFacesAsync(imageUri, options);
};

// ...
```

detectFacesAsync

Detect faces on a picture.

Arguments

- **uri (string)** -- `file://` URI to the image.
- **options? (object)** -- A map of options:
 - **mode? (FaceDetector.Constants.Mode)** -- Whether to detect faces in fast or accurate mode. Use `FaceDetector.Constants.Mode.{fast, accurate}`.
 - **detectLandmarks? (FaceDetector.Constants.Landmarks)** -- Whether to detect and return landmarks positions on the face (ears, eyes, mouth, cheeks, nose). Use `FaceDetector.Constants.Landmarks.{all, none}`.
 - **runClassifications? (FaceDetector.Constants.Classifications)** -- Whether to run additional classifications on detected faces (smiling probability, open eye probabilities). Use `FaceDetector.Constants.Classifications.{all, none}`.

Returns

Returns a Promise that resolves to an object: `{ faces, image }` where `faces` is an array of the detected faces and `image` is an object containing `uri: string` of the image, `width: number` of the image in pixels, `height: number` of the image in pixels and `orientation: number` of the image (value conforms to the EXIF orientation tag standard).

Detected face schema

A detected face is an object containing at most following fields:

- **bounds (object)** -- an object containing:
 - **origin ({ x: number, y: number })** -- position of the top left corner of a square containing the face in image coordinates,
 - **size ({ width: number, height: number })** -- size of the square containing the face in image coordinates,
- **rollAngle (number)** -- roll angle of the face (bank),
- **yawAngle (number)** -- yaw angle of the face (heading, turning head left or right),
- **smilingProbability (number)** -- probability that the face is smiling,
- **leftEarPosition ({ x: number, y: number })** -- position of the left ear in image coordinates,
- **rightEarPosition ({ x: number, y: number })** -- position of the right ear in image coordinates,
- **leftEyePosition ({ x: number, y: number })** -- position of the left eye in image coordinates,
- **leftEyeOpenProbability (number)** -- probability that the left eye is open,
- **rightEyePosition ({ x: number, y: number })** -- position of the right eye in image coordinates,

- **rightEyeOpenProbability (number)** -- probability that the right eye is open,
- **leftCheekPosition ({ x: number, y: number})** -- position of the left cheek in image coordinates,
- **rightCheekPosition ({ x: number, y: number})** -- position of the right cheek in image coordinates,
- **mouthPosition ({ x: number, y: number})** -- position of the center of the mouth in image coordinates,
- **leftMouthPosition ({ x: number, y: number})** -- position of the left edge of the mouth in image coordinates,
- **rightMouthPosition ({ x: number, y: number})** -- position of the right edge of the mouth in image coordinates,
- **noseBasePosition ({ x: number, y: number})** -- position of the nose base in image coordinates.

`smilingProbability`, `leftEyeOpenProbability` and `rightEyeOpenProbability` are returned only if `runClassifications` option is set to `.all`.

Positions of face landmarks are returned only if `detectLandmarks` option is set to `.all`.

#

FileSystem

Provides access to a file system stored locally on the device. Within the Expo client, each app has a separate file systems and has no access to the file system of other Expo apps.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { FileSystem } from 'expo';  
  
// in bare apps:  
import * as FileSystem from 'expo-file-system';
```

The API takes `file://` URIs pointing to local files on the device to identify files. Each app only has read and write access to locations under the following directories:

- `FileSystem.documentDirectory`

`file://` URI pointing to the directory where user documents for this app will be stored. Files stored here will remain until explicitly deleted by the app. Ends with a trailing `/`. Example uses are for files the user saves that they expect to see again.

- `FileSystem.cacheDirectory`

`file://` URI pointing to the directory where temporary files used by this app will be stored. Files stored here may be automatically deleted by the system when low on storage. Example uses are for downloaded or generated files that the app just needs for one-time usage.

So, for example, the URI to a file named `'myFile'` under `'myDirectory'` in the app's user documents directory would be `FileSystem.documentDirectory + 'myDirectory/myFile'`.

Expo APIs that create files generally operate within these directories. This includes `Audio` recordings, `Camera` photos, `ImagePicker` results, `SQLite` databases and `takeSnapShotAsync()` results. This allows their use with the `FileSystem` API.

Some `FileSystem` functions are able to read from (but not write to) other locations. Currently `FileSystem.getInfoAsync()` and `FileSystem.copyAsync()` are able to read from URIs returned by `CameraRoll.getPhotos()` from React Native.

- `FileSystem.EncodingTypes`

These constants can be used to define how data is read / written.

- `FileSystem.EncodingTypes.UTF8` -- Standard readable format.
- `FileSystem.EncodingTypes.Base64` -- Binary, radix-64 representation.

FileSystem.getInfoAsync(fileUri, options)

Get metadata information about a file or directory.

Arguments

- **fileUri (string)** -- `file://` URI to the file or directory, or a URI returned by `CameraRoll.getPhotos()`.
- **options (object)** -- A map of options:
 - **md5 (boolean)** -- Whether to return the MD5 hash of the file. `false` by default.
 - **size (boolean)** -- Whether to include the size of the file if operating on a source from `CameraRoll.getPhotos()` (skipping this can prevent downloading the file if it's stored in iCloud, for example). The size is always returned for `file://` locations.

Returns

If no item exists at this URI, returns `{ exists: false, isDirectory: false }`. Else returns an object with the following fields:

- **exists (boolean)** -- `true`.
- **isDirectory (boolean)** -- `true` if this is a directory, `false` if it is a file
- **modificationTime (number)** -- The last modification time of the file expressed in seconds since epoch.
- **size (number)** -- The size of the file in bytes. If operating on a source from `CameraRoll.getPhotos()`, only present if the `size` option was truthy.
- **uri (string)** -- A `file://` URI pointing to the file. This is the same as the `fileUri` input parameter.
- **md5 (string)** -- Present if the `md5` option was truthy. Contains the MD5 hash of the file.

FileSystem.readAsStringAsync(fileUri, options)

Read the entire contents of a file as a string. Binary will be returned in raw format, you will need to append `data:image/png;base64,` to use it as Base64.

Arguments

- **fileUri (string)** -- `file://` URI to the file or directory.
- **options (object)** -- Optional props that define how a file must be read.
 - **encoding (EncodingType)** -- The encoding format to use when reading the file. Options: `FileSystem.EncodingTypes.UTF8` , `FileSystem.EncodingTypes.Base64` . Default is `FileSystem.EncodingTypes.UTF8` .
 - **length (number)** -- Optional number of bytes to read. This option is only used when `encoding: FileSystem.EncodingTypes.Base64` and `position` is defined.
 - **position (number)** -- Optional number of bytes to skip. This option is only used when `encoding: FileSystem.EncodingTypes.Base64` and `length` is defined.

Returns

A string containing the entire contents of the file.

FileSystem.writeAsStringAsync(fileUri, contents, options)

Write the entire contents of a file as a string.

Arguments

- **fileUri (string)** -- `file://` URI to the file or directory.
- **contents (string)** -- The string to replace the contents of the file with.
- **options (object)** -- Optional props that define how a file must be written.
 - **encoding (string)** -- The encoding format to use when writing the file. Options:
`FileSystem.EncodingTypes.UTF8` , `FileSystem.EncodingTypes.Base64` . Default is `FileSystem.EncodingTypes.UTF8`

FileSystem.deleteAsStringAsync(fileUri, options)

Delete a file or directory. If the URI points to a directory, the directory and all its contents are recursively deleted.

Arguments

- **fileUri (string)** -- `file://` URI to the file or directory.
- **options (object)** -- A map of options:
 - **idempotent (boolean)** -- If `true` , don't throw an error if there is no file or directory at this URI. `false` by default.

FileSystem.moveAsStringAsync(options)

Move a file or directory to a new location.

Arguments

- **options (object)** -- A map of options:
 - **from (string)** -- `file://` URI to the file or directory at its original location.
 - **to (string)** -- `file://` URI to the file or directory at what should be its new location.

FileSystem.copyAsStringAsync(options)

Create a copy of a file or directory. Directories are recursively copied with all of their contents.

Arguments

- **options (object)** -- A map of options:
 - **from (string)** -- `file://` URI to the file or directory to copy, or a URI returned by `CameraRoll.getPhotos()` .
 - **to (string)** -- The `file://` URI to the new copy to create.

FileSystem.makeDirectoryAsync(fileUri, options)

Create a new empty directory.

Arguments

- **fileUri (string)** -- `file://` URI to the new directory to create.
- **options (object)** -- A map of options:
 - **intermediates (boolean)** -- If `true`, create any non-existent parent directories when creating the directory at `fileUri`. If `false`, raises an error if any of the intermediate parent directories does not exist. `false` by default.

FileSystem.readDirectoryAsync(fileUri)

Enumerate the contents of a directory.

Arguments

- **fileUri (string)** -- `file://` URI to the directory.

Returns

An array of strings, each containing the name of a file or directory contained in the directory at `fileUri`.

FileSystem.downloadAsync(uri, fileUri, options)

Download the contents at a remote URI to a file in the app's file system.

Example

```
FileSystem.downloadAsync(
  'http://techslides.com/demos/sample-videos/small.mp4',
  FileSystem.documentDirectory + 'small.mp4'
)
  .then(({ uri }) => {
    console.log('Finished downloading to ', uri);
  })
  .catch(error => {
    console.error(error);
  });
});
```

Arguments

- **url (string)** -- The remote URI to download from.
- **fileUri (string)** -- The local URI of the file to download to. If there is no file at this URI, a new one is created. If there is a file at this URI, its contents are replaced.
- **options (object)** -- A map of options:
 - **md5 (boolean)** -- If `true`, include the MD5 hash of the file in the returned object. `false` by default. Provided for convenience since it is common to check the integrity of a file immediately after

downloading.

Returns

Returns an object with the following fields:

- **uri (string)** -- A `file://` URI pointing to the file. This is the same as the `fileUri` input parameter.
- **status (number)** -- The HTTP status code for the download network request.
- **headers (object)** -- An object containing all the HTTP header fields and their values for the download network request. The keys and values of the object are the header names and values respectively.
- **md5 (string)** -- Present if the `md5` option was truthy. Contains the MD5 hash of the file.

`FileSystem.createDownloadResumable(uri, fileUri, options, callback, resumeData)`

Create a `DownloadResumable` object which can start, pause, and resume a download of contents at a remote URI to a file in the app's file system. Please note: You need to call `downloadAsync()`, on a `DownloadResumable` instance to initiate the download. The `DownloadResumable` object has a callback that provides download progress updates. Downloads can be resumed across app restarts by using `AsyncStorage` to store the `DownloadResumable.savable()` object for later retrieval. The `savable` object contains the arguments required to initialize a new `DownloadResumable` object to resume the download after an app restart.

Arguments

- **url (string)** -- The remote URI to download from.
- **fileUri (string)** -- The local URI of the file to download to. If there is no file at this URI, a new one is created. If there is a file at this URI, its contents are replaced.
- **options (object)** -- A map of options:
 - **md5 (boolean)** -- If `true`, include the MD5 hash of the file in the returned object. `false` by default. Provided for convenience since it is common to check the integrity of a file immediately after downloading.
 - **headers (object)** -- An object containing any additional HTTP header fields required for the request. The keys and values of the object are the header names and values respectively.
- **callback (function)** -- This function is called on each data write to update the download progress. An object with the following fields are passed:
 - **totalBytesWritten (number)** -- The total bytes written by the download operation.
 - **totalBytesExpectedToWrite (number)** -- The total bytes expected to be written by the download operation.
- **resumeData (string)** -- The string which allows the api to resume a paused download. This is set on the `DownloadResumable` object automatically when a download is paused. When initializing a new `DownloadResumable` this should be `null`.

`FileSystem.DownloadResumable.downloadAsync()`

Download the contents at a remote URI to a file in the app's file system.

Returns

Returns an object with the following fields:

- **uri (string)** -- A `file://` URI pointing to the file. This is the same as the `fileUri` input parameter.
- **status (number)** -- The HTTP status code for the download network request.
- **headers (object)** -- An object containing all the HTTP header fields and their values for the download network request. The keys and values of the object are the header names and values respectively.
- **md5 (string)** -- Present if the `md5` option was truthy. Contains the MD5 hash of the file.

FileSystem.DownloadResumable.pauseAsync()

Pause the current download operation. `resumeData` is added to the `DownloadResumable` object after a successful pause operation. Returns an object that can be saved with `AsyncStorage` for future retrieval (the same object that is returned from calling `FileSystem.DownloadResumable.savable()`). Please see the example below.

Returns

Returns an object with the following fields:

- **url (string)** -- The remote URI to download from.
- **fileUri (string)** -- The local URI of the file to download to. If there is no file at this URI, a new one is created. If there is a file at this URI, its contents are replaced.
- **options (object)** -- A map of options:
 - **md5 (boolean)** -- If `true`, include the MD5 hash of the file in the returned object. `false` by default. Provided for convenience since it is common to check the integrity of a file immediately after downloading.
- **resumeData (string)** -- The string which allows the API to resume a paused download.

FileSystem.DownloadResumable.resumeAsync()

Resume a paused download operation.

Returns

Returns an object with the following fields:

- **uri (string)** -- A `file://` URI pointing to the file. This is the same as the `fileUri` input parameter.
- **status (number)** -- The HTTP status code for the download network request.
- **headers (object)** -- An object containing all the HTTP header fields and their values for the download network request. The keys and values of the object are the header names and values respectively.
- **md5 (string)** -- Present if the `md5` option was truthy. Contains the MD5 hash of the file.

FileSystem.DownloadResumable.savable()

Returns an object which can be saved with `AsyncStorage` for future retrieval.

Returns

Returns an object with the following fields:

- **url (string)** -- The remote URI to download from.
- **fileUri (string)** -- The local URI of the file to download to. If there is no file at this URI, a new one is created.
If there is a file at this URI, its contents are replaced.
- **options (object)** -- A map of options:
 - **md5 (boolean)** -- If `true`, include the MD5 hash of the file in the returned object. `false` by default.
Provided for convenience since it is common to check the integrity of a file immediately after downloading.
- **resumeData (string)** -- The string which allows the api to resume a paused download.

Example

```
const callback = downloadProgress => {
  const progress =
    downloadProgress.totalBytesWritten /
    downloadProgress.totalBytesExpectedToWrite;
  this.setState({
    downloadProgress: progress,
  });
};

const downloadResumable = FileSystem.createDownloadResumable(
  'http://techslides.com/demos/sample-videos/small.mp4',
  FileSystem.documentDirectory + 'small.mp4',
  {},
  callback
);

try {
  const { uri } = await downloadResumable.downloadAsync();
  console.log('Finished downloading to ', uri);
} catch (e) {
  console.error(e);
}

try {
  await downloadResumable.pauseAsync();
  console.log('Paused download operation, saving for future retrieval');
  AsyncStorage.setItem(
    'pausedDownload',
    JSON.stringify(downloadResumable.savable())
  );
} catch (e) {
  console.error(e);
}

try {
  const { uri } = await downloadResumable.resumeAsync();
  console.log('Finished downloading to ', uri);
} catch (e) {
  console.error(e);
}

//To resume a download across app restarts, assuming the the DownloadResumable.savable() object was stored:
const downloadSnapshotJson = await AsyncStorage.getItem('pausedDownload');
const downloadSnapshot = JSON.parse(downloadSnapshotJson);
```

```
const downloadResumable = new FileSystem.DownloadResumable(
  downloadSnapshot.url,
  downloadSnapshot.fileUri,
  downloadSnapshot.options,
  callback,
  downloadSnapshot.resumeData
);

try {
  const { uri } = await downloadResumable.resumeAsync();
  console.log('Finished downloading to ', uri);
} catch (e) {
  console.error(e);
}
```

#

Font

Allows loading fonts from the web and using them in React Native components. See more detailed usage information in the [Using Custom Fonts](#) guide.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Font } from 'expo';  
  
// in bare apps:  
import * as Font from 'expo-font';
```

Font.loadAsync(object)

Convenience form of `Font.loadAsync()` that loads multiple fonts at once.

Arguments

- **map (object)** -- A map of names to `require` statements as in `Font.loadAsync()` .

Example

```
Font.loadAsync({  
  Montserrat: require('./assets/fonts/Montserrat.ttf'),  
  'Montserrat-SemiBold': require('./assets/fonts/Montserrat-SemiBold.ttf'),  
});
```

Returns

Returns a promise. The promise will be resolved when the fonts have finished loading.

GestureHandler

An API for handling complex gestures. From the project's README:

This library provides an API that exposes mobile platform specific native capabilities of touch & gesture handling and recognition. It allows for defining complex gesture handling and recognition logic that runs 100% in native thread and is therefore deterministic.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
import GestureHandler from 'react-native-gesture-handler';
```

Read the [react-native-gesture-handler docs](#) for more information on the API and usage.

GLView

A `View` that acts as an OpenGL ES render target. On mounting, an OpenGL ES context is created. Its drawing buffer is presented as the contents of the `view` every frame.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { GLView } from 'expo';  
  
// in bare apps:  
import { GLView } from 'expo-gl';
```

props

Other than the regular `view` props for layout and touch handling, the following props are available:

- **onContextCreate**

A function that will be called when the OpenGL ES context is created. The function is passed a single argument `gl` that has a [WebGLRenderingContext](#) interface.

- **msaaSamples**

`GLView` can enable iOS's built-in [multisampling](#). This prop specifies the number of samples to use. By default this is 4. Setting this to 0 turns off multisampling. On Android this is ignored.

Methods

`takeSnapshotAsync(options)`

Same as [GLView.takeSnapshotAsync](#) but uses WebGL context that is associated with the view on which the method is called.

Static methods

`GLView.createContextAsync()`

Imperative API that creates headless context which is devoid of underlying view. It's useful for headless rendering or in case you want to keep just one context per application and share it between multiple components. It is slightly faster than usual context as it doesn't swap framebuffers and doesn't present them on the canvas,

however it may require you to take a snapshot in order to present its results. Also, keep in mind that you need to set up a viewport and create your own framebuffer and texture that you will be drawing to, before you take a snapshot.

Returns

A promise that resolves to WebGL context object. See [WebGL API](#) for more details.

GLView.destroyContextAsync(gl)

Destroys given context.

Arguments

- **gl (object)** -- WebGL context to destroy.

Returns

A promise that resolves to boolean value that is `true` if given context existed and has been destroyed successfully.

GLView.takeSnapshotAsync(gl, options)

Takes a snapshot of the framebuffer and saves it as a file to app's cache directory.

Arguments

- **gl (object)** -- WebGL context to take a snapshot from.
- **options (object)** -- A map of options:
 - **framebuffer (WebGLFramebuffer)** -- Specify the framebuffer that we will be reading from. Defaults to underlying framebuffer that is presented in the view or the current framebuffer if context is headless.
 - **rect ({ x: number, y: number, width: number, height: number })** -- Rect to crop the snapshot. It's passed directly to `glReadPixels`.
 - **flip (boolean)** -- Whether to flip the snapshot vertically. Defaults to `false`.
 - **format (string)** -- Either `'jpeg'` or `'png'`. Specifies what type of compression should be used and what is the result file extension. PNG compression is lossless but slower, JPEG is faster but the image has visible artifacts. Defaults to `'jpeg'`.
 - **compress (number)** -- A value in range 0 - 1 specifying compression level of the result image. 1 means no compression and 0 the highest compression. Defaults to `1.0`.

Returns

Returns `{ uri, localUri, width, height }` where `uri` is a URI to the snapshot. `localUri` is a synonym for `uri` that makes this object compatible with `texImage2D`. `width, height` specify the dimensions of the snapshot.

High-level APIs

Since the WebGL API is quite low-level, it can be helpful to use higher-level graphics APIs rendering through a `GLView` underneath. The following libraries integrate popular graphics APIs:

- [expo-three](#) for [three.js](#)
- [expo-processing](#) for [processing.js](#)

Any WebGL-supporting library that expects a [WebGLRenderingContext](#) could be used. Some times such libraries assume a web JavaScript context (such as assuming `document`). Usually this is for resource loading or event handling, with the main rendering logic still only using pure WebGL. So these libraries can usually still be used with a couple workarounds. The Expo-specific integrations above include workarounds for some popular libraries.

WebGL API

Once the component is mounted and the OpenGL ES context has been created, the `gl` object received through the `onContextCreate` prop becomes the interface to the OpenGL ES context, providing a WebGL API. It resembles a [WebGL2RenderingContext](#) in the WebGL 2 spec. However, some older Android devices may not support WebGL2 features. To check whether the device supports WebGL2 it's recommended to use `gl instanceof WebGL2RenderingContext`. An additional method `gl.endFrameEXP()` is present which notifies the context that the current frame is ready to be presented. This is similar to a 'swap buffers' API call in other OpenGL platforms.

The following WebGL2RenderingContext methods are currently unimplemented:

- `getFramebufferAttachmentParameter()`
- `getRenderbufferParameter()`
- `compressedTexImage2D()`
- `compressedTexSubImage2D()`
- `getTexParameter()`
- `getUniform()`
- `getVertexAttrib()`
- `getVertexAttribOffset()`
- `getBufferSubData()`
- `getInternalformatParameter()`
- `renderbufferStorageMultisample()`
- `compressedTexImage3D()`
- `compressedTexSubImage3D()`
- `fenceSync()`
- `isSync()`
- `deleteSync()`
- `clientWaitSync()`
- `waitSync()`
- `getSyncParameter()`
- `getActiveUniformBlockParameter()`

The `pixels` argument of `texImage2D()` must be `null`, an `ArrayBuffer` with pixel data, or an object of the form `{ localUri }` where `localUri` is the `file://` URI of an image in the device's file system. Thus an `Asset` object could be used once `.downloadAsync()` has been called on it (and completed) to fetch the resource.

For efficiency reasons the current implementations of the methods don't perform type or bounds checking on their arguments. So, passing invalid arguments could cause a native crash. We plan to update the API to perform argument checking in upcoming SDK versions. Currently the priority for error checking is low since engines generally don't rely on the OpenGL API to perform argument checking and, even otherwise, checks performed by the underlying OpenGL ES implementation are often sufficient.

Google

As of SDK 32 `Expo.Google` is just a JS wrapper for the `Expo.AppAuth` library.

Provides Google authentication integration for Expo apps using a system web browser (not `WebView`, so credentials saved on the device can be re-used!).

You'll get an access token after a successful login. Once you have the token, if you would like to make further calls to the Google API, you can use Google's [REST APIs](#) directly through HTTP (using `fetch`, for example).

In the [managed workflow](#), native Google Sign-In functionality can be used only in standalone builds, not the Expo client. If you would like to use the native authentication flow, see [GoogleSignIn](#).

Installation

The web browser-based authentication flow is provided by the `expo-app-auth` package, which is pre-installed installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow the [expo-app-auth installation instructions](#) and API reference.

Usage

```
// Example of using the Google REST API
async function getUserInfo(accessToken) {
  let userInfoResponse = await fetch('https://www.googleapis.com/userinfo/v2/me', {
    headers: { Authorization: `Bearer ${accessToken}` },
  });

  return await userInfoResponse.json();
}
```

API

```
import { Google } from 'expo';
```

Methods

logInAsync

```
logInAsync(config: LogInConfig): Promise<LogInResult>
```

This method uses `AppAuth` to authenticate; for even more native functionality see [expo-google-sign-in](#).

Prompts the user to log into Google and grants your app permission to access some of their Google data, as specified by the scopes. The difference between this method and native authentication are very sparse. Google has done a very good job at making the web auth flow work consistently. The biggest difference is that you

cannot use `expo-google-sign-in` in the Expo Client (standalone apps only), which makes `Expo.Google.logInAsync` your best solution for testing in development.

Parameters

Name	Type	Description
config	<code>LogInConfig</code>	Used to log into your Google application.

LogInConfig

Name	Type	Description
clientId	<code>string</code>	Web API key that denotes the Google application to log in to
scopes	<code>string[]</code>	An array specifying the scopes to ask for from Google for this login (more information here). Default scopes are <code>'[profile', 'email']'</code>
androidClientId	<code>string</code>	DEPRECATED use <code>clientId</code> instead
iosClientId	<code>string</code>	DEPRECATED use <code>clientId</code> instead
androidStandaloneAppClientId	<code>string</code>	DEPRECATED use <code>clientId</code> instead
iosStandaloneAppClientId	<code>string</code>	DEPRECATED use <code>clientId</code> instead
webClientId	<code>string</code>	DEPRECATED use <code>clientId</code> instead

Returns

Name	Type	Description
logInResult	<code>Promise<LogInResult></code>	Resolves into the results of your login attempt.

LogInResult

Name	Type	Description	
type	<code>'cancel'</code>	<code>'success'</code>	Denotes the summary of the user event.
accessToken	<code>'string</code>	<code>undefined`</code>	Used for accessing data from Google, invalidate to "log out"
idToken	<code>'string</code>	<code>null`</code>	ID token
refreshToken	<code>'string</code>	<code>null`</code>	Refresh the other tokens.
user	<code>GoogleUser</code>	An object with data regarding the authenticated user.	

GoogleUser

Name	Type	Description	
id	<code>'string</code>	<code>undefined`</code>	optional ID for the user
name	<code>'string</code>	<code>undefined`</code>	optional name for the user
givenName	<code>'string</code>	<code>undefined`</code>	optional first name for the user

familyName	`string`	undefined`	optional last name for the user
photoUrl	`string`	undefined`	optional photo for the user
email	`string`	undefined`	optional email address for the user

Example

```
import { Google } from 'expo';

const clientId = '<YOUR_WEB_CLIENT_ID>';
const { type, accessToken, user } = await Google.logInAsync({ clientId });

if (type === 'success') {
  /* `accessToken` is now valid and can be used to get data from the Google API with HTTP requests */
  console.log(user);
}
```

logOutAsync

```
logOutAsync({ accessToken, clientId }): Promise<any>
```

Invalidates the provided `accessToken`, given the `clientId` used to sign-in is provided. This method is an alias for the following functionality:

```
import { AppAuth } from 'expo-app-auth';

async function logOutAsync({ accessToken, clientId }): Promise<any> {
  const config = {
    issuer: 'https://accounts.google.com',
    clientId,
  };

  return await AppAuth.revokeAsync(config, {
    token: accessToken,
    isClientIdProvided: !!clientId,
  });
}
```

Parameters

Name	Type	Description
options	{ accessToken: string, clientId: string }	Used to log out of the Google application.

options

Name	Type	Description
accessToken	string	Provided when the user authenticates with your Google application.
clientId	string	Used to identify the corresponding application.

Example

```
import { Google } from 'expo';

const clientId = '<YOUR_WEB_CLIENT_ID>';
```

```

const { type, accessToken } = await Google.logInAsync({ clientId });

if (type === 'success') {
  /* Log-Out */
  await Google.logOutAsync({ clientId, accessToken });
  /* `accessToken` is now invalid and cannot be used to get data from the Google API with HTTP requests */
}

```

Using it inside of the Expo app

In the Expo Client app, you can only use browser-based login (this works very well actually because it re-uses credentials saved in your system browser). If you build a standalone app, you can use the native login with the package `expo-google-sign-in`.

To use Google Sign In, you will need to create a project in Firebase (or on the Google Developer Console). In Firebase create a project, then enable Google Sign-In in the Authentication tab on the left side of the page.

Server side APIs

If you need to access Google APIs using the user's authorization you need to pass an additional web client id. This will add accessToken, idToken, refreshToken and serverAuthCode to the response object that you can use on your server with the client id secret.

1. Open your browser to [Google Developer Credentials](#)
2. Click **Create credentials** and then **OAuth client ID**, then choose **web** and press **Create**.

When your app is running as an Expo experience, the process is a little different. Due to Google's restrictions, the only way to make this work is via web authentication flow. Once you have your code, send it to your backend and exchange it, but make sure to set the redirect_uri parameter to the same value you used on your client side call to Google. (Something similar to <https://auth.expo.io/@username/your-app-slug>). With Expo, you can easily authenticate your user with the `AuthSession` module:

```

let result = await Expo.AuthSession.startAsync({
  authUrl:
    `https://accounts.google.com/o/oauth2/v2/auth?` +
    `&client_id=${googleWebAppId}` +
    `&redirect_uri=${encodeURIComponent(redirectUrl)}` +
    `&response_type=code` +
    `&access_type=offline` +
    `&scope=profile`,
});

```

GoogleSignIn

This library provides native Google authentication for **standalone** Expo apps or bare React Native apps. It cannot be used in the Expo client as the native `GoogleSignIn` library expects your `REVERSE_CLIENT_ID` in the `info.plist` at build-time. To use Google authentication in the Expo Client, check out [Google](#) or [AppAuth](#).

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Setup

For questions on setup, feel free to comment on this post: [React Native Google Sign-Up](#)

1. Go to your `app.json` and make sure you define your `ios.bundleIdentifier` and the `android.package` you want to use.
2. Open up the [Firebase Console](#) and setup a new project, or use an existing one.
3. Create a native iOS, and Android app using the Bundle ID and Android package you defined earlier.
4. Download the `GoogleService-info.plist` (iOS) & the `google-services.json` (Android). Move them to your Expo project.
5. In the `app.json`, set your `expo.ios.config.googleSignIn.reservedClientId` to the value of `REVERSE_CLIENT_ID` in the `GoogleService-info.plist`.
6. Also in `app.json`, set `expo.android.googleServicesFile` to the relative path of your `google-services.json`. Make sure the file is located somewhere in your Expo project.

```
// app.json
{
  "expo": {
    "ios": {
      // The bundle ID you used with your Firebase app
      "bundleIdentifier": "example.expo.googlesignin",
      "config": {
        "googleSignIn": {
          // Your REVERSE_CLIENT_ID from the GoogleService-info.plist
          "reservedClientId": "<YOUR_IOS_CLIENT_ID>"
        }
      }
    },
    "android": {
      // The package you used with your Firebase app
      "package": "example.expo.googlesignin",
      // Relative path to the file generated by Firebase
      "googleServicesFile": "./google-services.json"
    }
  }
}
```

At this point you can build your project and upload it to the App Store or the Google Play Store.

When your app is built you can verify that the iOS URL Scheme is properly set up by reading it using the `expo-app-auth` module in a standalone app.

```
import { AppAuth } from 'expo-app-auth';

// This value should contain your REVERSE_CLIENT_ID
const { URLschemes } = AppAuth;
```

Initialize the API

Before using the API we first need to call `GoogleSignIn.initAsync({ ... })` which configures how sign in functionality will work.

```
try {
  await GoogleSignIn.initAsync({ clientId: '<YOUR_IOS_CLIENT_ID>' });
} catch ({ message }) {
  alert('GoogleSignIn.initAsync(): ' + message);
}
```

Signing in/out

```
signInAsync = async () => {
  try {
    await GoogleSignIn.askForPlayServicesAsync();
    const { type, user } = await GoogleSignIn.signInAsync();
    if (type === 'success') {
      // ...
    }
  } catch ({ message }) {
    alert('login: Error:' + message);
  }
};

signOutAsync = async () => {
  try {
    await GoogleSignIn.signOutAsync();
    this.setState({ user: null });
  } catch ({ message }) {
    alert('signOutAsync: ' + message);
  }
};
```

Methods

getPlayServiceAvailability(shouldAsk: boolean = false): Promise<boolean>

Android Only, this method always returns true on iOS

Use this method to determine if a user's device can utilize Google Sign-In functionality. By default this method will assume the option is `false` and silently check for Play Services, whereas passing `true` will present a modal if the Play Services aren't available, prompting the user to update Play Services.

askForPlayServicesAsync(): Promise<boolean>

Android Only, this method always returns true on iOS

A convenience wrapper for `getPlayServiceAvailability(true)`, this method will present a modal for the user to update Play Services if they aren't already up-to-date.

Returns true after the user successfully updates.

initAsync(options: ?GoogleSignInOptions): Promise<void>

Configures how the `GoogleSignIn` module will attempt to sign in. You can call this method multiple times.

See all the available options under the `GoogleSignInOptions` type.

isSignedInAsync(): Promise<boolean>

Asynchronously returns a boolean representing the user's authentication status.

signInSilentlyAsync(): Promise<?GoogleUser>

This method will attempt to reauthenticate the user without initializing the authentication flow. If the method is successful, the currently authenticated `GoogleUser` will be returned, otherwise the method will return `null`.

signInAsync(): Promise<?GoogleSignInAuthResult>

Starts the native authentication flow with the information provided in `initAsync()`. If a user cancels, the method will return `{ type: 'cancel', user: null }`. However if a user successfully finishes the authentication flow, the returned value will be: `{ type: 'success', user: GoogleUser }`.

There are some errors that can be thrown while authenticating, check `GoogleSignIn.ERRORS` for available error codes.

signOutAsync(): Promise<void>

Signs out the currently authenticated user. Unlike `disconnectAsync()`, this method will not revoke the access token. This means you can specify the `accountName` and reauthenticate without extra user approval.

isConnectedAsync(): Promise<boolean>

Returns true if a user is authenticated and the access token has not been invalidated.

disconnectAsync(): Promise<void>

Signs out the current user and revokes the access tokens associated with the account. This will prevent reauthentication, whereas `signOutAsync()` will not.

getCurrentUserAsync(): Promise<GoogleUser | null>

If a user is authenticated, this method will return all the basic profile information in the form of a `GoogleUser`.

getCurrentUser(): GoogleUser | null

Get the most recent instance of the authenticated `GoogleUser`.

getPhotoAsync(size: number = 128): Promise<?string>

Returns an image URI for the currently authenticated user. This method will return `null` if no user is signed in, or if the current user doesn't have a profile image on Google. The default size is `128px`, if the requested image size is larger than the original image size, the full sized image will be returned.

Types

```
/* Android Only */
type GoogleSignInType = 'default' | 'games';

type GoogleSignInOptions = {
    /*
     * [iOS][Android][optional]: `accountName: ?string`
     * [default]: `[GoogleSignIn.SCOPES.PROFILE, GoogleSignIn.SCOPES.EMAIL]`
     * Pass the scopes you wish to have access to.
     */
    scopes: ?Array<string>,

    /*
     * [iOS][Android][optional]: `webClientId: ?string`
     * [default]: `undefined`
     * The client ID of the home web server. This will be returned as the |audience| property of the
     * OpenID Connect ID token. For more info on the ID token:
     * https://developers.google.com/identity/sign-in/ios/backend-auth
     */
    webClientId: ?string,

    /*
     * [iOS][Android][optional]: `hostedDomain: ?string`
     * [default]: `undefined`
     * The hosted G Suite domain of the user. Provided only if the user belongs to a hosted domain.
     */
    hostedDomain: ?string,

    /*
     * [iOS][Android][optional]: `accountName: ?string`
     * [default]: `undefined`
     * If you know the user's email address ahead of time, you can add it here and it will be the default option
     * if the user has approved access for this app, the Auth will return instantly.
     */
    accountName: ?string,

    /*
     * [Android][optional]: `signInType?: GoogleSignIn.TYPES.DEFAULT | GoogleSignIn.TYPES.GAMES`
     * [default]: `undefined`
     * The service you wish to sign in to
     * GoogleSignIn.TYPES.DEFAULT | GoogleSignIn.TYPES.GAMES
     */
    signInType: ?GoogleSignInType,

    /*
     * [Android][optional]: `isOfflineEnabled: ?boolean`
     * [default]: `undefined`
     * If true, the server will return refresh tokens that can be used to access data when the user has unauthenticated
     *
     * 1. Safely secure the refresh token as you can only get one during the initial auth flow.
     * 2. There are only so many refresh tokens that are issued, limit per user/app, you can also get one for a single
     * user across all clients in an app. If you requests too many tokens, older tokens will begin to be invalidated.
     */
}
```

```

isOfflineEnabled: ?boolean,

/*
 * [Android][optional]: `isPromptEnabled: ?boolean`
 * [default]: false
 * Forces the consent prompt to be shown everytime a user authenticates. Enable this only when necessary.
 */
isPromptEnabled: ?boolean,

/*
 * [iOS][optional]: `clientId: ?string`
 * [default]: Read from GoogleService-info.plist `CLIENT_ID` on iOS, and google-services.json `oauth_client.client_id` on Android.
 * The client ID of the app from the Google APIs (or Firebase) console, this must be set for sign in to work.
 * This value must be defined in the google-services.json on Android, you can define your custom google-services.json on
 */
clientId: ?string,

/*
 * [iOS][optional]: `language: ?string`
 * [default]: `undefined`
 * The language for sign in, in the form of ISO 639-1 language code optionally followed by a dash
 * and ISO 3166-1 alpha-2 region code, such as `@"it"` or `@"pt-PT"`. Only set if different from
 * system default.
 */
language: ?string,

/*
 * [iOS][optional]: `openIdRealm?: ?string`
 * [default]: `undefined`
 * The OpenID2 realm of the home web server. This allows Google to include the user's OpenID
 * Identifier in the OpenID Connect ID token..
 */
openIdRealm: ?string,
};


```

```
type GoogleSignInAuthResultType = 'success' | 'cancel';
```

```

type GoogleSignInAuthResult = {
  type: GoogleSignInAuthResultType,
  user: ?User,
};

```

Classes

GoogleAuthData

The base class for `GoogleSignIn` authentication data. This method enables you to compare and serialize objects.

Methods:

- `equals(other: ?any): boolean`
- `toJSON(): object`

GoogleIdentity

Extends `GoogleAuthData`, core management of user data.

Variables:

- `uid: string;`
- `email: string;`
- `displayName: ?string;`
- `photoURL: ?string;`
- `firstName: ?string;`
- `lastName: ?string;`

GoogleUser

Extends `GoogleIdentity`, manages all data regarding an authenticated user.

Variables:

- `auth: ?Authentication;`
- `scopes: Array<string>;`
- `hostedDomain: ?string;`
- `serverAuthCode: ?string;`

Methods:

- `clearCache(): void`
- `getHeaders(): Promise<{ [string]: string }>`
- `refreshAuth(): Promise<?GoogleAuthentication>`

GoogleAuthentication

Extends `GoogleAuthData`, manages the user tokens.

Variables:

- `clientId: ?string;`
- `accessToken: ?string;`
- `accessTokenExpirationDate: ?number;`
- `refreshToken: ?string;`
- `idToken: ?string;`
- `idTokenExpirationDate: ?number; | UNIX time in milliseconds`

Constants

GoogleSignIn.ERRORS

All of the available authentication error codes.

- `GoogleSignIn.ERRORS.SIGN_IN_CANCELLED` The user has cancelled the auth flow
- `GoogleSignIn.ERRORS.SIGN_IN_REQUIRED` Attempting to access user data before any user has been authenticated
- `GoogleSignIn.ERRORS.TASK_IN_PROGRESS` An existing auth task is already running.
- `GoogleSignIn.ERRORS.SIGN_IN_EXCEPTION` A general error has occurred
- `GoogleSignIn.ERRORS.SIGN_IN_FAILED` A Play Services error has occurred (Android only)
- `GoogleSignIn.ERRORS.INVALID_ACCOUNT` An invalid account has been provided with `accountName` (Android only)
- `GoogleSignIn.ERRORS.SIGN_IN_NETWORK_ERROR` An issue with the internet connection has caused the auth task to

fail (Android only)

GoogleSignIn.SCOPES

- GoogleSignIn.SCOPES.PROFILE
- GoogleSignIn.SCOPES.EMAIL
- GoogleSignIn.SCOPES.OPEN_ID
- GoogleSignIn.SCOPES.PLUS_ME
- GoogleSignIn.SCOPES.GAMES
- GoogleSignIn.SCOPES.GAMES_LITE
- GoogleSignIn.SCOPES.CLOUD_SAVE
- GoogleSignIn.SCOPES.APP_STATE
- GoogleSignIn.SCOPES.DRIVE_FILE
- GoogleSignIn.SCOPES.DRIVE_APPFOLDER
- GoogleSignIn.SCOPES.DRIVE_FULL
- GoogleSignIn.SCOPES.DRIVE_APPS
- GoogleSignIn.SCOPES.FITNESS_ACTIVITY_READ
- GoogleSignIn.SCOPES.FITNESS_ACTIVITY_READ_WRITE
- GoogleSignIn.SCOPES.FITNESS_LOCATION_READ
- GoogleSignIn.SCOPES.FITNESS_LOCATION_READ_WRITE
- GoogleSignIn.SCOPES.FITNESS_BODY_READ
- GoogleSignIn.SCOPES.FITNESS_BODY_READ_WRITE
- GoogleSignIn.SCOPES.FITNESS_NUTRITION_READ
- GoogleSignIn.SCOPES.FITNESS_NUTRITION_READ_WRITE
- GoogleSignIn.SCOPES.FITNESS_BLOOD_PRESSURE_READ
- GoogleSignIn.SCOPES.FITNESS_BLOOD_PRESSURE_READ_WRITE
- GoogleSignIn.SCOPES.FITNESS_BLOOD_GLUCOSE_READ
- GoogleSignIn.SCOPES.FITNESS_BLOOD_GLUCOSE_READ_WRITE
- GoogleSignIn.SCOPES.FITNESS_OXYGEN_SATURATION_READ
- GoogleSignIn.SCOPES.FITNESS_OXYGEN_SATURATION_READ_WRITE
- GoogleSignIn.SCOPES.FITNESS_BODY_TEMPERATURE_READ
- GoogleSignIn.SCOPES.FITNESS_BODY_TEMPERATURE_READ_WRITE
- GoogleSignIn.SCOPES.FITNESS_REPRODUCTIVE_HEALTH_READ
- GoogleSignIn.SCOPES.FITNESS_REPRODUCTIVE_HEALTH_READ_WRITE

GoogleSignIn.TYPES

All of the available sign in types.

- GoogleSignIn.TYPES.DEFAULT The standard login method.
- GoogleSignIn.TYPES.GAMES Sign in to Google Play Games (Android only)

Usage

```
import React from 'react';
import { Text } from 'react-native';
import { GoogleSignIn } from 'expo-google-sign-in';

export default class AuthScreen extends React.Component {
  state = { user: null };
```

```

componentDidMount() {
  this.initAsync();
}

initAsync = async () => {
  await GoogleSignIn.initAsync({
    clientId: '<YOUR_IOS_CLIENT_ID>',
  });
  this._syncUserWithStateAsync();
};

_syncUserWithStateAsync = async () => {
  const user = await GoogleSignIn.signInSilentlyAsync();
  this.setState({ user });
};

signOutAsync = async () => {
  await GoogleSignIn.signOutAsync();
  this.setState({ user: null });
};

signInAsync = async () => {
  try {
    await GoogleSignIn.askForPlayServicesAsync();
    const { type, user } = await GoogleSignIn.signInAsync();
    if (type === 'success') {
      this._syncUserWithStateAsync();
    }
  } catch ({ message }) {
    alert('login: Error:' + message);
  }
};

onPress = () => {
  if (this.state.user) {
    this.signOutAsync();
  } else {
    this.signInAsync();
  }
};

render() {
  return <Text onPress={this.onPress}>Toggle Auth</Text>;
}
}

```

Gyroscope

Access the device gyroscope sensor to respond to changes in rotation in 3d space.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Gyroscope } from 'expo';  
  
// in bare apps:  
import { Gyroscope } from 'expo-sensors';
```

Gyroscope.isAvailableAsync()

Returns whether the gyroscope is enabled on the device.

Returns

- A promise that resolves to a `boolean` denoting the availability of the sensor.

Gyroscope.addListener(listener)

Subscribe for updates to the gyroscope.

Arguments

- **listener (function)** -- A callback that is invoked when an gyroscope update is available. When invoked, the listener is provided a single argument that is an object containing keys x, y, z.

Returns

- A subscription that you can call `remove()` on when you would like to unsubscribe the listener.

Gyroscope.removeAllListeners()

Remove all listeners.

Gyroscope.setUpdateInterval(intervalMs)

Subscribe for updates to the gyroscope.

Arguments

- **intervalMs (number)** -- Desired interval in milliseconds between gyroscope updates.

Example: basic subscription

```

import React from 'react';
import { Gyroscope } from 'expo';
import { StyleSheet, Text, TouchableOpacity, View } from 'react-native';

export default class GyroscopeSensor extends React.Component {
  state = {
    gyroscopeData: {},
  };

  componentDidMount() {
    this._toggle();
  }

  componentWillUnmount() {
    this._unsubscribe();
  }

  _toggle = () => {
    if (this._subscription) {
      this._unsubscribe();
    } else {
      this._subscribe();
    }
  };

  _slow = () => {
    Gyroscope.setUpdateInterval(1000);
  };

  _fast = () => {
    Gyroscope.setUpdateInterval(16);
  };

  _subscribe = () => {
    this._subscription = Gyroscope.addListener(result => {
      this.setState({ gyroscopeData: result });
    });
  };

  _unsubscribe = () => {
    this._subscription && this._subscription.remove();
    this._subscription = null;
  };

  render() {
    let { x, y, z } = this.state.gyroscopeData;

    return (
      <View style={styles.sensor}>
        <Text>Gyroscope:</Text>
        <Text>
          x: {round(x)} y: {round(y)} z: {round(z)}
        </Text>

        <View style={styles.buttonContainer}>
          <TouchableOpacity onPress={this._toggle} style={styles.button}>
            <Text>Toggle</Text>
          </TouchableOpacity>
          <TouchableOpacity onPress={this._slow} style={[styles.button, styles.middleButton]}>

```

```

        <Text>Slow</Text>
    </TouchableOpacity>
    <TouchableOpacity onPress={this._fast} style={styles.button}>
        <Text>Fast</Text>
    </TouchableOpacity>
</View>
</View>
);
}
}

function round(n) {
if (!n) {
    return 0;
}

return Math.floor(n * 100) / 100;
}

const styles = StyleSheet.create({
container: {
    flex: 1,
},
buttonContainer: {
    flexDirection: 'row',
    alignItems: 'stretch',
    marginTop: 15,
},
button: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#eee',
    padding: 10,
},
middleButton: {
    borderLeftWidth: 1,
    borderRightWidth: 1,
    borderColor: '#ccc',
},
sensor: {
    marginTop: 15,
    paddingHorizontal: 10,
},
});

```

Haptic

This API will be renamed to Haptics in SDK 33. If you are using the bare workflow, use that API now instead: [Haptics](#).

Provides haptic feedback for iOS 10+ devices using the Taptic Engine. If this is used in Android, the device will use `ReactNative.Vibrate` instead.

The Taptic engine will do nothing given the following circumstances:

- Low Power Mode is enabled
 - [Feature Request](#)
- User disabled the Taptic Engine in settings
 - [Feature Request](#)
- Haptic engine generation is too low (less than 2nd gen) - Private API
 - Using private API will get your app rejected: `[[UIDevice currentDevice] valueForKey:@"_feedbackSupportLevel"]` so this is not added in Expo
- iOS version is less than 10 (iPhone 7 is the first phone to support this)
 - This could be found through: `Constants.platform.ios.systemVersion` OR `Constants.platform.ios.platform`

Installation

This API is pre-installed in [managed](#) apps. Use [Haptics](#) in a bare React Native app instead of Haptic.

API

```
// in managed apps:  
import { Haptic } from 'expo';
```

Haptic.selection()

Used to let a user know when a selection change has been registered

Example

```
Haptic.selection()
```

Haptic.notification(type: NotificationFeedbackType)

Property	Type	Description
type	NotificationFeedbackType	The kind of notification response used in the feedback

Example

```
Haptic.notification(Haptic.NotificationFeedbackType.Success)
```

Haptic.impact(style: ImpactFeedbackStyle)

Property	Type	Description
style	ImpactFeedbackStyle	The level of impact used in the feedback

Example

```
Haptic.impact(Haptic.ImpactFeedbackStyle.Light)
```

Constants

NotificationTypes

The type of notification generated by a UINotificationFeedbackGenerator object.

```
const NotificationTypes = {
  'Success': 'success',
  'Warning': 'warning',
  'Error': 'error',
}
```

Success

A notification feedback type, indicating that a task has completed successfully.

Warning

A notification feedback type, indicating that a task has produced a warning.

Error

A notification feedback type, indicating that a task has failed.

ImpactStyles

The mass of the objects in the collision simulated by a UIImpactFeedbackGenerator object.

```
const ImpactStyles = {
  'Light': 'light',
  'Medium': 'medium',
  'Heavy': 'heavy',
}
```

Light

A collision between small, light user interface elements.

Medium

A collision between moderately sized user interface elements.

Heavy

A collision between large, heavy user interface elements.

Types

NotificationFeedbackType

The type of notification generated by a UINotificationFeedbackGenerator object.

```
$Enum<{
  success: string,
  warning: string,
  error: string,
}>>
```

ImpactFeedbackStyle

The mass of the objects in the collision simulated by a UIImpactFeedbackGenerator object.

```
$Enum<{
  light: string,
  medium: string,
  heavy: string,
}>>
```

Haptics

The following API docs apply to the Haptics Unimodule available to the bare workflow. Use [Haptic](#) for the managed workflow.

Provides haptic feedback for

- iOS 10+ devices using the Taptic Engine
- Android devices using Vibrator system service.

On iOS, *the Taptic engine will do nothing if any of the following conditions are true on a user's device:*

- Low Power Mode is enabled ([Feature Request](#))
- User disabled the Taptic Engine in settings ([Feature Request](#))
- Haptic engine generation is to low (less than 2nd gen) - Private API
 - Using private API will get your app rejected: `[[UIDevice currentDevice] valueForKey:@"_feedbackSupportLevel"]` so this is not added in Expo
- iOS version is less than 10 (iPhone 7 is the first phone to support this)
 - This could be found through: `Constants.platform.ios.systemVersion` OR `Constants.platform.ios.platform`

Installation

This API is currently called [Haptic](#) in the managed workflow, refer to that page instead. To use Haptics in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in bare apps:  
import * as Haptics from 'expo-haptics';
```

Haptics.selectionAsync()

Used to let a user know when a selection change has been registered

Returns

A `Promise` resolving once native size haptics functionality is triggered.

Haptics.notificationAsync(type)

The kind of notification response used in the feedback

Arguments

- **type:** `NotificationFeedbackType` -- A notification feedback type that on `ios` is directly mapped to `UINotificationFeedbackType`, while on `Android` these are simulated using `Vibrator`. You can use one of `Haptics.NotificationFeedbackType.{Success, Warning, Error}` .

Returns

A `Promise` resolving once native size haptics functionality is triggered.

`Haptics.impactAsync(style)`

Arguments

- **style:** `ImpactFeedbackStyle` -- A collision indicator that on `ios` is directly mapped to `UIImpactFeedbackStyle`, while on `Android` these are simulated using `Vibrator`. You can use one of `Haptics.ImpactFeedbackStyle.{Light, Medium, Heavy}`.

Returns

A `Promise` resolving once native size haptics functionality is triggered.

Admob

Expo includes support for the [Google AdMob SDK](#) for mobile advertising. This module is largely based of the [react-native-admob](#) module, as the documentation and questions surrounding that module may prove helpful. A simple example implementing AdMob SDK can be found [here](#).

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

```
import {
  AdMobBanner,
  AdMobInterstitial,
  PublisherBanner,
  AdMobRewarded
} from 'expo';

// Display a banner
<AdMobBanner
  bannerSize="fullBanner"
  adUnitID="ca-app-pub-3940256099942544/6300978111" // Test ID, Replace with your-admob-unit-id
  testDeviceID="EMULATOR"
  onDidFailToReceiveAdWithError={this.bannerError} />

// Display a DFP Publisher banner
<PublisherBanner
  bannerSize="fullBanner"
  adUnitID="ca-app-pub-3940256099942544/6300978111" // Test ID, Replace with your-admob-unit-id
  testDeviceID="EMULATOR"
  onDidFailToReceiveAdWithError={this.bannerError}
  onAdMobDispatchAppEvent={this.adMobEvent} />

// Display an interstitial
AdMobInterstitial.setAdUnitID('ca-app-pub-3940256099942544/1033173712'); // Test ID, Replace with your-admob-unit-id
AdMobInterstitial.setTestDeviceID('EMULATOR');
await AdMobInterstitial.requestAdAsync();
await AdMobInterstitial.showAdAsync();

// Display a rewarded ad
AdMobRewarded.setAdUnitID('ca-app-pub-3940256099942544/5224354917'); // Test ID, Replace with your-admob-unit-id
AdMobRewarded.setTestDeviceID('EMULATOR');
await AdMobRewarded.requestAdAsync();
await AdMobRewarded.showAdAsync();
```

API

```
// in managed apps:
import { AdMobBanner, AdMobInterstitial, AdMobRewarded, PublisherBanner } from 'expo';

// in bare apps:
import { AdMobBanner, AdMobInterstitial, AdMobRewarded, PublisherBanner } from 'expo-ads-admob';
```

AdMobBanner

bannerSize property

Corresponding to [iOS framework banner size constants](#)

Prop value	Description	Size
banner	Standard Banner for Phones and Tablets	320x50
largeBanner	Large Banner for Phones and Tablets	320x100
mediumRectangle	IAB Medium Rectangle for Phones and Tablets	300x250
fullBanner	IAB Full-Size Banner for Tablet	468x60
leaderboard	IAB Leaderboard for Tablets	728x90
smartBannerPortrait	Smart Banner for Phones and Tablets (default)	Screen width x 32
smartBannerLandscape	Smart Banner for Phones and Tablets	Screen width x 32

Note: There is no `smartBannerPortrait` and `smartBannerLandscape` on Android. Both prop values will map to `smartBanner`

Events as function props

Corresponding to [Ad lifecycle event callbacks](#)

Prop
<code>onAdViewDidReceiveAd()</code>
<code>onDidFailToReceiveAdWithError(errorDescription: string)</code>
<code>onAdViewWillPresentScreen()</code>
<code>onAdViewWillDismissScreen()</code>
<code>onAdViewDidDismissScreen()</code>
<code>onAdViewWillLeaveApplication()</code>

AdMobInterstitials

Methods

Name	Description
<code>setAdUnitID(adUnitID)</code>	sets the AdUnit ID for all future ad requests.
<code>setTestDeviceID(deviceID)</code>	sets the test device ID
<code>requestAdAsync()</code>	requests an interstitial and resolves when <code>interstitialDidLoad</code> or <code>interstitialDidFailToLoad</code> event fires
<code>showAdAsync()</code>	shows an interstitial if it is ready and resolves when <code>interstitialDidOpen</code> event fires
<code>getIsReadyAsync()</code>	resolves with boolean whether interstitial is ready to be shown

For simulators/emulators you can use '`EMULATOR`' for the test device ID.

Events

Unfortunately, events are not consistent across iOS and Android. To have one unified API, new event names are introduced for pairs that are roughly equivalent.

iOS	<i>this library</i>	Android
<code>interstitialDidReceiveAd</code>	<code>interstitialDidLoad</code>	<code>onAdLoaded</code>
<code>interstitial:didFailToReceiveAdWithError</code>	<code>interstitialDidFailToLoad</code>	<code>onAdFailedToLoad</code>
<code>interstitialWillPresentScreen</code>	<code>interstitialDidOpen</code>	<code>onAdOpened</code>
<code>interstitialDidFailToPresentScreen</code>		
<code>interstitialWillDismissScreen</code>		
<code>interstitialDidDismissScreen</code>	<code>interstitialDidClose</code>	<code>onAdClosed</code>
<code>interstitialWillLeaveApplication</code>	<code>interstitialWillLeaveApplication</code>	<code>onAdLeftApplication</code>

Note that `interstitialWillLeaveApplication` and `onAdLeftApplication` are not exactly the same but share one event in this library.

AdMobRewarded

Opens a rewarded AdMob ad.

Methods

Name	Description
<code>setAdUnitID(adUnitID: string)</code>	sets the AdUnit ID for all future ad requests.
<code>setTestDeviceID(testDeviceID: string)</code>	sets the test device ID
<code>requestAdAsync()</code>	(async) requests a rewarded ad
<code>showAdAsync()</code>	(async) shows a rewarded if it is ready (async)

Events

iOS	<i>this library</i>	Android
<code>rewardBasedVideoAd:didRewardUserWithReward</code>	<code>rewardedVideoDidRewardUser</code>	<code>onRewarded</code>
<code>rewardBasedVideoAdDidReceiveAd</code>	<code>rewardedVideoDidLoad</code>	<code>onRewardedVideoAdLoaded</code>
<code>rewardBasedVideoAd:didFailToLoadWithError</code>	<code>rewardedVideoDidFailToLoad</code>	<code>onRewardedVideoAdFailedToLoad</code>
<code>rewardBasedVideoAdDidOpen</code>	<code>rewardedVideoDidOpen</code>	<code>onRewardedVideoAdOpened</code>
	<code>rewardedVideoDidComplete</code>	<code>onRewardedVideoCompleted</code>
<code>rewardBasedVideoAdDidClose</code>	<code>rewardedVideoDidClose</code>	<code>onRewardedVideoAdClosed</code>
<code>rewardBasedVideoAdWillLeaveApplication</code>	<code>rewardedVideoWillLeaveApplication</code>	<code>onRewardedVideoAdLeftApplication</code>

#

ImagePicker

Provides access to the system's UI for selecting images and videos from the phone's library or taking a photo with the camera.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

API

```
// in managed apps:  
import { ImagePicker } from 'expo';  
  
// in bare apps:  
import * as ImagePicker from 'expo-image-picker';
```

ImagePicker.launchImageLibraryAsync(options)

Display the system UI for choosing an image or a video from the phone's library. Requires `Permissions.CAMERA_ROLL` on iOS only.

Arguments

- **options (object)** --

A map of options for both:

- **mediaTypes (String)** -- Choose what type of media to pick. Usage: `ImagePicker.MediaTypeOptions.<Type>`, where `<Type>` is one of: `Images` , `Videos` , `All` .
- **allowsEditing (boolean)** -- Whether to show a UI to edit the image/video after it is picked. Images: On Android the user can crop and rotate the image and on iOS simply crop it. Videos: On iOS user can trim the video. Defaults to `false` .

A map of options for images:

- **aspect (array)** -- An array with two entries `[x, y]` specifying the aspect ratio to maintain if the user is allowed to edit the image (by passing `allowsEditing: true`). This is only applicable on Android, since on iOS the crop rectangle is always a square.
- **quality (number)** -- Specify the quality of compression, from 0 to 1. 0 means compress for small size, 1 means compress for maximum quality.
- **base64 (boolean)** -- Whether to also include the image data in Base64 format.
- **exif (boolean)** -- Whether to also include the EXIF data for the image.

Returns

If the user cancelled the picking, returns `{ cancelled: true }`.

Otherwise, returns `{ cancelled: false, uri, width, height, type }` where `uri` is a URI to the local media file (useable as the source for an `Image / Video` element), `width, height` specify the dimensions of the media and `type` is one of `image` or `video` telling what kind of media has been chosen. Images can contain also `base64` and `exif` keys. `base64` is included if the `base64` option was truthy, and is a string containing the JPEG data of the image in Base64--prepend that with `'data:image/jpeg;base64,'` to get a data URI, which you can use as the source for an `Image` element for example. `exif` is included if the `exif` option was truthy, and is an object containing EXIF data for the image--the names of its properties are EXIF tags and their values are the values for those tags. If a video has been picked the return object contains an additional key `duration` specifying the video's duration in miliseconds.

`ImagePicker.launchCameraAsync(options)`

Display the system UI for taking a photo with the camera. Requires `Permissions.CAMERA` along with `Permissions.CAMERA_ROLL`.

Arguments

- **options (object) --**

A map of options:

- **allowsEditing (boolean)** -- Whether to show a UI to edit the image after it is picked. On Android the user can crop and rotate the image and on iOS simply crop it. Defaults to `false`.
- **aspect (array)** -- An array with two entries `[x, y]` specifying the aspect ratio to maintain if the user is allowed to edit the image (by passing `allowsEditing: true`). This is only applicable on Android, since on iOS the crop rectangle is always a square.
- **quality (number)** -- Specify the quality of compression, from 0 to 1.0 means compress for small size, 1 means compress for maximum quality.
- **base64 (boolean)** -- Whether to also include the image data in Base64 format.
- **exif (boolean)** -- Whether to also include the EXIF data for the image. On iOS the EXIF data does not include GPS tags in the camera case.

Returns

If the user cancelled taking a photo, returns `{ cancelled: true }`.

Otherwise, returns `{ cancelled: false, uri, width, height, exif, base64 }` where `uri` is a URI to the local image file (useable as the source for an `Image` element) and `width, height` specify the dimensions of the image. `base64` is included if the `base64` option was truthy, and is a string containing the JPEG data of the image in Base64--prepend that with `'data:image/jpeg;base64,'` to get a data URI, which you can use as the source for an `Image` element for example. `exif` is included if the `exif` option was truthy, and is an object containing EXIF data for the image--the names of its properties are EXIF tags and their values are the values for those tags.

When you run this example and pick an image, you will see the image that you picked show up in your app, and something similar to the following logged to your console:

```
{  
  "cancelled":false,
```

```
    "height":1611,  
    "width":2148,  
    "uri":"file:///data/user/0/host.exp.exponent/cache/cropped1814158652.jpg"  
}
```

IntentLauncherAndroid

Provides a way to launch android intents. e.g. - opening a specific settings screen.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { IntentLauncherAndroid as IntentLauncher } from 'expo';  
  
// in bare apps:  
import * as IntentLauncher from 'expo-intent-launcher';
```

IntentLauncher.startActivityAsync(activityAction, intentParams)

Starts the specified activity. The method will return a promise which resolves when the user returns to the app.

Arguments

- **activityAction (string)** -- The action to be performed, e.g. `IntentLauncher.ACTION_WIRELESS_SETTINGS`. There are a few pre-defined constants you can use for this parameter. You can find them at [expo-intent-launcher/src/IntentLauncher.ts](#). **Required**
- **intentParams (IntentParams)** -- An object of intent parameters.

Returns

A promise resolving to an object of type [IntentResult](#).

Types

Type IntentParams

Key	Type	Description
type	string	A string specifying the MIME type of the data represented by the <code>data</code> parameter. Ignore this argument to allow Android to infer the correct MIME type.
category	string	Category provides more details about the action the intent performs. See Intent.addCategory).
extra	object	A map specifying additional key-value pairs which are passed with the intent as <code>extras</code> . The keys must include a package prefix, for example the app <code>com.android.contacts</code> would use names like

		<code>com.android.contacts.ShowAll</code> .
data	string	A URI specifying the data that the intent should operate upon. (<i>Note: Android requires the URI scheme to be lowercase, unlike the formal RFC.</i>)
flags	number	Bitmask of flags to be used. See Intent.setFlags) for more details.
packageName	string	Package name used as an identifier of ComponentName. Set this only if you want to explicitly set the component to handle the intent.
className	string	Class name of the ComponentName.

Type IntentResult

Key	Type	Description
resultCode	number	Result code returned by the activity. See resultCode for more details.
data	string	Optional data URI that can be returned by the activity.
extra	object	Optional extras object that can be returned by the activity.

Enums

Enum ResultCode

Result code	Value	Description
Success	-1	Indicates that the activity operation succeeded.
Canceled	0	Means that the activity was canceled, e.g. by tapping on the back button.
FirstUser	1	First custom, user-defined value that can be returned by the activity.

Example

```
import { IntentLauncher } from 'expo';

// Open location settings
IntentLauncher.startActivityAsync(IntentLauncher.ACTION_LOCATION_SOURCE_SETTINGS);
```

Accelerometer

Access the device accelerometer sensor(s) to respond to changes in acceleration in 3d space.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Accelerometer } from 'expo';  
  
// in bare apps:  
import { Accelerometer } from 'expo-sensors';
```

Accelerometer.isAvailableAsync()

Returns whether the accelerometer is enabled on the device.

Returns

- A promise that resolves to a `boolean` denoting the availability of the sensor.

Accelerometer.addListener(listener)

Subscribe for updates to the accelerometer.

Arguments

- **listener (function)** -- A callback that is invoked when an accelerometer update is available. When invoked, the listener is provided a single argument that is an object containing keys x, y, z.

Returns

- A subscription that you can call `remove()` on when you would like to unsubscribe the listener.

Accelerometer.removeAllListeners()

Remove all listeners.

Accelerometer.setUpdateInterval(intervalMs)

Subscribe for updates to the accelerometer.

Arguments

- **intervalMs (number)** Desired interval in milliseconds between accelerometer updates.

Example: basic subscription

```

import React from 'react';
import { StyleSheet, Text, TouchableOpacity, View } from 'react-native';
import { Accelerometer } from 'expo';

export default class AccelerometerSensor extends React.Component {
  state = {
    accelerometerData: {},
  };

  componentDidMount() {
    this._toggle();
  }

  componentWillUnmount() {
    this._unsubscribe();
  }

  _toggle = () => {
    if (this._subscription) {
      this._unsubscribe();
    } else {
      this._subscribe();
    }
  };

  _slow = () => {
    /* @info Request updates every 1000ms */ Accelerometer.setUpdateInterval(1000); /* @end */
  };

  _fast = () => {
    /* @info Request updates every 16ms, which is approximately equal to every frame at 60 frames per second */ Accelerometer.setUpdateInterval(
      16
    ); /* @end */
  };

  _subscribe = () => {
    /* @info Subscribe to events and update the component state with the new data from the Accelerometer. We save the
    subscription object away so that we can remove it when the component is unmounted*/ this._subscription = Accelerometer.addListener(
      accelerometerData => {
        this.setState({ accelerometerData });
      }
    ); /* @end */
  };

  _unsubscribe = () => {
    /* @info Be sure to unsubscribe from events when the component is unmounted */ this._subscription && this._subscription.remove(); /* @end */
  };

  this._subscription = null;
};

render() {
  /* @info A data point is provided for each of the x, y, and z axes */ let {
    x,
  }
}

```

```

        y,
        z,
    } = this.state.accelerometerData; /* @end */

    return (
        <View style={styles.sensor}>
            <Text>Accelerometer:</Text>
            <Text>
                x: {round(x)} y: {round(y)} z: {round(z)}
            </Text>

            <View style={styles.buttonContainer}>
                <TouchableOpacity onPress={this._toggle} style={styles.button}>
                    <Text>Toggle</Text>
                </TouchableOpacity>
                <TouchableOpacity onPress={this._slow} style={[styles.button, styles.middleButton]}>
                    <Text>Slow</Text>
                </TouchableOpacity>
                <TouchableOpacity onPress={this._fast} style={styles.button}>
                    <Text>Fast</Text>
                </TouchableOpacity>
            </View>
        </View>
    );
}

function round(n) {
    if (!n) {
        return 0;
    }

    return Math.floor(n * 100) / 100;
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
    },
    buttonContainer: {
        flexDirection: 'row',
        alignItems: 'stretch',
        marginTop: 15,
    },
    button: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#eee',
        padding: 10,
    },
    middleButton: {
        borderLeftWidth: 1,
        borderRightWidth: 1,
        borderColor: '#ccc',
    },
    sensor: {
        marginTop: 15,
        paddingHorizontal: 10,
    },
});

```

#

KeepAwake

A React component that prevents the screen sleeping when rendered. It also exposes static methods to control the behavior imperatively.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

```
// in managed apps:  
import { KeepAwake } from 'expo';  
  
// in bare apps:  
import KeepAwake from 'expo-keep-awake';
```

Example: component

```
import React from 'react';  
import { Text, View } from 'react-native';  
import { KeepAwake } from 'expo';  
  
export default class KeepWakeExample extends React.Component {  
  render() {  
    return (  
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>  
        /* @info As long as this component is mounted, the screen will not turn off from being idle. */ <KeepAwake />  
      /* @end */  
  
        <Text>This screen will never sleep!</Text>  
      </View>  
    );  
  }  
}
```

Example: static methods

```
import React from 'react';  
import { Button, View } from 'react-native';  
import { KeepAwake } from 'expo';  
  
export default class KeepWakeExample extends React.Component {  
  render() {  
    return (  
      <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>  
        <Button onPress={this._activate}>Activate</Button>  
        <Button onPress={this._deactivate}>Deactivate</Button>  
      </View>  
    );  
  }  
}  
  
KeepAwake.activate();  
KeepAwake.deactivate();
```

```
_activate = () => {
    /* @info Screen will remain on after called until <strong>KeepAwake.deactivate()</strong> is called. */KeepAwake.activate();/* @end */
}

_deactivate = () => {
    /* @info Deactivates KeepAwake, or does nothing if it was never activated. */KeepAwake.deactivate();/* @end */
}
```

LinearGradient

A React component that renders a gradient view.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

API

```
// in managed apps:  
import { LinearGradient } from 'expo';  
  
// in bare apps:  
import { LinearGradient } from 'expo-linear-gradient';
```

props

`colors`

An array of colors that represent stops in the gradient. At least two colors are required (otherwise it's not a gradient, it's just a fill!).

`start`

An array of `[x, y]` where x and y are floats. They represent the position that the gradient starts at, as a fraction of the overall size of the gradient. For example, `[0.1, 0.2]` means that the gradient will start 10% from the left and 20% from the top.

`end`

Same as start but for the end of the gradient.

`locations`

An array of the same length as `colors`, where each element is a float with the same meaning as the `start` and `end` values, but instead they indicate where the color at that index should be.

Linking

This module allows your app to interact with other apps via deep links. It provides helper methods for constructing and parsing deep links into your app.

This module is an extension of the React Native [Linking module](#), meaning that all methods in the RN module can be accessed via `Linking`, on top of the extra methods provided by Expo (detailed here). **These methods only apply to the managed workflow, you cannot use them in a bare React Native app.**

API

`Linking.makeUrl(path, queryParams)`

Helper method for constructing a deep link into your app, given an optional path and set of query parameters.

Arguments

- **path (string)** -- Any path into your app.
- **queryParams (object)** -- An object with a set of query parameters. These will be merged with any Expo-specific parameters that are needed (e.g. release channel) and then appended to the url as a query string.

Returns

A URL string which points to your app with the given deep link information.

`Linking.parse(url)`

Helper method for parsing out deep link information from a URL.

Arguments

- **url (string)** -- A URL that points to the currently running experience (e.g. an output of `Linking.makeUrl()`).

Returns

An object with the following keys:

- **path (string)** -- The path into the app specified by the url.
- **queryParams (object)** -- The set of query parameters specified by the query string of the url.

`Linking.parseInitialURLAsync()`

Helper method which wraps React Native's `Linking.getInitialURL()` in `Linking.parse()`. Parses the deep link information out of the URL used to open the experience initially.

Returns

A promise that resolves to an object with the following keys:

- **path (*string*)** -- The path specified by the url used to open the app.
- **queryParams (*object*)** -- The set of query parameters specified by the query string of the url used to open the app.

LocalAuthentication

Use FaceID and TouchID (iOS) or the Fingerprint API (Android) to authenticate the user with a face or fingerprint scan.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { LocalAuthentication } from 'expo';  
  
// in bare apps:  
import * as LocalAuthentication from 'expo-local-authentication';
```

LocalAuthentication.hasHardwareAsync()

Determine whether a face or fingerprint scanner is available on the device.

Returns

Returns a promise resolving to boolean value indicating whether a face or fingerprint scanner is available on this device.

LocalAuthentication.supportedAuthenticationTypesAsync()

Determine what kinds of authentications are available on the device.

Returns

Returns a promise resolving to an array containing `LocalAuthentication.AuthenticationType.{FINGERPRINT, FACIAL_RECOGNITION}`. A value of `1` indicates Fingerprint support and `2` indicates Facial Recognition support. Eg: `[1,2]` means the device has both types supported.

LocalAuthentication.isEnrolledAsync()

Determine whether the device has saved fingerprints or facial data to use for authentication.

Returns

Returns a promise resolving to boolean value indicating whether the device has saved fingerprints or facial data for authentication.

LocalAuthentication.authenticateAsync()

Attempts to authenticate via Fingerprint (or FaceID on iPhone X).

Note: When using the fingerprint module on Android, you need to provide a UI component to prompt the user to scan their fingerprint, as the OS has no default alert for it.

Note: Apple requires apps which use FaceID to provide a description of why they use this API. If you try to use FaceID on an iPhone with FaceID without providing `infoPlist.NSFaceIDUsageDescription` in `app.json`, the module will authenticate using device passcode. For more information about usage descriptions on iOS, see [Deploying to App Stores](#).

Arguments

- (**iOS only**) **promptMessage (string)** A message that is shown alongside the TouchID or FaceID prompt.

Returns

Returns a promise resolving to an object containing `success`, a boolean indicating whether or not the authentication was successful, and `error` containing the error code in the case where authentication fails.

LocalAuthentication.cancelAuthenticate() - (Android Only)

Cancels the fingerprint authentication flow.

Localization

You can use this module to Localize your app, and access the locale data on the native device. Using the popular library `i18n-js` with `expo-localization` will enable you to create a very accessible experience for users.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

```
import React from 'react';
import { Text } from 'react-native';
import { Localization } from 'expo-localization';
import i18n from 'i18n-js';
const en = {
  foo: 'Foo',
  bar: 'Bar {{someValue}}',
};
const fr = {
  foo: 'como telle fous',
  bar: 'chatouiller {{someValue}}',
};

i18n.fallbacks = true;
i18n.translations = { fr, en };
i18n.locale = Localization.locale;
export default class LitView extends React.Component {
  render() {
    return (
      <Text>
        {i18n.t('foo')} {i18n.t('bar', { someValue: Date.now() })}
      </Text>
    );
  }
}
```

API

```
// in managed apps:
import { Localization } from 'expo';

// in bare apps:
import * as Localization from 'expo-local-authentication';
```

Constants

This API is mostly synchronous and driven by constants. On iOS the constants will always be correct, on Android you should check if the locale has updated using `AppState` and `Localization.getLocalizationAsync()`. Initially the constants will be correct on both platforms, but on Android a user can change the language and return, more on

this later.

Localization.locale: string

Native device language, returned in standard format. Ex: en , en-US , es-US .

Localization.locales: Array<string>

List of all the native languages provided by the user settings. These are returned in the order the user defines in their native settings.

Localization.country: ?string

Country code for your device.

Localization.isoCurrencyCodes: ?Array<string>

A list of all the supported ISO codes.

Localization.timezone: string

The current time zone in display format. ex: America/Los_Angeles

On Web `timezone` is calculated with `Intl.DateTimeFormat().resolvedOptions().timeZone`. For a better guess you could use the `moment-timezone` library but is a very large library and will add significant bloat to your bundle.

Localization.isRTL: boolean

This will return `true` if the current language is Right-to-Left.

Methods

Localization.getLocalizationAsync(): Promise<Localization>

Android only, on iOS changing the locale settings will cause all the apps to reset.

```
type NativeEvent = {
  locale: string,
  locales: Array<string>,
  timezone: string,
  isoCurrencyCodes: ?Array<string>,
  country: ?string,
  isRTL: boolean,
};
```

Example

```
// When the app returns from the background on Android...

const { locale } = await Localization.getLocalizationAsync();
```


Location

This module allows reading geolocation information from the device. Your app can poll for the current location or subscribe to location update events.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

You must request permission to access the user's location before attempting to get it. To do this, you will want to use the [Permissions](#) API. You can see this in practice in the following example.

API

```
// in managed apps:  
import { Location } from 'expo';  
  
// in bare apps:  
import * as Location from 'expo-location';
```

Location.hasServicesEnabledAsync()

Checks whether location services are enabled by the user.

Returns

Returns a promise resolving to `true` if location services are enabled on the device, or `false` if not.

Location.requestPermissionsAsync()

Requests the user for location permissions, similarly to `Permissions.askAsync(Permissions.LOCATION)`.

Returns

Returns a promise that resolves when the permissions are granted and rejects when denied.

Location.getCurrentPositionAsync(options)

Get the current position of the device.

Arguments

- **options (object)** -- A map of options:
 - **accuracy** : [Location.Accuracy](#) -- Location manager accuracy. Pass one of [Location.Accuracy](#) enum

values. For low-accuracy the implementation can avoid geolocation providers that consume a significant amount of power (such as GPS).

- **maximumAge (number)** -- (Android only). If specified, allow returning a previously cached position that is at most this old in milliseconds. If not specified, always gets a new location. On iOS this option is ignored and a new location is always returned.

Returns

Returns a promise resolving to an object representing [Location](#) type.

Location.watchPositionAsync(options, callback)

Subscribe to location updates from the device. Please note that updates will only occur while the application is in the foreground. To get location updates while in background you'll need to use [Location.startLocationUpdatesAsync](#).

Arguments

- **options (object)** -- A map of options:
 - **accuracy : Location.Accuracy** -- Location manager accuracy. Pass one of [Location.Accuracy](#) enum values. For low accuracy the implementation can avoid geolocation providers that consume a significant amount of power (such as GPS).
 - **timeInterval (number)** -- Minimum time to wait between each update in milliseconds.
 - **distanceInterval (number)** -- Receive updates only when the location has changed by at least this distance in meters.
 - **mayShowUserSettingsDialog (boolean)** -- Specifies whether to ask the user to turn on improved accuracy location mode which uses Wi-Fi, cell networks and GPS sensor. The dialog can be shown only when the location mode is set to **Device only**. Defaults to `true`. (**Android only**)
- **callback (function)** --

This function is called on each location update. It is passed exactly one parameter: an object representing [Location](#) type.

Returns

Returns a promise resolving to a subscription object, which has one field:

- **remove (function)** -- Call this function with no arguments to remove this subscription. The callback will no longer be called for location updates.

Location.getProviderStatusAsync()

Check status of location providers.

Returns

Returns a promise resolving to an object with the following fields:

- **locationServicesEnabled (boolean)** -- Whether location services are enabled. See [Location.hasServicesEnabledAsync](#) for a more convenient solution to get this value.

- **gpsAvailable (boolean)** (android only) -- If the GPS provider is available, if yes, location data will be from GPS.
- **networkAvailable (boolean)** (android only) -- If the network provider is available, if yes, location data will be from cellular network.
- **passiveAvailable (boolean)** (android only) -- If the passive provider is available, if yes, location data will be determined passively.

Location.getHeadingAsync()

Gets the current heading information from the device

Returns

Object with:

- **magHeading (number)** — measure of magnetic north in degrees
- **trueHeading (number)** — measure of true north in degrees (needs location permissions, will return -1 if not given)
- **accuracy (number)** — level of calibration of compass.
 - 3: high accuracy, 2: medium accuracy, 1: low accuracy, 0: none
 - Reference for iOS: 3: < 20 degrees uncertainty, 2: < 35 degrees, 1: < 50 degrees, 0: > 50 degrees

Location.watchHeadingAsync(callback)

Subscribe to compass updates from the device.

Arguments

- **callback (function)** --

This function is called on each compass update. It is passed exactly one parameter: an object with the following fields:

- **magHeading (number)** — measure of magnetic north in degrees
- **trueHeading (number)** — measure of true north in degrees (needs location permissions, will return -1 if not given)
- **accuracy (number)** — level of calibration of compass.
 - 3: high accuracy, 2: medium accuracy, 1: low accuracy, 0: none
 - Reference for iOS: 3: < 20 degrees uncertainty, 2: < 35 degrees, 1: < 50 degrees, 0: > 50 degrees

Returns

Returns a promise resolving to a subscription object, which has one field:

- **remove (function)** — Call this function with no arguments to remove this subscription. The callback will no longer be called for location updates.

Location.geocodeAsync(address)

Geocode an address string to latitude-longitude location.

Note: Geocoding is resource consuming and has to be used reasonably. Creating too many requests at a time can result in an error so they have to be managed properly.

On Android, you must request a location permission (`Permissions.LOCATION`) from the user before geocoding can be used.

Arguments

- **address (string)** -- A string representing address, eg. "Baker Street London"

Returns

Returns a promise resolving to an array (in most cases its size is 1) of geocoded location objects with the following fields:

- **latitude (number)** -- The latitude in degrees.
- **longitude (number)** -- The longitude in degrees.
- **altitude (number)** -- The altitude in meters above the WGS 84 reference ellipsoid.
- **accuracy (number)** -- The radius of uncertainty for the location, measured in meters.

Location.reverseGeocodeAsync(location)

Reverse geocode a location to postal address.

Note: Geocoding is resource consuming and has to be used reasonably. Creating too many requests at a time can result in an error so they have to be managed properly.

On Android, you must request a location permission (`Permissions.LOCATION`) from the user before geocoding can be used.

Arguments

- **location (object)** -- An object representing a location:
 - **latitude (number)** -- The latitude of location to reverse geocode, in degrees.
 - **longitude (number)** -- The longitude of location to reverse geocode, in degrees.

Returns

Returns a promise resolving to an array (in most cases its size is 1) of address objects with following fields:

- **city (string)** -- City name of the address.
- **street (string)** -- Street name of the address.
- **region (string)** -- Region/area name of the address.
- **postalCode (string)** -- Postal code of the address.
- **country (string)** -- Localized country name of the address.
- **name (string)** -- Place name of the address, for example, "Tower Bridge".

Location.setApiKey(apiKey)

Sets a Google API Key for using Geocoding API. This method can be useful for Android devices that do not have Google Play Services, hence no Geocoder Service. After setting the key using Google's API will be possible.

Arguments

- **apiKey (string)** -- API key collected from Google Developer site.

Location.installWebGeolocationPolyfill()

Polyfills `navigator.geolocation` for interop with the core React Native and Web API approach to geolocation.

Background Location

Background Location API can notify your app about new locations, also while it's in background. There are some requirements in order to use Background Location API:

- `Permissions.LOCATION` permission must be granted. On iOS it must be granted with `Always` option — see [Permissions.LOCATION](#) for more details.
- "location" background mode must be specified in `Info.plist` file. See [background tasks configuration guide](#). (iOS only)
- Background location task must be defined in the top-level scope, using `TaskManager.defineTask`.

Location.startLocationUpdatesAsync(taskName, options)

Registers for receiving location updates that can also come when the app is in the background.

Arguments

- **taskName (string)** -- Name of the task receiving location updates.
- **options (object)** -- An object of options passed to the location manager.
 - **accuracy : Location.Accuracy** -- Location manager accuracy. Pass one of `Location.Accuracy` enum values. For low-accuracy the implementation can avoid geolocation providers that consume a significant amount of power (such as GPS).
 - **timeInterval (number)** -- Minimum time to wait between each update in milliseconds. Default value depends on `accuracy` option. (Android only)
 - **distanceInterval (number)** -- Receive updates only when the location has changed by at least this distance in meters. Default value may depend on `accuracy` option.
 - **showsBackgroundLocationIndicator (boolean)** -- A boolean indicating whether the status bar changes its appearance when location services are used in the background. Defaults to `false`. (Takes effect only on iOS 11.0 and later)

Returns

A promise resolving once the task with location updates is registered.

Task parameters

Background location task will be receiving following data:

- **locations : Location[]** - An array of the new locations.

```
import { TaskManager } from 'expo';

TaskManager.defineTask(YOUR_TASK_NAME, ({ data: { locations }, error }) => {
```

```
if (error) {
    // check `error.message` for more details.
    return;
}
console.log('Received new locations', locations);
});
```

Location.stopLocationUpdatesAsync(taskName)

Stops location updates for given task.

Arguments

- **taskName (string)** -- Name of the background location task to stop.

Returns

A promise resolving as soon as the task is unregistered.

Location.hasStartedLocationUpdatesAsync(taskName)

Arguments

- **taskName (string)** -- Name of the location task to check.

Returns

A promise resolving to boolean value indicating whether the location task is started or not.

Geofencing

Geofencing API notifies your app when the device enters or leaves geographical regions you set up. To make it work in the background, it uses [TaskManager](#) Native API under the hood. There are some requirements in order to use Geofencing API:

- `Permissions.LOCATION` permission must be granted. On iOS it must be granted with `Always` option — see [Permissions.LOCATION](#) for more details.
- "location" background mode must be specified in `Info.plist` file. See [background tasks configuration guide](#). (iOS only)
- Geofencing task must be defined in the top-level scope, using [TaskManager.defineTask](#).

Location.startGeofencingAsync(taskName, regions)

Starts geofencing for given regions. When the new event comes, the task with specified name will be called with the region that the device enter to or exit from. If you want to add or remove regions from already running geofencing task, you can just call `startGeofencingAsync` again with the new array of regions.

Arguments

- **taskName (string)** -- Name of the task that will be called when the device enters or exits from specified

regions.

- **regions (array)** -- Array of region objects to be geofenced.
 - **identifier (string)** -- The identifier of the region object. Defaults to auto-generated UUID hash.
 - **latitude (number)** -- The latitude in degrees of region's center point. *required*
 - **longitude (number)** -- The longitude in degrees of region's center point. *required*
 - **radius (number)** -- The radius measured in meters that defines the region's outer boundary. *required*
 - **notifyOnEnter (boolean)** -- Boolean value whether to call the task if the device enters the region. Defaults to `true`.
 - **notifyOnExit (boolean)** -- Boolean value whether to call the task if the device exits the region. Defaults to `true`.

Returns

A promise resolving as soon as the task is registered.

Task parameters

Geofencing task will be receiving following data:

- **eventType : [Location.GeofencingEventType](#)** -- Indicates the reason for calling the task, which can be triggered by entering or exiting the region. See [Location.GeofencingEventType](#).
- **region : [Region](#)** -- Object containing details about updated region. See [Region](#) for more details.

```
import { Location, TaskManager } from 'expo';

TaskManager.defineTask(YOUR_TASK_NAME, ({ data: { eventType, region }, error }) => {
  if (error) {
    // check `error.message` for more details.
    return;
  }
  if (eventType === Location.GeofencingEventType.Enter) {
    console.log("You've entered region:", region);
  } else if (eventType === Location.GeofencingEventType.Exit) {
    console.log("You've left region:", region);
  }
});
```

Location.stopGeofencingAsync(taskName)

Stops geofencing for specified task. It unregisters the background task so the app will not be receiving any updates, especially in the background.

Arguments

- **taskName (string)** -- Name of the task to unregister.

Returns

A promise resolving as soon as the task is unregistered.

Location.hasStartedGeofencingAsync(taskName)

Arguments

- **taskName (string)** -- Name of the geofencing task to check.

Returns

A promise resolving to boolean value indicating whether the geofencing task is started or not.

Types

Type Location

Object of type `Location` contains following keys:

- **coords (object)** -- The coordinates of the position, with the following fields:
 - **latitude (number)** -- The latitude in degrees.
 - **longitude (number)** -- The longitude in degrees.
 - **altitude (number)** -- The altitude in meters above the WGS 84 reference ellipsoid.
 - **accuracy (number)** -- The radius of uncertainty for the location, measured in meters.
 - **altitudeAccuracy (number)** -- The accuracy of the altitude value, in meters (iOS only).
 - **heading (number)** -- Horizontal direction of travel of this device, measured in degrees starting at due north and continuing clockwise around the compass. Thus, north is 0 degrees, east is 90 degrees, south is 180 degrees, and so on.
 - **speed (number)** -- The instantaneous speed of the device in meters per second.
- **timestamp (number)** -- The time at which this position information was obtained, in milliseconds since epoch.

Type Region

Object of type `Region` includes following fields:

- **identifier (string)** -- The identifier of the region object passed to `startGeofencingAsync` or auto-generated.
- **latitude (number)** -- The latitude in degrees of region's center point.
- **longitude (number)** -- The longitude in degrees of region's center point.
- **radius (number)** -- The radius measured in meters that defines the region's outer boundary.
- **state : Location.GeofencingRegionState** -- One of `Location.GeofencingRegionState` region state.

Determines whether the device is inside or outside a region.

Enums

Location.Accuracy

Accuracy	Value	Description
Accuracy.Lowest	1	Accurate to the nearest three kilometers.
Accuracy.Low	2	Accurate to the nearest kilometer.
Accuracy.Balanced	3	Accurate to within one hundred meters.
Accuracy.High	4	Accurate to within ten meters of the desired target.

Accuracy.Highest	5	The best level of accuracy available.
Accuracy.BestForNavigation	6	The highest possible accuracy that uses additional sensor data to facilitate navigation apps.

Location.GeofencingEventType

Event type	Value	Description
GeofencingEventType.Enter	1	Emitted when the device entered observed region.
GeofencingEventType.Exit	2	Occurs as soon as the device left observed region.

Location.GeofencingRegionState

Region state	Value	Description
GeofencingRegionState.Inside	1	Indicates that the device is inside the region.
GeofencingRegionState.Outside	2	Inverse of inside state.

Lottie

Expo includes support for [Lottie](#), the animation library from AirBnB.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow the [lottie-react-native installation instructions](#).

Usage

Importing Lottie

The Lottie SDK currently lives under Expo's **DangerZone** namespace because its implementation is still in Alpha. You can import it like this:

```
import { DangerZone } from 'expo';
let { Lottie } = DangerZone;
```

Using the Lottie API

We pull in the API from [lottie-react-native](#), so the documentation there is the best resource to follow.

Magnetometer

Access the device magnetometer sensor(s) to respond to measure the changes in the magnetic field. You can access the calibrated values with `Magnetometer`. and uncalibrated raw values with `MagnetometerUncalibrated`.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Magnetometer } from 'expo';  
  
// in bare apps:  
import { Magnetometer } from 'expo-sensors';
```

`Magnetometer.isAvailableAsync()`

Returns whether the magnetometer is enabled on the device.

Returns

- A promise that resolves to a `boolean` denoting the availability of the sensor.

`Magnetometer.addListener(listener)`

Subscribe for updates to the Magnetometer.

Arguments

- **listener (function)** -- A callback that is invoked when an Magnetometer update is available. When invoked, the listener is provided a single argument that is an object containing keys x, y, z.

Returns

- A subscription that you can call `remove()` on when you would like to unsubscribe the listener.

`Magnetometer.removeAllListeners()`

Remove all listeners.

`Magnetometer.setUpdateInterval(intervalMs)`

Subscribe for updates to the Magnetometer.

Arguments

- **intervalMs (number)** Desired interval in milliseconds between Magnetometer updates.

Example: basic subscription

```
import React from 'react';
import { Magnetometer } from 'expo';
import { StyleSheet, Text, TouchableOpacity, View } from 'react-native';

export default class MagnetometerSensor extends React.Component {
  state = {
    MagnetometerData: {},
  };

  componentDidMount() {
    this._toggle();
  }

  componentWillUnmount() {
    this._unsubscribe();
  }

  _toggle = () => {
    if (this._subscription) {
      this._unsubscribe();
    } else {
      this._subscribe();
    }
  };

  _slow = () => {
    Magnetometer.setUpdateInterval(1000);
  };

  _fast = () => {
    Magnetometer.setUpdateInterval(16);
  };

  _subscribe = () => {
    this._subscription = Magnetometer.addListener(result => {
      this.setState({ MagnetometerData: result });
    });
  };

  _unsubscribe = () => {
    this._subscription && this._subscription.remove();
    this._subscription = null;
  };
}

render() {
  let { x, y, z } = this.state.MagnetometerData;

  return (
    <View style={styles.sensor}>
      <Text>Magnetometer:</Text>
      <Text>
        x: {round(x)} y: {round(y)} z: {round(z)}
      </Text>

      <View style={styles.buttonContainer}>
        <TouchableOpacity onPress={this._toggle} style={styles.button}>
          <Text>Toggle</Text>
        </TouchableOpacity>
      </View>
    </View>
  );
}
```

```

        </TouchableOpacity>
        <TouchableOpacity onPress={this._slow} style={[styles.button, styles.middleButton]}>
            <Text>Slow</Text>
        </TouchableOpacity>
        <TouchableOpacity onPress={this._fast} style={styles.button}>
            <Text>Fast</Text>
        </TouchableOpacity>
    </View>
</View>
);
}
}

function round(n) {
if (!n) {
    return 0;
}

return Math.floor(n * 100) / 100;
}

const styles = StyleSheet.create({
container: {
    flex: 1,
},
buttonContainer: {
    flexDirection: 'row',
    alignItems: 'stretch',
    marginTop: 15,
},
button: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#eee',
    padding: 10,
},
middleButton: {
    borderLeftWidth: 1,
    borderRightWidth: 1,
    borderColor: '#ccc',
},
sensor: {
    marginTop: 15,
    paddingHorizontal: 10,
},
});
}

```

MailComposer

An API to compose mails using OS specific UI.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { MailComposer } from 'expo';  
  
// in bare apps:  
import * as MailComposer from 'expo-mail-composer';
```

MailComposer.composeAsync(options)

Opens a mail modal for iOS and a mail app intent for Android and fills the fields with provided data.

Arguments

- **saveOptions (object)** -- A map defining the data to fill the mail:
 - **recipients (_array)** -- An array of e-mail addressess of the recipients.
 - **ccRecipients (array)** -- An array of e-mail addressess of the CC recipients.
 - **bccRecipients (array)** -- An array of e-mail addressess of the BCC recipients.
 - **subject (string)** -- Subject of the mail.
 - **body (string)** -- Body of the mail.
 - **isHtml (boolean)** -- Whether the body contains HTML tags so it could be formatted properly. Not working perfectly on Android.
 - **attachments (array)** -- An array of app's internal file uris to attach.

Returns

Resolves to a promise with object containing `status` field that could be either `sent`, `saved` or `cancelled`. Android does not provide such info so it always resolves to `sent`.

MapView

A Map component that uses Apple Maps or Google Maps on iOS and Google Maps on Android. Expo uses react-native-maps at [react-community/react-native-maps](#). No setup required for use within the Expo app, or within a standalone app for iOS. See below for instructions on how to configure for deployment as a standalone app on Android.

Expo includes version 0.22.0 of react-native-maps (the latest as of the time of this writing).

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow the [react-native-maps installation instructions](#).

Usage

See full documentation at [react-community/react-native-maps](#).

Configuration

Deploying to a standalone app on Android

If you have already integrated Google Sign In into your standalone app, this is very easy. Otherwise, there are some additional steps.

- **If you already have Google Sign In configured**
 1. Open your browser to the [Google API Manager](#).
 2. Select your project and enable the **Google Maps Android API**
 3. In `app.json`, copy the API key from `android.config.googleSignIn` to `android.config.googleMaps.apiKey`.
 4. Rebuild your standalone app.
- **If you already have not configured Google Sign In**
 1. Build your app, take note of your Android package name (eg: `ca.brentvatne.growlerprowler`)
 2. Open your browser to the [Google API Manager](#) and create a project.
 3. Once it's created, go to the project and enable the **Google Maps Android API**
 4. Go back to <https://console.developers.google.com/apis/credentials> and click **Create Credentials**, then **API Key**.
 5. In the modal that popped up, click **RESTRICT KEY**.
 6. Choose the **Android apps** radio button under **Key restriction**.
 7. Click the **+ Add package name and fingerprint** button.
 8. Add your `android.package` from `app.json` (eg: `ca.brentvatne.growlerprowler`) to the Package name field.
 9. Run `expo fetch:android:hashes`.
 10. Copy `Google Certificate Fingerprint` from the output from step 9 and insert it in the "SHA-1 certificate fingerprint" field.
 11. Copy the API key (the first text input on the page) into `app.json` under the `android.config.googleMaps.apiKey` field. [See an example diff](#).
 12. Press `Save` and then rebuild the app like in step 1.

Note that if you've enabled Google Play's app signing service, you will need to grab their app signing certificate in production rather than the upload certificate returned by `expo fetch:android:hashes`. You can do this by grabbing the signature from Play Console -> Your App -> Release management -> App signing, and then going to the [API Dashboard](#) -> Credentials and adding the signature to your existing credential.

Deploying Google Maps to a standalone app on iOS

Apple Maps should just work with no extra configuration. For Google Maps, you can specify your own Google Maps API key using the `ios.config.googleMapsApiKey` configuration in your project's `app.json`.

Deploying Google Maps to ExpoKit for iOS

If you want to add MapView with Google Maps to an [ExpoKit](#) (ejected) project on iOS, you may need to manually provide a key by calling:

```
[GMSServices provideApiKey:@"your api key"]
```

Alternatively, you can provide the `GMSApiKey` key in your app's `Info.plist` and ExpoKit will pick it up automatically. If you ejected after already configuring Google Maps, the eject step may have already provided this for you.

Web Setup

Web is experimental! You may need to add the web target to your Expo app.

To use this in web, add the following script to your `web/index.html`. This script may already be present, if this is the case, just replace the `API_KEY` with your Google Maps API key which you can obtain here: [Google Maps: Get API key](#)

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- At the end of the <head/> element... -->

    <script
      async
      defer
      src="https://maps.googleapis.com/maps/api/js?key=API_KEY"
      type="text/javascript"
    ></script>

    <!-- Use your web API Key in place of API_KEY: https://developers.google.com/maps/documentation/javascript/get-api-key -->
  </head>

  <!-- <body /> -->
</html>
```

MediaLibrary

Provides access to user's media library.

Requires `Permissions.CAMERA_ROLL` permissions.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { MediaLibrary } from 'expo';  
  
// in bare apps:  
import * as MediaLibrary from 'expo-media-library';
```

MediaLibrary.createAssetAsync(localUri)

Creates an asset from existing file. The most common use case is to save a picture taken by [Camera](#).

```
const { uri } = await camera.takePictureAsync();  
const asset = await MediaLibrary.createAssetAsync(uri);
```

Arguments

- **localUri (string)** -- A URI to the image or video file. On Android it must be a local path, so it must start with `file:///`.

Returns

An object representing an [asset](#).

MediaLibrary.getAssetsAsync(options)

Fetches a page of assets matching the provided criteria.

Arguments

- **options (object)**
 - **first (number)** -- The maximum number of items on a single page.
 - **after (string)** -- Asset ID of the last item returned on the previous page.
 - **album (string | Album)** -- [Album](#) or its ID to get assets from specific album.
 - **sortBy (array)** -- An array of [SortBy](#) keys. By default, all keys are sorted in descending order, however you can also pass a pair `[key, ascending]` where the second item is a `boolean`

- value that means whether to use ascending order. Earlier items have higher priority when sorting out the results. If empty, this method will use the default sorting that is provided by the platform.
- **mediaType (array)** -- An array of [MediaType](#) types. By default `MediaType.photo` is set.

Returns

A promise that resolves to an object that contains following keys:

- **assets (array)** -- A page of [assets](#) fetched by the query.
- **endCursor (string)** -- ID of the last fetched asset. It should be passed as `after` option in order to get the next page.
- **hasNextPage (boolean)** -- Whether there are more assets to fetch.
- **totalCount (number)** -- Estimated total number of assets that match the query.

MediaLibrary.getAssetInfoAsync(asset)

Provides more informations about an asset, including GPS location, local URI and EXIF metadata.

Arguments

- **asset (string | Asset)** -- [Asset](#) or its ID.

Returns

Asset object extended by additional fields listed [in the table](#).

MediaLibrary.deleteAssetsAsync(assets)

Deletes assets from the library. On iOS it deletes assets from all albums they belong to, while on Android it keeps all copies of them (album is strictly connected to the asset). Also, there is additional dialog on iOS that requires user to confirm this action.

Arguments

- **assets (array)** -- An array of [assets](#) or their IDs.

Returns

Returns `true` if the assets were successfully deleted.

MediaLibrary.getAlbumsAsync()

Queries for user-created albums in media gallery.

Returns

An array of [albums](#).

MediaLibrary.getAlbumAsync(albumName)

Queries for an album with a specific name.

Arguments

- **albumName (string)** -- Name of the album to look for.

Returns

An object representing an [album](#) if album with given name exists, otherwise returns `null`.

MediaLibrary.createAlbumAsync(albumName, asset, copyAsset)

Creates an album with given name and initial asset. The asset parameter is required on Android, since it's not possible to create empty album on this platform. On Android, by default it copies given asset from the current album to the new one, however it's also possible to move it by passing `false` as `copyAsset` argument. In case it's copied you should keep in mind that `getAssetsAsync` will return duplicated asset.

Arguments

- **albumName (string)** -- Name of the album to create.
- **asset (string | Asset)** -- [Asset](#) or its ID. Required on Android.
- **copyAsset (boolean)** -- Whether to copy asset to the new album instead of move it. Defaults to `true`.
(Android only)

Returns

Newly created [album](#).

MediaLibrary.deleteAlbumsAsync(albums, deleteAssets)

Deletes given albums from the library.

On Android by default it deletes assets belonging to given albums from the library. On iOS it doesn't delete these assets, however it's possible to do by passing `true` as `deleteAssets`.

Arguments

- **albums (array)** -- Array of [albums](#) or their IDs, that will be removed from the library.
- **deleteAssets (boolean)** -- Whether to also delete assets belonging to given albums. Defaults to `false`.
(iOS only)

Returns

Returns a promise resolving to `true` if the albums were successfully deleted from the library.

MediaLibrary.addAssetsToAlbumAsync(assets, album, copyAssets)

Adds array of assets to the album.

On Android, by default it copies assets from the current album to provided one, however it's also possible to move them by passing `false` as `copyAssets` argument. In case they're copied you should keep in mind that `getAssetsAsync` will return duplicated assets.

Arguments

- **assets (array)** -- Array of [assets](#) to add.
- **album (string | Album)** -- [Album](#) or its ID, to which the assets will be added.
- **copyAssets (boolean)** -- Whether to copy assets to the new album instead of move them. Defaults to `true`. **(Android only)**

Returns

Returns `true` if the assets were successfully added to the album.

MediaLibrary.removeAssetsFromAlbumAsync(assets, album)

Removes given assets from album.

On Android, album will be automatically deleted if there are no more assets inside.

Arguments

- **assets (array)** -- Array of [assets](#) to remove from album.
- **album (string | Album)** -- [Album](#) or its ID, from which the assets will be removed.

Returns

Returns `true` if the assets were successfully removed from the album.

MediaLibrary.getMomentsAsync()

Available on iOS only. Fetches a list of moments, which is a group of assets taken around the same place and time.

Returns

An array of [albums](#) whose type is `moment`.

MediaLibrary.addListener(listener)

Subscribes for updates in user's media library.

Arguments

- **listener (function)** -- A callback that is called when any assets have been inserted or deleted from the library. **On Android** it's invoked with an empty object. **On iOS** it's invoked with an object that contains following keys:
 - **insertedAssets (array)** -- Array of [assets](#) that have been inserted to the library.
 - **deletedAssets (array)** -- Array of [assets](#) that have been deleted from the library.

Returns

An `EventSubscription` object that you can call `remove()` on when you would like to unsubscribe the listener.

MediaLibrary.removeAllListeners()

Removes all listeners.

Types

Asset

Field name	Type	Platforms	Description	Possible values
id	string	both	Internal ID that represents an asset	
filename	string	both	Filename of the asset	
uri	string	both	URI that points to the asset	assets:///* (iOS), file:///* (Android)
mediaType	string	both	Media type	MediaType.audio , MediaType.photo , MediaType.video , MediaType.unknown
width	number	both	Width of the image or video	
height	number	both	Height of the image or video	
creationTime	number	both	File creation timestamp	
modificationTime	number	both	Last modification timestamp	
duration	number	both	Duration of the video or audio asset	
mediaSubtypes	array	iOS	An array of media subtypes	hdr , panorama , stream , timelapse , screenshot , highFrameRate , livePhoto , depthEffect
albumId	string	Android	Album ID that the asset belongs to	
localUri *	string	both	Local URI for the asset	
location *	object	both	GPS location if available	latitude: number, longitude: number or null
exif *	object	both	EXIF metadata associated with the image	
orientation *	number	iOS	Display orientation of the image	Numbers 1-8, see EXIF orientation specification
			Whether the	

isFavorite *	boolean	iOS	asset is marked as favorite	true , false
--------------	---------	-----	-----------------------------	--------------

* These fields can be obtained only by calling `getAssetInfoAsync` method

Album

Field name	Type	Platforms	Description	Possible values
id	string	both		
title	string	both		
assetCount	number	both	Estimated number of assets in the album	
type	string	iOS	The type of the assets album	album , moment , smartAlbum
startTime *	number	iOS	Earliest creation timestamp of all assets in the moment	
endTime *	number	iOS	Latest creation timestamp of all assets in the moment	
approximateLocation *	object	iOS	Approximated location of all assets in the moment	latitude: number, longitude: number Or null
locationNames *	array	iOS	Names of locations grouped in the moment	

* These fields apply only to albums whose type is `moment`

Constants

MediaLibrary.MediaType

Possible media types:

- `MediaType.photo`
- `MediaType.video`
- `MediaType.audio`
- `MediaType.unknown`

MediaLibrary.SortBy

Supported keys that can be used to sort `getAssetsAsync` results:

- `SortBy.default`
- `SortBy.id`
- `SortBy.creationTime`
- `SortBy.modificationTime`
- `SortBy.mediaType`
- `SortBy.width`

- `SortBy.height`
- `SortBy.duration`

Notifications

Provides access to remote notifications (also known as push notifications) and local notifications (scheduling and immediate) related functions.

Installation

This API is pre-installed in [managed](#) apps. It is not available for [bare](#) React Native apps, although there are some comparable libraries that you may use instead.

API

```
import { Notifications } from 'expo';
```

Subscribing to Notifications

Notifications.addListener(listener)

Arguments

- **listener (function)** -- A callback that is invoked when a remote or local notification is received or selected, with a Notification object.

Returns

An [EventSubscription](#) object that you can call `remove()` on when you would like to unsubscribe the listener.

Related types

EventSubscription

Returned from `addListener`.

- **remove() (function)** -- Unsubscribe the listener from future notifications. [Notification](#)

An object that is passed into each event listener when a notification is received:

- **origin (string)** -- Either `selected` OR `received`. `selected` if the notification was tapped on by the user, `received` if the notification was received while the user was in the app.
- **data (object)** -- Any data that has been attached with the notification.
- **remote (boolean)** -- `true` if the notification is a push notification, `false` if it is a local notification.

Notifications

Notifications.getExpoPushTokenAsync()

Returns

Returns a Promise that resolves to a token string. This token can be provided to the Expo notifications backend to send a push notification to this device. [Read more in the Push Notifications guide.](#)

The Promise will be rejected if the app does not have permission to send notifications. Be sure to check the result of `Permissions.askAsync(Permissions.NOTIFICATIONS)` before attempting to get an Expo push token.

Notifications.presentLocalNotificationAsync(localNotification)

Trigger a local notification immediately.

Arguments

- **localNotification (object)** -- An object with the properties described in [LocalNotification](#).

Returns

A Promise that resolves to a unique notification id.

Notifications.scheduleLocalNotificationAsync(localNotification, schedulingOptions)

Schedule a local notification to fire at some specific time in the future or at a given interval.

Arguments

- **localNotification (object)** --

An object with the properties described in [LocalNotification](#).

- **schedulingOptions (object)** --

An object that describes when the notification should fire.

- **time (date or number)** -- A Date object representing when to fire the notification or a number in Unix epoch time. Example: `(new Date()).getTime() + 1000` is one second from now.
- **repeat (optional) (string)** -- `'minute'`, `'hour'`, `'day'`, `'week'`, `'month'`, or `'year'`.
- **(Android only) intervalMs (optional) (number)** -- Repeat interval in number of milliseconds

Returns

A Promise that resolves to a unique notification id.

Notifications.dismissNotificationAsync(localNotificationId)

Android only. Dismisses the notification with the given id.

Arguments

- **localNotificationId (number)** -- A unique id assigned to the notification, returned from `scheduleLocalNotificationAsync` OR `presentLocalNotificationAsync` .

Notifications.dismissAllNotificationsAsync()

Android only. Clears any notifications that have been presented by the app.

Notifications.cancelScheduledNotificationAsync(localNotificationId)

Cancels the scheduled notification corresponding to the given id.

Arguments

- **localNotificationId (number)** -- A unique id assigned to the notification, returned from `scheduleLocalNotificationAsync` OR `presentLocalNotificationAsync` .

Notifications.cancelAllScheduledNotificationsAsync()

Cancel all scheduled notifications.

Notification categories

A notification category defines a set of actions with which a user may respond to the incoming notification. You can read more about it [here \(for iOS\)](#) and [here \(for Android\)](#).

Notifications.createCategoryAsync(name: string, actions: ActionType[])

Registers a new set of actions under given `name` .

Arguments

- **name (string)** -- A string to assign as the ID of the category. When you present notifications later, you will pass this ID in order to associate them with your category.
- **actions (array)** -- An array of objects describing actions to associate to the category, of shape:
 - **actionId (string)** -- A unique identifier of the ID of the action. When a user executes your action, your app will receive this `actionId` .
 - **buttonTitle (string)** -- A title of the button triggering this action.
 - **textInput (object)** -- An optional object of shape: `{ submitButtonTitle: string, placeholder: string }` , which when provided, will prompt the user to enter a text value.
 - **isDestructive (boolean)** -- (iOS only) If this property is truthy, on iOS the button title will be highlighted (as if [this native option](#) was set)
 - **isAuthenticationRequired (boolean)** -- (iOS only) If this property is truthy, triggering the action will require authentication from the user (as if [this native option](#) was set)

Notifications.deleteCategoryAsync(name: string)

Deletes category for given `name` .

Android channels

Notifications.createChannelAndroidAsync(id, channel)

Android only. On Android 8.0+, creates a new notification channel to which local and push notifications may be posted. Channels are visible to your users in the OS Settings app as "categories", and they can change settings or disable notifications entirely on a per-channel basis. NOTE: after calling this method, you may no longer be able to alter the settings for this channel, and cannot fully delete the channel without uninstalling the app. Notification channels are required on Android 8.0+, but use this method with caution and be sure to plan your channels carefully.

According to the [Android docs](#),

You should create a channel for each distinct type of notification you need to send. You can also create notification channels to reflect choices made by users of your app. For example, you can set up separate notification channels for each conversation group created by a user in a messaging app.

On devices with Android 7.1 and below, Expo will "polyfill" channels for you by saving your channel's settings and automatically applying them to any notifications you designate with the `channelId`.

Arguments

- **id (string)** -- A unique string to assign as the ID of this channel. When you present notifications later, you will pass this ID in order to associate them with your channel.
- **channel (object)** -- An object with the properties described in [ChannelAndroid](#).

Notifications.deleteChannelAndroidAsync(id)

Android only. On Android 8.0+, deletes the notification channel with the given ID. Note that the OS Settings UI will display the number of deleted notification channels to the user as a spam prevention mechanism, so the only way to fully delete a channel is to uninstall the app or clearing all app data.

Arguments

- **id (string)** -- ID string of the channel to delete.

Related types

LocalNotification

An object used to describe the local notification that you would like to present or schedule.

- **title (string)** -- title text of the notification
- **body (string)** -- body text of the notification.
- **data (optional) (object)** -- any data that has been attached with the notification.
- **categoryId (optional) (string)** -- ID of the category (first created with `Notifications.createCategoryAsync`) associated to the notification.
- **ios (optional) (object)** -- notification configuration specific to iOS.
 - **sound (optional) (boolean)** -- if `true`, play a sound. Default: `false`.
- **android (optional) (object)** -- notification configuration specific to Android.
 - **channelId (optional, but recommended) (string)** -- ID of the channel to post this notification to in Android

8.0+. If null, defaults to the "Default" channel which Expo will automatically create for you. If you don't want Expo to create a default channel, make sure to always specify this field for all notifications.

- **icon (optional) (string)** -- URL of icon to display in notification drawer.
- **color (optional) (string)** -- color of the notification icon in notification drawer.
- **sticky (optional) (boolean)** -- if `true`, the notification will be sticky and not dismissable by user. The notification must be programmatically dismissed. Default: `false`.
- **link (optional) (string)** -- external link to open when notification is selected.

ChannelAndroid

An object used to describe an Android notification channel that you would like to create.

- **name (string)** -- user-facing name of the channel (or "category" in the Settings UI). Required.
- **description (optional) (string)** -- user-facing description of the channel, which will be displayed in the Settings UI.
- **sound (optional) (boolean)** -- if `true`, notifications posted to this channel will play a sound. Default: `false`.
- **priority (optional) (min | low | default | high | max)** -- Android may present notifications in this channel differently according to the priority. For example, a `high` priority notification will likely to be shown as a heads-up notification. Note that the Android OS gives no guarantees about the user-facing behavior these abstractions produce -- for example, on many devices, there is no noticeable difference between `high` and `max`.
- **vibrate (optional) (boolean or array)** -- if `true`, vibrate the device whenever a notification is posted to this channel. An array can be supplied instead to customize the vibration pattern, e.g. - `[0, 500]` or `[0, 250, 250, 250]`. Default: `false`.
- **badge (optional) (boolean)** -- if `true`, unread notifications posted to this channel will cause the app launcher icon to be displayed with a badge on Android 8.0+. If `false`, notifications in this channel will never cause a badge. Default: `true`.

App Icon Badge Number (iOS)

`Notifications.getBadgeNumberAsync()`

Returns

Returns a promise that resolves to the number that is displayed in a badge on the app icon. This method returns zero when there is no badge (or when on Android).

`Notifications.setBadgeNumberAsync(number)`

Sets the number displayed in the app icon's badge to the given number. Setting the number to zero will both clear the badge and the list of notifications in the device's notification center on iOS. On Android this method does nothing.

Standalone App Only

`Notifications.getDevicePushTokenAsync(config)`

Note: Most people do not need to use this. It is easier to use `getExpoPushTokenAsync` unless you have a specific reason to need the actual device tokens. We also don't guarantee that the iOS and Android clients will continue expecting the same push notification payload format.

Returns a native APNS, FCM or GCM token that can be used with another push notification service. If firebase cloud messaging is configured on your standalone Android app ([see guide here](#)), it will return an FCM token, otherwise it will return a GCM token.

Arguments

- **config (object)** -- An object with the following fields:
 - **gcmSenderId (string)** -- GCM sender ID.

Returns

A Promise that resolves to an object with the following fields:

- **type (string)** -- Either "apns", "fcm", or "gcm".
- **data (string)** -- The push token as a string.

Amplitude

Provides access to [Amplitude](#) mobile analytics which basically lets you log various events to the Cloud. This module wraps Amplitude's [iOS](#) and [Android](#) SDKs. For a great example of usage, see the [Expo app source code](#).

Note: Session tracking may not work correctly when running Experiences in the main Expo app. It will work correctly if you create a standalone app.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Amplitude } from 'expo';  
  
// in bare apps:  
import * as Amplitude from 'expo-analytics-amplitude';
```

Amplitude.initialize(apiKey)

Initializes Amplitude with your Amplitude API key. If you're having trouble finding your API key, see [step 4 of these instructions](#).

Arguments

- **apiKey (string)** -- Your Amplitude application's API key.

Amplitude.setUserId(userId)

Assign a user ID to the current user. If you don't have a system for user IDs you don't need to call this. See [this page](#) for details.

Arguments

- **userId (string)** -- User ID for the current user.

Amplitude.setUserProperties(userProperties)

Set properties for the current user. See [here for details](#).

Arguments

- **userProperties (object)** -- A map of custom properties.

Amplitude.clearUserProperties()

Clear properties set by `Amplitude.setUserProperties()`.

`Amplitude.logEvent(eventName)`

Log an event to Amplitude. For more information about what kind of events to track, [see here](#).

Arguments

- **eventName (string)** -- The event name.

`Amplitude.logEventWithProperties(eventName, properties)`

Log an event to Amplitude with custom properties. For more information about what kind of events to track, [see here](#).

Arguments

- **eventName (string)** -- The event name.
- **properties (object)** -- A map of custom properties.

`Amplitude.setGroup(groupType, groupNames)`

Add the current user to a group. For more information, see here for [iOS](#) and see here for [Android](#).

Arguments

- **groupType (string)** -- The group name, e.g. "sports".
- **groupNames (object)** -- An array of group names, e.g. ["tennis", "soccer"]. Note: the iOS and Android Amplitude SDKs allow you to use a string or an array of strings. We only support an array of strings. Just use an array with one element if you only want one group name.

Payments

Expo includes support for payments through [Stripe](#) and [Apple Pay](#) on iOS via ExpoKit, and Stripe on Android (plus Android Pay via ExpoKit).

Need more help than what's on the page? The Payments module is largely based off [tipsi-stripe](#). The documentation and questions there may prove helpful.

We encourage you to look at our [examples](#) of ExpoKit apps.

Note: (Android only) If you are using Expo Client then the setup has already been done for you. Also, the way you should use payments is slightly different. Instead of importing from `'expo-payments-stripe'` use the following code:

```
import { DangerZone } from 'expo';
const { Stripe } = DangerZone;
```

Setup

If you haven't done payments with Stripe before, create an account with [Stripe](#). After getting the account set up, navigate to the [Stripe API dashboard](#). Here, you'll need to make a note of the Publishable key and Secret key listed.

Adding the Payments Module on iOS

The Payments module is currently only supported through `EXPaymentsStripe` pod on iOS.

First, eject your Expo project using ExpoKit (refer to [Eject to ExpoKit](#) for more information). Then, add `expo-payments-stripe` to the list of dependencies of your project and install the dependencies. Then, navigate to and open `your-project-name/ios/Podfile`. Add `EXPaymentsStripe` to your Podfile's subspecs. Example:

```
...
target 'your-project-name' do
  ...
  pod 'EXPaymentsStripe',
    :path => "../node_modules/expo-payments-stripe/ios",
    :inhibit_warnings => true
  ...
  pod 'React',
  ...

```

Finally, make sure [CocoaPods](#) is installed and run `pod install` in `your-project-name/ios`. This will add the Payments module files to your project and the corresponding dependencies.

Register hook in order to let Stripe process source authorization

You don't need to make this step if you're not going to use [sources](#).

Follow [Stripe instructions](#). If you have problems with this step just look at files: `Info.plist` and `AppDelegate.m` in one of our [examples](#).

Adding the Payments Module on Android

Note: These steps are required only if you have ejected your app with `SDK < 30`. If at the moment of ejecting you had `sdkVersion` set to 30 or higher in your `app.json`, the following setup should have been performed automatically.

1. Add these lines into your `settings.gradle` file.

```
include ':expo-payments-stripe'  
project(':expo-payments-stripe').projectDir = new File(rootProject.projectDir, '../node_modules/expo-payments-stripe/android')
```

1. Add dependencies in your `build.gradle` file.

```
implementation project(':expo-payments-stripe')
```

1. Force specific `com.android.support:design` version in your `build.gradle` file.

```
android {  
    ...  
    configurations.all {  
        resolutionStrategy.force 'com.android.support:design:27.1.0'  
    }  
    ...  
}
```

1. Exclude old version of `CreditCardEntry` in `your-project/android/app/build.gradle` file.

```
implementation('host.exp.exponent:expoview:29.0.0@aar') {  
    transitive = true  
    exclude group: 'com.squareup.okhttp3', module: 'okhttp'  
    exclude group: 'com.github.thefuntasty', module: 'CreditCardEntry' // add this line  
    exclude group: 'com.squareup.okhttp3', module: 'okhttp-urlconnection'  
}
```

1. Make sure your list of repositories in `build.gradle` contains `jitpack`.

```
allprojects {  
    repositories {  
        ...  
        maven { url "https://jitpack.io" }  
        ...  
    }  
}
```

Register hook in order to let Stripe process source authorization

You don't need to make this step if you're not going to use [sources](#).

Add the following code to your `AndroidManifest.xml`, replacing `your_scheme` with the URI scheme you're going to use when specifying return URL for payment process.

```
...
<activity
    android:exported="true"
    android:launchMode="singleTask"
    android:name="expo.modules.payments.stripe.RedirectUriReceiver"
    android:theme="@android:style/Theme.Translucent.NoTitleBar.Fullscreen">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />

        <data android:scheme="your_scheme" />
    </intent-filter>
</activity>
...
```

If you have problems with this step just look at `AndroidManifest.xml` in one of our [examples](#). Remember to use the same scheme as the one which was set in `Info.plist` file (only if you are also developing app for iOS).

Importing Payments

```
import { PaymentsStripe as Stripe } from 'expo-payments-stripe';

...
Stripe.setOptionsAsync({
    ...
});
```

Using the Payments SDK

First, initialize the Payments module with your credentials:

This `setOptionsAsync` method must put under the `componentWillMount` in android's production mode, unlike iOS that it works outside any component.

```
Stripe.setOptionsAsync({
    publishableKey: 'PUBLISHABLE_KEY', // Your key
    androidPayMode: 'test', // [optional] used to set wallet environment (AndroidPay)
    merchantId: 'your_merchant_id', // [optional] used for payments with ApplePay
});
```

Creating token [Android, iOS]

Creates token based on passed card params.

params — An object with the following keys:

Key	Type	Description
number (Required)	String	The card's number

expMonth (Required)	Number	The card's expiration month
expYear (Required)	Number	The card's expiration year
cvc	String	The card's security code, found on the back
name	String	The cardholder's name
addressLine1	String	The first line of the billing address
addressLine2	String	The second line of the billing address
addressCity	String	City of the billing address
addressState	String	State of the billing address
addressZip	String	Zip code of the billing address
addressCountry	String	Country for the billing address
brand (Android)	String	Brand of this card. Can be one of: JCB American Express Visa Discover Diners Club MasterCard Unknown
last4 (Android)	String	last 4 digits of the card
fingerprint (Android)	String	The card fingerprint
funding (Android)	String	The funding type of the card. Can be one of: debit credit prepaid unknown
country (Android)	String	ISO country code of the card itself
currency	String	Three-letter ISO currency code representing the currency paid out to the bank account. This is only applicable when tokenizing debit cards to issue payouts to managed accounts. You should not set it otherwise. The card can then be used as a transfer destination for funds in this currency

```

const params = {
  // mandatory
  number: '4242424242424242',
  expMonth: 11,
  expYear: 17,
  cvc: '223',
  // optional
  name: 'Test User',
  currency: 'usd',
  addressLine1: '123 Test Street',
  addressLine2: 'Apt. 5',
  addressCity: 'Test City',
  addressState: 'Test State',
  addressCountry: 'Test Country',
  addressZip: '55555',
};

const token = await stripe.createTokenWithCardAsync(params);

// Client specific code
// api.sendTokenToBackend(token)

```

Remember to initialize the Payments module before creating token.

Payment request with card form [Android, iOS]

Launch `Add Card` view to accept payment.

options (iOS only) — An object with the following keys:

Key	Type	Description
requiredBillingAddressFields	String	The billing address fields the user must fill out when prompted for their payment details. Can be one of: full or zip or not specify to disable
prefilledInformation	Object	You can set this property to pre-fill any information you've already collected from your user
managedAccountCurrency	String	Required to be able to add the card to an account (in all other cases, this parameter is not used). More info
theme	Object	Can be used to visually style Stripe-provided UI

options.prefilledInformation — An object with the following keys:

Key	Type	Description
email	String	The user's email address
phone	String	The user's phone number
billingAddress	Object	The user's billing address. When set, the add card form will be filled with this address

options.prefilledInformation.billingAddress — An object with the following keys:

Key	Type	Description
name	String	The user's full name (e.g. "Jane Doe")
line1	String	The first line of the user's street address (e.g. "123 Fake St")
line2	String	The apartment, floor number, etc of the user's street address (e.g. "Apartment 1A")
city	String	The city in which the user resides (e.g. "San Francisco")
state	String	The state in which the user resides (e.g. "CA")
postalCode	String	The postal code in which the user resides (e.g. "90210")
country	String	The ISO country code of the address (e.g. "US")
phone	String	The phone number of the address (e.g. "8885551212")
email	String	The email of the address (e.g. "jane@doe.com")

options.theme — An object with the following keys:

Key	Type	Description
primaryBackgroundColor	String	The primary background color of the theme
secondaryBackgroundColor	String	The secondary background color of this theme
		The primary foreground color of this theme. This will be used as

primaryForegroundColor	String	the text color for any important labels in a view with this theme (such as the text color for a text field that the user needs to fill out)
secondaryForegroundColor	String	The secondary foreground color of this theme. This will be used as the text color for any supplementary labels in a view with this theme (such as the placeholder color for a text field that the user needs to fill out)
accentColor	String	The accent color of this theme - it will be used for any buttons and other elements on a view that are important to highlight
errorColor	String	The error color of this theme - it will be used for rendering any error messages or view

Example

```
const options = {
  requiredBillingAddressFields: 'full',
  prefilledInformation: {
    billingAddress: {
      name: 'Gunilla Haugeh',
      line1: 'Canary Place',
      line2: '3',
      city: 'Macon',
      state: 'Georgia',
      country: 'US',
      postalCode: '31217',
    },
  },
};

const token = await stripe.paymentRequestWithCardFormAsync(options);

// Client specific code
// api.sendTokenToBackend(token)
```

Creating source [Android, iOS]

Creates source object based on params. Sources are used to create payments for a variety of [payment methods](#)

NOTE: For sources that require redirecting your customer to authorize the payment, you need to specify a return URL when you create the source. This allows your customer to be redirected back to your app after they authorize the payment. The prefix before ':' in your return URL should be the same as the scheme in your `info.plist` and `AndroidManifest.xml`. If You are not sure about this step look at above sections "Register hook in order to Stripe could process source authorization".

NOTE: If you are using Expo Client or an ejected Expo application, do not specify `returnURL`.

`params` — An object with the following keys:

Depending on the type you need to provide different params. Check the `STPSourceParams` docs for reference

Key	Type	Description
type (Required)	String	The type of the source to create. Can be one of: bancontact bitcoin card griopay ideal sepaDebit sofort threeDSecure alipay

amount	Number	A positive number in the smallest currency unit representing the amount to charge the customer (e.g., 1099 for a €10.99 payment)
name	String	The full name of the account holder
returnURL	String	The URL the customer should be redirected to after they have successfully verified the payment
statementDescriptor	String	A custom statement descriptor for the payment
currency	String	The currency associated with the source. This is the currency for which the source will be chargeable once ready
email	String	The customer's email address
bank	String	The customer's bank
iban	String	The IBAN number for the bank account you wish to debit
addressLine1	String	The bank account holder's first address line (optional)
city	String	The bank account holder's city
postalCode	String	The bank account holder's postal code
country	String	The bank account holder's two-letter country code (sepaDebit) or the country code of the customer's bank (sofort)
card	String	The ID of the card source

Example

```
const params = {
  type: 'alipay',
  amount: 5,
  currency: 'EUR',
  returnUrl: 'expaymentsstripe://stripe-redirect',
};

const source = await stripe.createSourceWithParamsAsync(params);

// Client specific code
// api.sendSourceToBackend(source)
```

ApplePay [iOS]

Remember: to use Apple Pay on a real device, you need to [set up apple pay first](#).

openApplePaySetupAsync()

Opens the user interface to set up credit cards for Apple Pay.

canMakeApplePayPaymentsAsync([options]) -> Promise

Returns whether the user can make Apple Pay payments with specified options. If there are no configured payment cards, this method always returns `false`. Return `true` if the user can make Apple Pay payments through any of the specified networks; otherwise, `false`.

NOTE: iOS Simulator always returns `true`

`options`

Key	Type	Description
networks	Array[String]	Indicates whether the user can make Apple Pay payments through the specified network. Available networks: american_express discover master_card visa . If option does not specify we pass all available networks under the hood.

Example

```
import { PaymentsStripe as Stripe } from 'expo-payments-stripe';

await Stripe.canMakeApplePayPaymentsAsync();
```

```
import { PaymentsStripe as Stripe } from 'expo-payments-stripe';

await Stripe.canMakeApplePayPaymentsAsync(['american_express', 'discover']);
```

`deviceSupportsApplePayAsync() -> Promise`

Returns whether the user can make Apple Pay payments. User may not be able to make payments for a variety of reasons. For example, this functionality may not be supported by their hardware, or it may be restricted by parental controls. Returns `true` if the device supports making payments; otherwise, `false`.

NOTE: iOS Simulator always returns `true`

`paymentRequestWithApplePayAsync(items, [options]) -> Promise`

Launch the `:Pay` view to accept payment.

`items` — An array of object with the following keys:

Key	Type	Description
label	<code>String</code>	A short, localized description of the item.
amount	<code>String</code>	The summary item's amount.
type	<code>String</code>	The summary item's type. Must be "pending" or "final". Defaults to "final".

NOTE: The final item should represent your company; it'll be prepended with the word "Pay" (i.e. "Pay Tipsi, Inc \$50")

`options` — An object with the following keys:

Key	Type	Description
requiredBillingAddressFields	Array[String]	A bit field of billing address fields that you need in order to process the transaction. Array should contain one of: all name email phone postal_address or not specify to disable
		A bit field of shipping address fields that you need in

requiredShippingAddressFields	Array[String]	order to process the transaction. Array should contain one of: all name email phone postal_address or not specify to disable
shippingMethods	Array	An array of <code>shippingMethod</code> objects that describe the supported shipping methods.
currencyCode	String	The three-letter ISO 4217 currency code. Default is USD
countryCode	String	The two-letter code for the country where the payment will be processed. Default is US
shippingType	String	An optional value that indicates how purchased items are to be shipped. Default is shipping . Available options are: shipping delivery store_pickup service_pickup

`shippingMethod` — An object with the following keys:

Key	Type	Description
id	String	A unique identifier for the shipping method, used by the app
id	String	A short, localized description of the shipping method
label	String	A unique identifier for the shipping method, used by the app
detail	String	A user-readable description of the shipping method
amount	String	The shipping method's amount

Example

```

const items = [
  {
    label: 'Whisky',
    amount: '50.00',
  },
  {
    label: 'Tipsi, Inc',
    amount: '50.00',
  },
];

const shippingMethods = [
  {
    id: 'fedex',
    label: 'FedEX',
    detail: 'Test @ 10',
    amount: '10.00',
  },
];

const options = {
  requiredBillingAddressFields: ['all'],
  requiredShippingAddressFields: ['phone', 'postal_address'],
  shippingMethods,
};

const token = await stripe.paymentRequestWithApplePayAsync(items, options);

```

Token structure — `paymentRequestWithApplePayAsync` response

`extra` — An object with the following keys

Key	Type	Description
<code>shippingMethod</code>	Object	Selected shippingMethod object
<code>billingContact</code>	Object	The user's billing contact object
<code>shippingContact</code>	Object	The user's shipping contact object

`contact` — An object with the following keys

Key	Type	Description
<code>name</code>	String	The contact's name
<code>phoneNumber</code>	String	The contact's phone number
<code>emailAddress</code>	String	The contact's email address
<code>street</code>	String	The street name in a postal address
<code>city</code>	String	The city name in a postal address
<code>state</code>	String	The state name in a postal address
<code>country</code>	String	The country name in a postal address
<code>ISOCountryCode</code>	String	The ISO country code for the country in a postal address
<code>postalCode</code>	String	The postal code in a postal address
<code>supplementarySubLocality</code>	String	The contact's sublocality

`completeApplePayRequestAsync()`/`cancelApplePayRequestAsync()` -> Promise

After `paymentRequestWithApplePayAsync` you should complete the operation by calling `completeApplePayRequestAsync` or cancel if an error occurred. This closes Apple Pay. (resolves to undefined, you do not need to store the Promise)

```
const items = [
  {
    label: 'Whisky',
    amount: '50.00',
  },
  {
    label: 'Tipsi, Inc',
    amount: '50.00',
  },
];

const shippingMethods = [
  {
    id: 'fedex',
    label: 'FedEX',
    detail: 'Test @ 10',
    amount: '10.00',
  },
];
```

```

const options = {
  requiredBillingAddressFields: 'all',
  requiredShippingAddressFields: 'all',
  shippingMethods,
};

try {
  const token = await stripe.paymentRequestWithApplePayAsync(items, options);

  // Client specific code
  // api.sendTokenToBackend(token)

  // You should complete the operation by calling
  stripe.completeApplePayRequestAsync();
} catch (error) {
  // Or cancel if an error occurred
  // stripe.cancelApplePayRequestAsync()
}

```

AndroidPay

Android Pay (also known as Google Pay) is currently only supported on ExpoKit apps. To add it to your app, add the following lines to your `AndroidManifest.xml` file, inside of the `<application>....</application>` tags:

```

<meta-data
  android:name="com.google.android.gms.wallet.api.enabled"
  android:value="true" />

```

`deviceSupportsAndroidPayAsync() -> Promise`

Indicates whether or not the device supports AndroidPay. Returns a `Boolean` value.

```

import { PaymentsStripe as Stripe } from 'expo-payments-stripe';

await Stripe.deviceSupportsAndroidPayAsync();

```

`canMakeAndroidPayPaymentsAsync() -> Promise`

Indicates whether or not the device supports AndroidPay and user has existing payment method. Returns a `Boolean` value.

```

import { PaymentsStripe as Stripe } from 'expo-payments-stripe';

await Stripe.canMakeAndroidPayPaymentsAsync();

```

`paymentRequestWithAndroidPayAsync(options) -> Promise`

options — An object with the following keys:

Key	Type	Description
<code>total_price</code>	String	Total price for items
<code>currency_code</code>	String	Three-letter ISO currency code representing the currency paid out to the bank account

shipping_address_required (Optional)	Bool	Is shipping address menu required? Default is false
billing_address_required (Optional)	Bool	Is billing address menu required? Default is false
line_items	Array	Array of purchased items. Each item contains line_item

line_item — An object with the following keys:

Key	Type	Description
currency_code	String	Currency code string
description	String	Short description that will shown to user
total_price	String	Total order price
unit_price	String	Price per unit
quantity	String	Number of items

Example

```
const options = {
  total_price: '80.00',
  currency_code: 'USD',
  shipping_address_required: false,
  billing_address_required: true,
  shipping_countries: ['US', 'CA'],
  line_items: [
    {
      currency_code: 'USD',
      description: 'Whisky',
      total_price: '50.00',
      unit_price: '50.00',
      quantity: '1',
    },
    {
      currency_code: 'USD',
      description: 'Vine',
      total_price: '30.00',
      unit_price: '30.00',
      quantity: '1',
    },
  ],
};

const token = await stripe.paymentRequestWithAndroidPayAsync(options);

// Client specific code
// api.sendTokenToBackend(token)
```

Example of token:

```
{ card:
  { currency: null,
    fingerprint: null,
    funding: "credit",
    brand: "MasterCard",
    number: null,
    addressState: null,
    country: "US",
```

```

    cvc: null,
    expMonth: 12,
    addressLine1: null,
    expYear: 2022,
    addressCountry: null,
    name: null,
    last4: "4448",
    addressLine2: null,
    addressCity: null,
    addressZip: null
  },
  created: 1512322244000,
  used: false,
  extra: {
    email: "randomemail@mail.com",
    billingContact: {
      postalCode: "220019",
      name: "John Doe",
      locality: "NY",
      countryCode: "US",
      administrativeArea: "US",
      address1: "Time square 1/11"
    },
    shippingContact: {}
  },
  livemode: false,
  tokenId: "tok_1BV1IeDZwq0ES60ZphBXBoDr"
}

```

Structures of the objects

The Token object

A `token` object returned from submitting payment details to the Stripe API via:

- `paymentRequestWithApplePayAsync`
- `paymentRequestWithCardFormAsync`
- `createTokenWithCardAsync`

`token` — an object with the following keys

Key	Type	Description
tokenId	String	The value of the token. You can store this value on your server and use it to make charges and customers
created	Number	When the token was created
livemode	Number	Whether or not this token was created in livemode. Will be 1 if you used your Live Publishable Key, and 0 if you used your Test Publishable Key
card	Object	The credit card details object that were used to create the token
bankAccount	Object	The external (bank) account details object that were used to create the token
extra	Object	An additional information that method can provide

`card` — an object with the following keys

Key	Type	Description
cardId	String	The Stripe ID for the card
brand	String	The card's brand. Can be one of: JCB American Express Visa Discover Diners Club MasterCard Unknown
funding (iOS)	String	The card's funding. Can be one of: debit credit prepaid unknown
last4	String	The last 4 digits of the card
dynamicLast4 (iOS)	String	For cards made with Apple Pay, this refers to the last 4 digits of the Device Account Number for the tokenized card
isApplePayCard (iOS)	Bool	Whether or not the card originated from Apple Pay
expMonth	Number	The card's expiration month. 1-indexed (i.e. 1 == January)
expYear	Number	The card's expiration year
country	String	Two-letter ISO code representing the issuing country of the card
currency	String	This is only applicable when tokenizing debit cards to issue payouts to managed accounts. The card can then be used as a transfer destination for funds in this currency
name	String	The cardholder's name
addressLine1	String	The cardholder's first address line
addressLine2	String	The cardholder's second address line
addressCity	String	The cardholder's city
addressState	String	The cardholder's state
addressCountry	String	The cardholder's country
addressZip	String	The cardholder's zip code

bankAccount

Key	Type	Description
routingNumber	String	The routing number of this account
accountNumber	String	The account number for this BankAccount.
countryCode	String	The two-letter country code that this account was created in
currency	String	The currency of this account
accountHolderName	String	The account holder's name
accountHolderType	String	the bank account type. Can be one of: company individual
fingerprint	String	The account fingerprint
bankName	String	The name of bank
last4	String	The last four digits of the account number

Example

```
{
  tokenId: 'tok_19GCAQI5NuVQgnjeKNE32K0p',
  created: 1479236426,
  livemode: 0,
  card: {
    cardId: 'card_19GCAQI5NuVQgnjeRZizG4U3',
    brand: 'Visa',
    funding: 'credit',
    last4: '4242',
    expMonth: 4,
    expYear: 2024,
    country: 'US',
    name: 'Eugene Grissom',
    addressLine1: 'Green Street',
    addressLine2: '3380',
    addressCity: 'Nashville',
    addressState: 'Tennessee',
    addressCountry: 'US',
    addressZip: '37211',
  },
  bankAccount: {
    bankName: 'STRIPE TEST BANK',
    accountHolderType: 'company',
    last4: '6789',
    accountHolderName: 'Test holder name',
    currency: 'usd',
    fingerprint: 'afghsajhaartkjasd',
    countryCode: 'US',
    accountNumber: '424542424',
    routingNumber: '110000000',
  },
}
}
```

The Source object

A source object returned from creating a source (via `createSourceWithParamsAsync`) with the Stripe API.

`source` — an object with the following keys:

Key	Type	Description
amount	Number	The amount associated with the source
clientSecret	String	The client secret of the source. Used for client-side polling using a publishable key
created	Number	When the source was created
currency	String	The currency associated with the source
flow	String	The authentication flow of the source. Can be one of: none redirect verification receiver unknown
livemode	Bool	Whether or not this source was created in <i>livemode</i> . Will be <i>true</i> if you used your Live Publishable Key, and <i>false</i> if you used your Test Publishable Key
metadata	Object	A set of key/value pairs associated with the source object
owner	Object	Information about the owner of the payment instrument
receiver	Object (Optional)	Information related to the receiver flow. Present if the source is a receiver

redirect	Object (Optional)	Information related to the redirect flow. Present if the source is authenticated by a redirect
status	String	The status of the source. Can be one of: pending chargeable consumed cancelled failed
type	String	The type of the source. Can be one of: bancontact card giropay ideal sepaDebit sofort threeDSecure alipay unknown
usage	String	Whether this source should be reusable or not. Can be one of: reusable single unknown
verification	Object (Optional)	Information related to the verification flow. Present if the source is authenticated by a verification
details	Object	Information about the source specific to its type
cardDetails	Object (Optional)	If this is a card source, this property contains information about the card
sepaDebitDetails	Object (Optional)	If this is a SEPA Debit source, this property contains information about the sepaDebit

owner

Key	Type	Description
address	Object (Optional)	Owner's address
email	String (Optional)	Owner's email address
name	String (Optional)	Owner's full name
phone	String (Optional)	Owner's phone number
verifiedAddress	Object (Optional)	Verified owner's address
verifiedEmail	String (Optional)	Verified owner's email address
verifiedName	String (Optional)	Verified owner's full name
verifiedPhone	String (Optional)	Verified owner's phone number

receiver

Key	Type	Description
address	Object	The address of the receiver source. This is the value that should be communicated to the customer to send their funds to
amountCharged	Number	The total amount charged by you
amountReceived	Number	The total amount received by the receiver source
amountReturned	Number	The total amount that was returned to the customer

redirect

Key	Type	Description
returnURL	String	The URL you provide to redirect the customer to after they authenticated their payment
status	String	The status of the redirect. Can be one of: pending succeeded failed unknown

url	String	The URL provided to you to redirect a customer to as part of a redirect authentication flow
-----	--------	---

verification

Key	Type	Description
attemptsRemaining	Number	The number of attempts remaining to authenticate the source object with a verification code
status	String	The status of the verification. Can be one of: pending succeeded failed unknown

cardDetails

Key	Type	Description
last4	String	The last 4 digits of the card
expMonth	Number	The card's expiration month. 1-indexed (i.e. 1 == January)
expYear	Number	The card's expiration year
brand	String	The issuer of the card. Can be one of: JCB American Express Visa Discover Diners Club MasterCard Unknown
funding (iOS)	String	The funding source for the card. Can be one of: debit credit prepaid unknown
country	String	Two-letter ISO code representing the issuing country of the card
threeDSecure	String	Whether 3D Secure is supported or required by the card. Can be one of: required optional notSupported unknown

sepaDebitDetails

Key	Type	Description
last4	String	The last 4 digits of the account number
bankCode	String	The account's bank code
country	String	Two-letter ISO code representing the country of the bank account
fingerprint	String	The account's fingerprint
mandateReference	String	The reference of the mandate accepted by your customer
mandateURL	String	The details of the mandate accepted by your customer

Example

```
{
  livemode: false,
  amount: 50,
  owner: {},
  metadata: {},
  clientSecret: 'src_client_secret_BLnXIZxZprDmdhw3zv12123L',
  details: {
    native_url: null,
    statement_descriptor: null
}
```

```
        },
        type: 'alipay',
        redirect: {
          url: 'https://hooks.stripe.com/redirect/authenticate/src_1Az5vzE5aJKqY779Kes5s61m?client_secret=src_client_secret_BLnXIZxZprDmdhw3zv12123L',
          returnUrl: 'example://stripe-redirect?redirect_merchant_name=example',
          status: 'succeeded'
        },
        usage: 'single',
        created: 1504713563,
        flow: 'redirect',
        currency: 'euro',
        status: 'chargeable',
      }
    }
```

Enabling Apple Pay in ExpoKit

If you want to use Apple Pay for payments, you'll need to set up your merchant ID in XCode first. Note that you do not need to go through this process to use the Payments module - you can still process payments with Stripe without going through the steps in this section.

If you haven't already, set up an Apple Merchant ID via the [Apple Developer Portal](#). Then, open the application in XCode and navigate to the capabilities tab. Enable Apple Pay and insert your merchant ID into the corresponding space.

Note: Apple Pay can be used only for real world items (ex. appeal, car sharing, food) and not virtual goods. For more information about proper usage of Apple Pay, visit Apple's [Apple Pay guidelines](#) and [Acceptable Use](#).

Pedometer

Use Core Motion (iOS) or Google Fit (Android) to get the user's step count.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

API

```
// in managed apps:  
import { Pedometer } from 'expo';  
  
// in bare apps:  
import { Pedometer } from 'expo-sensors';
```

Pedometer.isAvailableAsync()

Returns whether the pedometer is enabled on the device.

Returns

- Returns a promise that resolves to a `Boolean`, indicating whether the pedometer is available on this device.

Pedometer.getStepCountAsync(start, end)

Get the step count between two dates.

Arguments

- **start (Date)** -- A date indicating the start of the range over which to measure steps.
- **end (Date)** -- A date indicating the end of the range over which to measure steps.

Returns

- Returns a promise that resolves to an `object` with a `steps` key, which is a `Number` indicating the number of steps taken between the given dates.

Pedometer.watchStepCount(callback)

Subscribe to pedometer updates.

Arguments

- **callback (*function*)** A callback that is invoked when new step count data is available. The callback is provided a single argument that is an object with a `steps` key.

Returns

- An EventSubscription object that you can call `remove()` on when you would like to unsubscribe the listener.

Standalone Applications

You'll need to configure an Android OAuth client for your app on the Google Play console for it to work as a standalone application on the Android platform. See <https://developers.google.com/fit/android/get-api-key>

Permissions

When it comes to adding functionality that can access potentially sensitive information on a user's device, such as their location, or possibly send them possibly unwanted push notifications, you will need to ask the user for their permission first. Unless you've already asked their permission, then no need. And so we have the `Permissions` module.

If you are deploying your app to the Apple iTunes Store, you should consider adding additional metadata to your app in order to customize the system permissions dialog and explain why your app requires permissions. See more info in the [App Store Deployment Guide](#).

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

Manually testing permissions

Often you want to be able to test what happens when you reject a permission to ensure that it has the desired behavior. An operating-system level restriction on both iOS and Android prohibits an app from asking for the same permission more than once (you can imagine how this could be annoying for the user to be repeatedly prompted for permissions). So in order to test different flows involving permissions, you may need to uninstall and reinstall the Expo app. In the simulator this is as easy as deleting the app and expo-cli will automatically install it again next time you launch the project from it.

API

```
// in managed apps:  
import { Permissions } from 'expo';  
  
// in bare apps:  
import * as Permissions from 'expo-permissions';
```

Permissions.getAsync(...permissionTypes)

Determines whether your app has already been granted access to the provided permissions types.

Arguments

- **permissionTypes (string)** -- The names of the permissions types.

Returns

Returns a `Promise` that is resolved with the information about the permissions, including status, expiration and scope (if it applies to the permission type). Top-level `status` and `expires` keys stores combined info of each component permission that is asked for. If any permission resulted in a negative result, then that negative result is propagated here; that means top-level values are positive only if all component values are positive.

Examples `[...componentsValues] => topLevelStatus :`

- `[granted, denied, granted] => denied`
- `[granted, granted, granted] => granted`

```
{  
  status, // combined status of all component permissions being asked for, if any of has status !== 'granted' then th  
  at status is propagated here  
  expires, // combined expires of all permissions being asked for, same as status  
  permissions: { // an object with an entry for each permission requested  
    [Permissions.TYPE]: {  
      status,  
      expires,  
      ... // any additional permission-specific fields  
    },  
    ...  
  },  
}
```

Example

```
async function alertIfRemoteNotificationsDisabledAsync() {  
  const { Permissions } = Expo;  
  const { status } = await Permissions.getAsync(Permissions.NOTIFICATIONS);  
  if (status !== 'granted') {  
    alert('Hey! You might want to enable notifications for my app, they are good.');  
  }  
  
  async function checkMultiPermissions() {  
    const { Permissions } = Expo;  
    const { status, expires, permissions } = await Permissions.getAsync(Permissions.CALENDAR, Permissions.CONTACTS)  
    if (status !== 'granted') {  
      alert('Hey! You have not enabled selected permissions');  
    }  
  }  
}
```

Permissions.askAsync(...types)

Prompt the user for types of permissions. If they have already granted access, response will be success.

Arguments

- `types (string)` -- The names of the permissions types.

Returns

Same as for `Permissions.getAsync`

Example

```

async function getLocationAsync() {
  const { Location, Permissions } = Expo;
  // permissions returns only for location permissions on iOS and under certain conditions, see Permissions.LOCATION
  const { status, permissions } = await Permissions.askAsync(Permissions.LOCATION);
  if (status === 'granted') {
    return Location.getCurrentPositionAsync({enableHighAccuracy: true});
  } else {
    throw new Error('Location permission not granted');
  }
}

```

Permissions types

Permissions.NOTIFICATIONS

The permission type for user-facing notifications **and** remote push notifications.

Note: On iOS, asking for this permission asks the user not only for permission to register for push/remote notifications, but also for showing notifications as such. At the moment remote notifications will only be received when notifications are permitted to play a sound, change the app badge or be displayed as an alert. As iOS is more detailed when it comes to notifications permissions, this permission status will contain not only `status` and `expires`, but also Boolean values for `allowsSound`, `allowsAlert` and `allowsBadge`.

Note: On iOS, this does not disambiguate `undetermined` from `denied` and so will only ever return `granted` or `undetermined`. This is due to the way the underlying native API is implemented.

Note: Android does not differentiate between permissions for local and remote notifications, so status of permission for `NOTIFICATIONS` should always be the same as the status for `USER_FACING_NOTIFICATIONS`.

Permissions.USER_FACING_NOTIFICATIONS

The permission type for user-facing notifications. This does **not** register your app to receive remote push notifications; see the `NOTIFICATIONS` permission.

Note: iOS provides more detailed permissions, so the permission status will contain not only `status` and `expires`, but also Boolean values for `allowsSound`, `allowsAlert` and `allowsBadge`.

Note: Android does not differentiate between permissions for local and remote notifications, so status of permission for `USER_FACING_NOTIFICATIONS` should always be the same as the status for `NOTIFICATIONS`.

Permissions.LOCATION

The permission type for location access.

Note: iOS is not working with this permission being not individually,

`Permissions.askAsync(Permissions.SOME_PERMISSIONS, Permissions.LOCATION, Permissions.CAMERA, ...)` would throw.

On iOS ask for this permission type individually.

Note (iOS): In Expo Client this permission will always ask the user for permission to access location data while the app is in use.

Note (iOS): iOS provides more detailed permissions, returning `{ status, permissions: { location: { ios } } }` where `ios` which is an object containing: `{ scope: 'whenInUse' | 'always' | 'none' }`. If you would like to access location data in a standalone app, note that you'll need to provide location usage descriptions in

`app.json`. For more information see [Deploying to App Stores guide](#).

What location usage descriptions should I provide? Due to the design of the location permission API on iOS we aren't able to provide you with methods for asking for `whenInUse` OR `always` location usage permission specifically. However, you can customize the behavior by providing the following sets of usage descriptions:

- if you provide only `NSLocationWhenInUseUsageDescription`, your application will only ever ask for location access permission "when in use",
- if you provide both `NSLocationWhenInUseUsageDescription` and `NSLocationAlwaysAndWhenInUseUsageDescription`, your application will only ask for "when in use" permission on iOS 10, whereas on iOS 11+ it will show a dialog to the user where he'll be able to pick whether he'd like to give your app permission to access location always or only when the app is in use,
- if you provide all three: `NSLocationWhenInUseUsageDescription`, `NSLocationAlwaysAndWhenInUseUsageDescription` and `NSLocationAlwaysUsageDescription`, your application on iOS 11+ will still show a dialog described above and on iOS 10 it will only ask for "always" location permission.

Permissions.CAMERA

The permission type for photo and video taking.

Permissions.AUDIO_RECORDING

The permission type for audio recording.

Permissions.CONTACTS

The permission type for reading contacts.

Permissions.CAMERA_ROLL

The permission type for reading or writing to the camera roll.

Permissions.CALENDAR

The permission type for reading or writing to the calendar.

Permissions.REMINDERS

The permission type for reading or writing reminders. (iOS only, on Android would return `granted` immediately)

Permissions.SYSTEM_BRIGHTNESS

The permissions type for changing brightness of the screen

Android: permissions equivalents inside `app.json`

In order to request permissions in a standalone Android app, you need to specify the corresponding native permission types in the `android.permissions` key inside `app.json` ([read more about configuration](#)). The mapping between `Permissions` values and native permission types is as follows:

Expo	Android
LOCATION	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
CAMERA	CAMERA
AUDIO_RECORDING	RECORD_AUDIO
CONTACTS	READ_CONTACTS
CAMERA_ROLL	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE
CALENDAR	READ_CALENDAR, WRITE_CALENDAR

For example, if your app asks for `AUDIO_RECORDING` permission at runtime but no other permissions, you should set `android.permissions` to `["RECORD_AUDIO"]` in `app.json`.

Note: If you don't specify `android.permissions` inside your `app.json`, by default your standalone Android app will require all of the permissions listed above.

Print

An API for iOS (AirPrint) and Android printing functionality.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Print } from 'expo';  
  
// in bare apps:  
import { Print } from 'expo-print';
```

Print.printAsync(options)

Prints a document or HTML.

Arguments

- **options (object)** -- A map defining what should be printed:
 - **uri (string)** -- URI of a PDF file to print. Remote, local (ex. selected via `DocumentPicker`) or base64 data URI starting with `data:application/pdf;base64,`. This only supports PDF, not other types of document (e.g. images).
 - **html (string)** -- HTML string to print.
 - **width (number)** -- Width of the single page in pixels. Defaults to `612` which is a width of US Letter paper format with 72 PPI. **Available only with `html` option.**
 - **height (number)** -- Height of the single page in pixels. Defaults to `792` which is a height of US Letter paper format with 72 PPI. **Available only with `html` option.**
 - **markupFormatterIOS (string)** -- **Available on iOS only.** Alternative to `html` option that uses `UIMarkupTextPrintFormatter` instead of `WebView`. Might be removed in the future releases.
 - **printerUrl (string)** -- **Available on iOS only.** URL of the printer to use. Returned from `selectPrinterAsync`.
 - **orientation (string)** -- **Available on iOS only.** The orientation of the printed content, `Print.Orientation.portrait` OR `Print.Orientation.landscape`.

Returns

- Resolves to an empty promise if printing started.

Print.printToFileAsync(options)

Prints HTML to PDF file and saves it to [app's cache directory](#).

Arguments

- **options (object)** -- A map of options:
 - **html (string)** -- HTML string to print into PDF file.
 - **width (number)** -- Width of the single page in pixels. Defaults to `612` which is a width of US Letter paper format with 72 PPI.
 - **height (number)** -- Height of the single page in pixels. Defaults to `792` which is a height of US Letter paper format with 72 PPI.
 - **base64 (boolean)** -- Whether to include base64 encoded string of the file in the returned object.

Returns

- Resolves to an object with following keys:
 - **uri (string)** -- A URI to the printed PDF file.
 - **numberOfPages (number)** -- Number of pages that were needed to render given content.
 - **base64 (string)** -- Base64 encoded string containing the data of the PDF file. **Available only if `base64` option is truthy.** It doesn't include data URI prefix `data:application/pdf;base64,`.

Print.selectPrinterAsync()

Available on iOS only. Chooses a printer that can be later used in `printAsync`.

Returns

- Resolves to an object containing `name` and `url` of the selected printer.

Page margins

If you're using `html` option in `printAsync` or `printToFileAsync`, the resulting print might contain page margins (it depends on WebView engine). They are set by `@page` style block and you can override them in your HTML code:

```
<style>
  @page {
    margin: 20px;
  }
</style>
```

See [@page docs on MDN](#) for more details.

registerRootComponent

This function tells Expo what component to use as the root component for your app.

Installation

This API is pre-installed in [managed](#) apps. It is not available for [bare](#) React Native apps.

API

```
import { registerRootComponent } from 'expo';
```

registerRootComponent(component)

Sets the main component for Expo to use for your app.

Note: Prior to SDK 18, it was necessary to use `registerRootComponent` directly, but for projects created as of SDK 18 or later, this is handled automatically in the Expo SDK.

Arguments

- **component (ReactComponent)** -- The React component class that renders the rest of your app.

Returns

No return value.

Note: `registerRootComponent` is roughly equivalent to React Native's [AppRegistry.registerComponent](#), with some additional hooks to provide Expo specific functionality.

Common questions

I created my project before SDK 18 and I want to remove registerRootComponent , how do I do this?

- Before continuing, make sure your project is running on SDK 18 or later.
- Open up `main.js` (or if you changed it, whatever your `"main"` is in `package.json`).
- Set `"main"` to `"node_modules/expo/AppEntry.js"`.
- Delete the `registerRootComponent` call from `main.js` and put `export default` before your root component's class declaration.
- Rename `main.js` to `App.js`.

What if I want to name my main app file something other than App.js?

You can set the `"main"` in `package.json` to any file within your project. If you do this, then you need to use `registerRootComponent`; `export default` will not make this component the root for the Expo app if you are using a custom entry file.

For example, let's say you want to make `"src/main.js"` the entry file for your app -- maybe you don't like having JavaScript files in the project root, for example. First, set this in `package.json`:

```
{  
  "main": "src/main.js"  
}
```

Then in `"src/main.js"`, make sure you call `registerRootComponent` and pass in the component you want to render at the root of the app.

```
import { registerRootComponent } from 'expo';  
import React from 'react';  
import { View } from 'react-native';  
  
class App extends React.Component {  
  render() {  
    return <View />;  
  }  
}  
  
registerRootComponent(App);
```

ScreenOrientation

Allows changing supported screen orientations at runtime. This will take priority over the `orientation` key in `app.json`.

Installation

This API is pre-installed in [managed](#) apps. It is not yet available for [bare](#) React Native apps.

API

```
import { ScreenOrientation } from 'expo';
```

ScreenOrientation.allow(orientation)

Deprecated in favour of `ScreenOrientation.allowAsync`.

ScreenOrientation.allowAsync(orientation)

Allow a screen orientation. You can call this function multiple times with multiple orientations to allow multiple orientations.

Arguments

- **orientation (string)** -- The allowed orientation. See the `Orientation` enum for possible values.

Returns

Returns a `Promise` with `null` value.

Example

```
function changeScreenOrientation() {
  ScreenOrientation.allowAsync(ScreenOrientation.Orientation.LANDSCAPE);
}
```

Orientation types

ScreenOrientation.Orientation

An object containing the values that can be passed to the `allow` function.

- `ALL` -- All 4 possible orientations
- `ALL_BUT_UPSIDE_DOWN` -- All but reverse portrait, could be all 4 orientations on certain Android devices.
- `PORTRAIT` -- Portrait orientation, could also be reverse portrait on certain Android devices.

- `PORTRAIT_UP` -- Upside portrait only.
- `PORTRAIT_DOWN` -- Upside down portrait only.
- `LANDSCAPE` -- Any landscape orientation.
- `LANDSCAPE_LEFT` -- Left landscape only.
- `LANDSCAPE_RIGHT` -- Right landscape only.

Detecting when the orientation changes

The best way to do this is to listen for changes to [Dimensions](#).

SecureStore

Provides a way to encrypt and securely store key–value pairs locally on the device. Each Expo project has a separate storage system and has no access to the storage of other Expo projects.

iOS: Values are stored using the [keychain services](#) as `kSecClassGenericPassword`. iOS has the additional option of being able to set the value's `kSecAttrAccessible` attribute, which controls when the value is available to be fetched.

Android: Values are stored in `SharedPreferences`, encrypted with [Android's Keystore system](#).

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { SecureStore } from 'expo';  
  
// in bare apps:  
import * as SecureStore from 'expo-secure-store';
```

SecureStore.setItemAsync(key, value, options)

Store a key–value pair.

Arguments

- **key (string)** -- The key to associate with the stored value. Keys may contain alphanumeric characters `.`, `-`, and `_`.
- **value (string)** -- The value to store.
- **options (object) (optional)** -- A map of options:
 - **keychainService (string)**--
 - iOS: The item's service, equivalent to `kSecAttrService`
 - Android: Equivalent of the public/private key pair `Alias`
 - **keychainAccessible (enum)**--
 - iOS only: Specifies when the stored entry is accessible, using iOS's `kSecAttrAccessible` property. See Apple's documentation on [keychain item accessibility](#). The available options are:
 - `SecureStore.WHEN_UNLOCKED` : The data in the keychain item can be accessed only while the device is unlocked by the user.
 - `SecureStore.AFTER_FIRST_UNLOCK` : The data in the keychain item cannot be accessed after a restart until the device has been unlocked once by the user. This may be useful if you need to access

- the item when the phone is locked.
- `SecureStore.ALWAYS` : The data in the keychain item can always be accessed regardless of whether the device is locked. This is the least secure option.
- `SecureStore.WHEN_UNLOCKED_THIS_DEVICE_ONLY` : Similar to `WHEN_UNLOCKED`, except the entry is not migrated to a new device when restoring from a backup.
- `SecureStore.WHEN_PASSCODE_SET_THIS_DEVICE_ONLY` : Similar to `WHEN_UNLOCKED_THIS_DEVICE_ONLY`, except the user must have set a passcode in order to store an entry. If the user removes their passcode, the entry will be deleted.
- `SecureStore.AFTER_FIRST_UNLOCK_THIS_DEVICE_ONLY` : Similar to `AFTER_FIRST_UNLOCK`, except the entry is not migrated to a new device when restoring from a backup.
- `SecureStore.ALWAYS_THIS_DEVICE_ONLY` : Similar to `ALWAYS`, except the entry is not migrated to a new device when restoring from a backup.

Returns

A promise that will reject if value cannot be stored on the device.

`SecureStore.getItemAsync(key, options)`

Fetch the stored value associated with the provided key.

Arguments

- **key (string)** -- The key that was used to store the associated value.
- **options (object)** (optional) -- A map of options:
 - **keychainService (string)** -- iOS: The item's service, equivalent to `kSecAttrService`. Android: Equivalent of the public/private key pair `Alias`.

NOTE If the item is set with the `keychainService` option, it will be required to later fetch the value.

Returns

A promise that resolves to the previously stored value, or null if there is no entry for the given key. The promise will reject if an error occurred while retrieving the value.

`SecureStore.deleteItemAsync(key, options)`

Delete the value associated with the provided key.

Arguments

- **key (string)** -- The key that was used to store the associated value.
- **options (object)** (optional) -- A map of options:
 - **keychainService (string)** -- iOS: The item's service, equivalent to `kSecAttrService`. Android: Equivalent of the public/private key pair `Alias`. If the item is set with a keychainService, it will be required to later fetch the value.

Returns

A promise that will reject if the value couldn't be deleted.

Segment

Provides access to <https://segment.com/> mobile analytics. Wraps Segment's iOS and Android sources.

Note: Session tracking may not work correctly when running Experiences in the main Expo app. It will work correctly if you create a standalone app.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Segment } from 'expo';  
  
// in bare apps:  
import * as Segment from 'expo-analytics-segment';
```

Segment.initialize({ androidWriteKey, iosWriteKey })

Segment requires separate write keys for iOS and Android. You will need to log in to Segment to receive these <https://segment.com/docs/guides/setup/how-do-i-find-my-write-key/>

Arguments

Accepts an object with the following keys:

- **androidWriteKey (string)** – Write key for Android source.
- **iosWriteKey (string)** – Write key for iOS source.

Segment.identify(userId)

Associates the current user with a user ID. Call this after calling `Segment.initialize()` but before other segment calls. See <https://segment.com/docs/spec/identify/>.

Arguments

- **userId (string)** – User ID for the current user.

Segment.identifyWithTraits(userId, traits)

Arguments

- **userId (string)** – User ID for the current user.
- **traits (object)** – A map of custom properties.

Segment.reset()

Clears the current user. See <https://segment.com/docs/sources/mobile/ios/#reset>.

Segment.track(event)

Log an event to Segment. See <https://segment.com/docs/spec/track/>.

Arguments

- **event (string)** – The event name.

Segment.trackWithProperties(event, properties)

Log an event to Segment with custom properties. See <https://segment.com/docs/spec/track/>.

Arguments

- **event (string)** – The event name.
- **properties (object)** – A map of custom properties.

Segment.group(groupId)

Associate the user with a group. See <https://segment.com/docs/spec/group/>.

Arguments

- **groupId (string)** – ID of the group.

Segment.groupWithTraits(groupId, traits)

Associate the user with a group with traits. See <https://segment.com/docs/spec/group/>.

Arguments

- **groupId (string)** – ID of the group.
- **traits (object)** – free-form dictionary of traits of the group.

Segment.alias(newId, [options])

Associate current identity with a new identifier. See <https://segment.com/docs/spec/alias/>.

Arguments

- **newId (string)** – Identifier to associate with.
- **options (object) – (optional)** extra dictionary with options for the call. You could pass a dictionary of form `{ [integrationKey]: { enabled: boolean, options: object } }` to configure destinations of the call.

Returns

- A `Promise` resolving to a `boolean` indicating whether the method has been executed on the underlying Segment instance or not.

Segment.screen(screenName)

Record that a user has seen a screen to Segment. See <https://segment.com/docs/spec/screen/>.

Arguments

- `screenName (string)` – Name of the screen.

Segment.screenWithProperties(screenName, properties)

Record that a user has seen a screen to Segment with custom properties. See <https://segment.com/docs/spec/screen/>.

- `screenName (string)` – Name of the screen.
- `properties (object)` – A map of custom properties.

Segment.flush()

Manually flush the event queue. You shouldn't need to call this in most cases.

Opting out (enabling/disabling all tracking)

Depending on the audience for your app (e.g. children) or the countries where you sell your app (e.g. the EU), you may need to offer the ability for users to opt-out of analytics data collection inside your app. You can turn off forwarding to ALL destinations including Segment itself: ([Source – Segment docs](#))

```
import { Segment } from 'expo';

Segment.setEnabledAsync(false);

// Or if they opt-back-in, you can re-enable data collection:
Segment.setEnabledAsync(true);
```

Note: disabling the Segment SDK ensures that all data collection method invocations (eg. `track`, `identify`, etc) are ignored.

This method is only supported in standalone and detached apps. In Expo Client the promise will reject.

The setting value will be persisted across restarts, so once you call `setEnabledAsync(false)`, Segment won't track the users even when the app restarts. To check whether tracking is enabled, use `Segment.getEnabledAsync()` which returns a promise which should resolve to a boolean.

Sensors

Various APIs for accessing device sensors are included under this umbrella package called `expo-sensors`.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import {  
  Accelerometer,  
  Barometer,  
  Gyroscope,  
  Magnetometer,  
  MagnetometerUncalibrated,  
  Pedometer,  
} from 'expo';  
  
// in bare apps:  
import {  
  Accelerometer,  
  Barometer,  
  Gyroscope,  
  Magnetometer,  
  MagnetometerUncalibrated,  
  Pedometer,  
} from 'expo-sensors';
```

See documentation for the sensors you are interested in using for more information.

SMS

Provides access to the system's UI/app for sending SMS messages.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { SMS } from 'expo';  
  
// in bare apps:  
import * as SMS from 'expo-sms';
```

SMS.isAvailableAsync()

Determines whether SMS is available.

Returns

Returns a promise that resolves to a `Boolean`, indicating whether SMS is available on this device.

Example

```
const isAvailable = await SMS.isAvailableAsync();  
if (isAvailable) {  
  // do your SMS stuff here  
} else {  
  // misfortune... there's no SMS available on this device  
}
```

SMS.sendSMSAsync(addresses, message)

Opens the default UI/app for sending SMS messages with prefilled addresses and message.

Arguments

- **addresses (`Array|String`)** -- An array of addresses (*phone numbers*) or single address passed as strings.
Those would appear as recipients of the prepared message.
- **message (`String`)** -- Message to be sent

Returns

Returns a `Promise` that resolves when the SMS action is invoked by the user, with corresponding result:

- If the user cancelled the SMS sending process: `{ result: 'cancelled' }` .
- If the user has sent/scheduled message for sending: `{ result: 'sent' }` .
- If the status of the SMS message cannot be determined: `{ result: 'unknown' }` .

Android does not provide information about the status of the SMS message, so on Android devices the `Promise` will always resolve with `{ result: 'unknown' }` .

Note: The only feedback collected by this module is whether any message has been sent. That means we do not check actual content of message nor recipients list.

Example

```
const { result } = await SMS.sendSMSAsync(['0123456789', '9876543210'], 'My sample HelloWorld message');
```

Speech

This module allows using Text-to-speech utility.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { Speech } from 'expo';  
  
// in bare apps:  
import * as Speech from 'expo-speech';
```

Speech.speak(text, options)

Speak out loud the `text` given `options`. Calling this when another text is being spoken adds an utterance to queue.

Arguments

- **text (string)** -- The text to be spoken.
- **options (object)** --

A map of options:

- **language (string)** -- The code of a language that should be used to read the `text`, check out IETF BCP 47 to see valid codes.
- **pitch (number)** -- Pitch of the voice to speak `text`. 1.0 is the normal pitch.
- **rate (number)** -- Rate of the voice to speak `text`. 1.0 is the normal rate.
- **onStart (function)** -- A callback that is invoked when speaking starts.
- **onDone (function)** -- A callback that is invoked when speaking finishes.
- **onStopped (function)** -- A callback that is invoked when speaking is stopped by calling `Speech.stop()`.
- **onError (function)** -- (Android only). A callback that is invoked when an error occurred while speaking.

Speech.stop()

Interrupts current speech and deletes all in queue.

Speech.pause()

Pauses current speech.

Speech.resume()

Resumes speaking previously paused speech or does nothing if there's none.

Speech.isSpeakingAsync()

Determine whether the Text-to-speech utility is currently speaking. Will return `true` if speaker is paused.

Returns

Returns a Promise that resolves to a boolean, `true` if speaking, `false` if not.

SplashScreen

A module that tells Expo to keep the splash screen visible until you make it hide.

This is useful to let you create an impression of a pure React component splash screen. You can combine it with [AppLoading](#). Read more about [creating a splash screen](#).

Installation

This API is pre-installed in [managed](#) apps. It is not available for [bare](#) React Native apps.

API

```
import { SplashScreen } from 'expo';
```

SplashScreen.preventAutoHide()

Makes the native splash screen (configured in `app.json`) stay visible until `hide` is called.

SplashScreen.hide()

Hides the native splash screen.

Example with AppLoading

```
import React from 'react';
import { Image, Text, View } from 'react-native';
import { Asset, AppLoading, SplashScreen } from 'expo';

export default class App extends React.Component {
  state = {
    isSplashReady: false,
    isAppReady: false,
  };

  render() {
    if (!this.state.isSplashReady) {
      return (
        <AppLoading
          startAsync={this._cacheSplashResourcesAsync}
          onFinish={() => this.setState({ isSplashReady: true })}
          onError={console.warn}
          autoHideSplash={false}
        />
      );
    }

    if (!this.state.isAppReady) {
      return (
        <View style={{ flex: 1 }}>
          <Image
            source={require('./assets/logo.png')}
          />
        </View>
      );
    }

    return (
      <View style={{ flex: 1 }}>
        <Text>Welcome to your new app!</Text>
      </View>
    );
  }
}
```

```

        source={require('./assets/images/splash.gif')}
        onLoad={this._cacheResourcesAsync}
      />
    </View>
  );
}

return (
  <View style={{ flex: 1 }}>
    <Image source={require('./assets/images/expo-icon.png')} />
    <Image source={require('./assets/images/slack-icon.png')} />
  </View>
);
}

_cacheSplashResourcesAsync = async () => {
  const gif = require('./assets/images/splash.gif');
  return Asset.fromModule(gif).downloadAsync()
}

_cacheResourcesAsync = async () => {
  SplashScreen.hide();
  const images = [
    require('./assets/images/expo-icon.png'),
    require('./assets/images/slack-icon.png'),
  ];

  const cacheImages = images.map((image) => {
    return Asset.fromModule(image).downloadAsync();
  });

  await Promise.all(cacheImages);
  this.setState({ isAppReady: true });
}
}

```

Example without AppLoading

```

import React from 'react';
import { Image, Text, View } from 'react-native';
import { Asset, SplashScreen } from 'expo';

export default class App extends React.Component {
  state = {
    isReady: false,
  };

  componentDidMount() {
    SplashScreen.preventAutoHide();
  }

  render() {
    if (!this.state.isReady) {
      return (
        <View style={{ flex: 1 }}>
          <Image
            source={require('./assets/images/splash.gif')}
            onLoad={this._cacheResourcesAsync}
          />
        </View>
      );
    }
  }
}

```

```

        return (
          <View style={{ flex: 1 }}>
            <Image source={require('./assets/images/expo-icon.png')} />
            <Image source={require('./assets/images/slack-icon.png')} />
          </View>
        );
      }

      _cacheSplashResourcesAsync = async () => {
        const gif = require('./assets/images/splash.gif');
        return Asset.fromModule(gif).downloadAsync()
      }

      _cacheResourcesAsync = async () => {
        SplashScreen.hide();
        const images = [
          require('./assets/images/expo-icon.png'),
          require('./assets/images/slack-icon.png'),
        ];

        const cacheImages = images.map((image) => {
          return Asset.fromModule(image).downloadAsync();
        });

        await Promise.all(cacheImages);
        this.setState({ isReady: true });
      }
    }
  }
}

```

Example without any flickering between SplashScreen and it's later continuation

```

import React from 'react';
import { Image, Text, View } from 'react-native';
import { Asset, AppLoading, SplashScreen } from 'expo';

export default class App extends React.Component {
  state = { areResourcesReady: false };

  constructor(props) {
    super(props);
    SplashScreen.preventAutoHide(); // Instruct SplashScreen not to hide yet
  }

  componentDidMount() {
    this.cacheResourcesAsync() // ask for resources
      .then(() => this.setState({ areResourcesReady: true })) // mark resources as loaded
      .catch(error => console.error(`Unexpected error thrown when loading:\n${error.stack}`));
  }

  render() {
    if (!this.state.areResourcesReady) {
      return null;
    }

    return (
      <View style={{ flex: 1 }}>
        <Image
          style={{ flex: 1, resizeMode: 'contain', width: undefined, height: undefined }}
          source={require('./assets/splash.png')}
          onLoadEnd={() => { // wait for image's content to fully load [`Image#onLoadEnd`] (https://facebook.github.io/react-native/docs/image#onloadend)

```

```
        console.log('Image#onLoadEnd: hiding SplashScreen');
        SplashScreen.hide(); // Image is fully presented, instruct SplashScreen to hide
    //}
    fadeDuration={0} // we need to adjust Android devices (https://facebook.github.io/react-native/docs/image#fadeDuration) fadeDuration prop to `0` as it's default value is `300`
    />
</View>
);
}

async cacheResourcesAsync() {
  const images = [
    require('./assets/splash.png'),
  ];
  const cacheImages = images.map(image => Asset.fromModule(image).downloadAsync());
  return Promise.all(cacheImages)
}
}
```

SQLite

This module gives access to a database that can be queried through a [WebSQL](#)-like API. The database is persisted across restarts of your app.

An [example to do list app](#) is available that uses this module for storage.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

API

```
// in managed apps:  
import { SQLite } from 'expo';  
  
// in bare apps:  
import { SQLite } from 'expo-sqlite';
```

`SQLite.openDatabase(name, version, description, size)`

Open a database, creating it if it doesn't exist, and return a `Database` object. On disk, the database will be created under the app's [documents directory](#), i.e. `${Expo.FileSystem.documentDirectory}/SQLite/${name}` .

Arguments

- **name (string)** -- Name of the database file to open.

The `version`, `description` and `size` arguments are ignored, but are accepted by the function for compatibility with the WebSQL specification.

Returns

Returns a `Database` object, described below.

`Database` objects

`Database` objects are returned by calls to `SQLite.openDatabase()`. Such an object represents a connection to a database on your device. They support one method:

- `db.transaction(callback, error, success)`

Execute a database transaction.

Parameters

- **callback (function)** -- A function representing the transaction to perform. Takes a `Transaction` (see below) as its only parameter, on which it can add SQL statements to execute.

- **error (function)** -- Called if an error occurred processing this transaction. Takes a single parameter describing the error.
- **success (function)** -- Called when the transaction has completed executing on the database.

Transaction objects

A `Transaction` object is passed in as a parameter to the `callback` parameter for the `db.transaction()` method on a `Database` (see above). It allows enqueueing SQL statements to perform in a database transaction. It supports one method:

- `tx.executeSql(sqlStatement, arguments, success, error)`

Enqueue a SQL statement to execute in the transaction. Authors are strongly recommended to make use of the `?` placeholder feature of the method to avoid against SQL injection attacks, and to never construct SQL statements on the fly.

Parameters

- **sqlStatement (string)** -- A string containing a database query to execute expressed as SQL. The string may contain `?` placeholders, with values to be substituted listed in the `arguments` parameter.
- **arguments (array)** -- An array of values (numbers or strings) to substitute for `?` placeholders in the SQL statement.
- **success (function)** -- Called when the query is successfully completed during the transaction. Takes two parameters: the transaction itself, and a `ResultSet` object (see below) with the results of the query.
- **error (function)** -- Called if an error occurred executing this particular query in the transaction. Takes two parameters: the transaction itself, and the error object.

ResultSet objects

`ResultSet` objects are returned through second parameter of the `success` callback for the `tx.executeSql()` method on a `Transaction` (see above). They have the following form:

```
{
  insertId,
  rowsAffected,
  rows: {
    length,
    item(),
    _array,
  },
}
```

- **insertId (number)** -- The row ID of the row that the SQL statement inserted into the database, if a row was inserted.
- **rowsAffected (number)** -- The number of rows that were changed by the SQL statement.
- **rows.length (number)** -- The number of rows returned by the query.
- **rows.item (function)** -- `rows.item(index)` returns the row with the given `index`. If there is no such row, returns `null`.
- **rows.array (_number)** -- The actual array of rows returned by the query. Can be used directly instead of getting rows through `rows.item()`.

StoreReview

Provides access to the `SKStoreReviewController` API in iOS 10.3+ devices.

If this is used in Android the device will attempt to link to the Play Store using `ReactNative.Linking` and the `android.playStoreUrl` from the `app.json` instead.

Installation

This API is pre-installed in [managed](#) apps. It is not yet available for [bare](#) React Native apps.

API

```
import { StoreReview } from 'expo';
```

StoreReview.requestReview()

In the ideal circumstance this will open a native modal and allow the user to select a star rating that will then be applied to the App Store without leaving the app. If the users device is running a version of iOS lower than 10.3, or the user is on an Android device, this will attempt to get the store URL and link the user to it.

Example

```
StoreReview.requestReview()
```

StoreReview.isSupported()

This will return true if the device is running iOS 10.3 or greater.

Example

```
StoreReview.isSupported()
```

StoreReview.storeUrl()

This uses the `Constants` API to get the `Constants.manifest.ios.appStoreUrl` on iOS, or the `Constants.manifest.android.playStoreUrl` on Android.

Example

```
const url = StoreReview.storeUrl()
```

StoreReview.hasAction()

This returns a boolean that let's you know if the module can perform any action. This is used for cases where the `app.json` doesn't have the proper fields, and `StoreReview.isSupported()` returns false.

Example

```
if (StoreReview.hasAction()) {  
}
```

Usage

It is important that you follow the [Human Interface Guidelines](#) when using this API.

Specifically:

- Don't call `StoreReview.requestReview()` from a button - instead try calling it after the user has finished some signature interaction in the app.
- Don't spam the user
- Don't request a review when the user is doing something time sensitive like navigating.

Svg

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow the [react-native-svg installation instructions](#).

API

```
import Svg from 'react-native-svg';
```

Svg

A set of drawing primitives such as `Circle`, `Rect`, `Path`, `ClipPath`, and `Polygon`. It supports most SVG elements and properties. The implementation is provided by [react-native-svg](#), and documentation is provided in that repository.

takeSnapshotAsync

Given a view, `takeSnapshotAsync` will essentially screenshot that view and return an image for you. This is very useful for things like signature pads, where the user draws something and then you want to save an image from it.

If you're interested in taking snapshots from the GLView, we recommend you use [GLView's takeSnapshotAsync](#) instead.

Installation

This API is pre-installed in [managed](#) apps. For [bare](#) React Native apps, use the [react-native-view-shot](#) library instead.

API

```
import { takeSnapshotAsync } from 'expo';
```

takeSnapshotAsync(view, options)

Snapshots the given view.

Arguments

- **view (number|ReactElement)** -- The `ref` or `reactTag` (also known as node handle) for the view to snapshot.
- **options (object)** --

An optional map of optional options

- **format (string)** -- `"png"` | `"jpg"` | `"webm"`, defaults to `"png"`, `"webm"` supported only on Android.
- **quality (number)** -- Number between 0 and 1 where 0 is worst quality and 1 is best, defaults to `1`
- **result (string)** -- The type for the resulting image.

```
\- ``tmpfile`` -- (default) Return a temporary file uri.  
\- ``base64`` -- base64 encoded image.  
\- ``data-uri`` -- base64 encoded image with data-uri prefix.
```

- **height (number)** -- Height of result in pixels
- **width (number)** -- Width of result in pixels
- **snapshotContentContainer (bool)** -- if true and when view is a ScrollView, the "content container" height will be evaluated instead of the container height

Returns

An image of the format specified in the options parameter.

Note on pixel values

Remember to take the device `PixelRatio` into account. When you work with pixel values in a UI, most of the time those units are "logical pixels" or "device-independent pixels". With images like PNG files, you often work with "physical pixels". You can get the `PixelRatio` of the device using the React Native API: `PixelRatio.get()`

For example, to save a 'FullHD' picture of `1080x1080`, you would do something like this:

```
const targetPixelCount = 1080; // If you want full HD pictures
const pixelRatio = PixelRatio.get(); // The pixel ratio of the device
// pixels * pixelratio = targetPixelCount, so pixels = targetPixelCount / pixelRatio
const pixels = targetPixelCount / pixelRatio;

const result = await takeSnapshotAsync(this.imageContainer, {
  result: 'file',
  height: pixels,
  width: pixels,
  quality: 1,
  format: 'png',
});
```

TaskManager

An API that allows to manage tasks, especially these running while your app is in the background. Some features of this module are used by other modules under the hood. Here is a list of modules using TaskManager:

- [Location](#)
- [BackgroundFetch](#)

Installation

This API is pre-installed in [managed](#) apps. It is not yet available for [bare](#) React Native apps.

Configuration for standalone apps

`TaskManager` works out of the box in the Expo Client, but some extra configuration is needed for standalone apps. On iOS, each background feature requires a special key in `UIBackgroundModes` array in your `Info.plist` file. In standalone apps this array is empty by default, so in order to use background features you will need to add appropriate keys to your `app.json` configuration. Example of `app.json` that enables background location and background fetch:

```
{
  "expo": {
    ...
    "ios": {
      ...
      "infoPlist": {
        ...
        "UIBackgroundModes": [
          "location",
          "fetch"
        ]
      }
    }
  }
}
```

API

```
import { TaskManager } from 'expo';
```

TaskManager.defineTask(taskName, task)

Defines task function. It must be called in the global scope of your JavaScript bundle. In particular, it **cannot** be called in any of React lifecycle methods like `componentDidMount`. This limitation is due to the fact that when the application is launched in the background, we need to spin up your JavaScript app, run your task and then shut down — no views are mounted in this scenario.

Arguments

- **taskName** (*string*) -- Name of the task.
- **task** (*function*) -- A function that will be invoked when the task with given **taskName** is executed.

TaskManager.isTaskRegisteredAsync(taskName)

Determine whether the task is registered. Registered tasks are stored in a persistent storage and preserved between sessions.

Arguments

- **taskName** (*string*) -- Name of the task.

Returns

Returns a promise resolving to a boolean value whether or not the task with given name is already registered.

TaskManager.getTaskOptionsAsync(taskName)

Retrieves options associated with the task, that were passed to the function registering the task (eg. `Location.startLocationUpdatesAsync`).

Arguments

- **taskName** (*string*) -- Name of the task.

Returns

Returns a promise resolving to the options object that was passed while registering task with given name or `null` if task couldn't be found.

TaskManager.getRegisteredTasksAsync()

Provides information about tasks registered in the app.

Returns

Returns a promise resolving to an array of tasks registered in the app. Example:

```
[
  {
    taskName: 'location-updates-task-name',
    taskType: 'location',
    options: {
      accuracy: Location.Accuracy.High,
      showsBackgroundLocationIndicator: false,
    },
  },
  {
    taskName: 'geofencing-task-name',
    taskType: 'geofencing',
    options: {
      regions: [...],
    },
  },
]
```

]

TaskManager.unregisterTaskAsync(taskName)

Unregisters task from the app, so the app will not be receiving updates for that task anymore. *It is recommended to use methods specialized by modules that registered the task, eg. [Location.stopLocationUpdatesAsync](#).*

Arguments

- **taskName (string)** -- Name of the task to unregister.

Returns

Returns a promise resolving as soon as the task is unregistered.

TaskManager.unregisterAllTasksAsync()

Unregisters all tasks registered for the running app.

Returns

Returns a promise that resolves as soon as all tasks are completely unregistered.

Examples

```
import React from 'react';
import { Location, TaskManager } from 'expo';
import { Text, TouchableOpacity } from 'react-native';

const LOCATION_TASK_NAME = 'background-location-task';

export default class Component extends React.Component {
  onPress = async () => {
    await Location.startLocationUpdatesAsync(LOCATION_TASK_NAME, {
      accuracy: Location.Accuracy.Balanced,
    });
  };

  render() {
    return (
      <TouchableOpacity onPress={this.onPress}>
        <Text>Enable background location</Text>
      </TouchableOpacity>
    );
  }
}

TaskManager.defineTask(LOCATION_TASK_NAME, ({ data, error }) => {
  if (error) {
    // Error occurred - check `error.message` for more details.
    return;
  }
  if (data) {
    const { locations } = data;
    // do something with the locations captured in the background
  }
}
```

});

Updates

API for controlling and responding to over-the-air updates to your app.

Installation

This API is pre-installed in [managed](#) apps. It is not yet available for [bare](#) React Native apps.

API

```
import { Updates } from 'expo';
```

Updates.reload()

Immediately reloads the current experience. This will use your app.json `updates` configuration to fetch and load the newest available JS supported by the device's Expo environment. This is useful for triggering an update of your experience if you have published a new version.

Updates.reloadFromCache()

Immediately reloads the current experience using the most recent cached version. This is useful for triggering an update of your experience if you have published and already downloaded a new version.

Updates.checkForUpdateAsync()

Check if a new published version of your project is available. Does not actually download the update. Rejects if `updates.enabled` is `false` in app.json.

Returns

An object with the following keys:

- **isAvailable (boolean)** -- True if an update is available, false if you're already running the most up-to-date JS bundle.
- **manifest (object)** -- If `isAvailable` is true, the manifest of the available update. Undefined otherwise.

Updates.fetchUpdateAsync(params?)

Downloads the most recent published version of your experience to the device's local cache. Rejects if `updates.enabled` is `false` in app.json.

Arguments

An optional `params` object with the following keys:

- **eventListener (function)** -- A callback to receive updates events. Will be called with the same events as a

function passed into `Updates.addListener` but will be subscribed and unsubscribed from events automatically.

Returns

An object with the following keys:

- **isNew (boolean)** -- True if the fetched bundle is new (i.e. a different version than the what's currently running).
- **manifest (object)** -- Manifest of the fetched update.

Updates.addListener(eventListener)

Invokes a callback when updates-related events occur, either on the initial app load or as a result of a call to `Updates.fetchUpdateAsync`.

Arguments

- **eventListener (function)** -- A callback that is invoked with an updates event.

Returns

An `EventSubscription` object that you can call `remove()` on when you would like to unsubscribe from the listener.

Related types

EventSubscription

Returned from `addListener`.

- **remove() (function)** -- Unsubscribe the listener from future updates.

Event

An object that is passed into each event listener when a new version is available.

- **type (string)** -- Type of the event (see `EventType`).
- **manifest (object)** -- If `type === Updates.EventType.DOWNLOAD_FINISHED`, the manifest of the newly downloaded update. Undefined otherwise.
- **message (string)** -- If `type === Updates.EventType.ERROR`, the error message. Undefined otherwise.

EventType

- `Updates.EventType.DOWNLOAD_STARTED` -- A new update is available and has started downloading.
- `Updates.EventType.DOWNLOAD_FINISHED` -- A new update has finished downloading and is now stored in the device's cache.
- `Updates.EventType.NO_UPDATE_AVAILABLE` -- No updates are available, and the most up-to-date bundle of this experience is already running.
- `Updates.EventType.ERROR` -- An error occurred trying to fetch the latest update.

Video

A component that displays a video inline with the other React Native UI elements in your app. The display dimensions and position of the video on screen can be set using usual React Native styling.

Much of Video and Audio have common APIs that are documented in [AV documentation](#). This page covers video-specific props and APIs. We encourage you to skim through this document to get basic video working, and then move on to [AV documentation](#) for more advanced functionality. The audio experience of video (such as whether to interrupt music already playing in another app, or whether to play sound while the phone is on silent mode) can be customized using the [Audio API](#).

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

Here's a simple example of a video that autoplays and loops.

```
<Video
  source={{ uri: 'http://d23dyxeqlo5psv.cloudfront.net/big_buck_bunny.mp4' }}
  rate={1.0}
  volume={1.0}
  isMuted={false}
  resizeMode="cover"
  shouldPlay
  isLooping
  style={{ width: 300, height: 300 }}>
</Video>
```

For more advanced examples, check out the [Playlist example](#), and the [custom videoplayer controls component](#) that wraps `<Video>`, adds custom controls and use the `<Video>` API extensively. The videoplayer controls is used in [this app](#).

API

```
// in managed apps:
import { Video } from 'expo';

// in bare apps:
import { Video } from 'expo-av';
```

props

The `source` and `posterSource` props customize the source of the video content.

- `source`

The source of the video data to display. If this prop is `null`, or left blank, the video component will display nothing.

Note that this can also be set on the `ref` via `loadAsync()`; see below or the [AV documentation](#) for further information.

The following forms for the source are supported:

- A dictionary of the form `{ uri: string, headers?: { [string]: string }, overrideFileExtensionAndroid?: string }` with a network URL pointing to a video file on the web, an optional headers object passed in a network request to the `uri` and an optional Android-specific `overrideFileExtensionAndroid` string overriding extension inferred from the URL. The `overrideFileExtensionAndroid` property may come in handy if the player receives an URL like `example.com/play` which redirects to `example.com/player.m3u8`. Setting this property to `m3u8` would allow the Android player to properly infer the content type of the media and use proper media file reader.
- `require('path/to/file')` for a video file asset in the source code directory.
- An `Asset` object for a video file asset.

The [iOS developer documentation](#) lists the video formats supported on iOS.

The [Android developer documentation](#) lists the video formats supported on Android.

- `posterSource`

The source of an optional image to display over the video while it is loading. The following forms are supported:

- A dictionary of the form `{ uri: 'http://path/to/file' }` with a network URL pointing to a image file on the web.
- `require('path/to/file')` for an image file asset in the source code directory.

The `useNativeControls`, `resizeMode`, and `usePoster` props customize the UI of the component.

- `useNativeControls`

A boolean which, if set to `true`, will display native playback controls (such as play and pause) within the `Video` component. If you'd prefer to use custom controls, you can write them yourself, and/or check out the [Videoplayer component](#).

- `resizeMode`

A string describing how the video should be scaled for display in the component view bounds. Must be one of the following values:

- `Video.RESIZE_MODE_STRETCH` -- Stretch to fill component bounds.
- `Video.RESIZE_MODE_CONTAIN` -- Fit within component bounds while preserving aspect ratio.
- `Video.RESIZE_MODE_COVER` -- Fill component bounds while preserving aspect ratio.

- `usePoster`

A boolean which, if set to `true`, will display an image (whose source is set via the prop `posterSource`) while the video is loading.

The `onPlaybackStatusUpdate`, `onReadyForDisplay`, and `onIOSFullscreenUpdate` props pass information of the state of the `Video` component. The `onLoadStart`, `onLoad`, and `onError` props are also provided for backwards compatibility with `Image` (but they are redundant with `onPlaybackStatusUpdate`).

- `onPlaybackStatusUpdate`

A function to be called regularly with the `PlaybackStatus` of the video. You will likely be using this a lot. See the [AV documentation](#) for further information on `onPlaybackStatusUpdate`, and the interval at which it is called.

- `onReadyForDisplay`

A function to be called when the video is ready for display. Note that this function gets called whenever the video's natural size changes.

The function is passed a dictionary with the following key-value pairs:

- `naturalSize` : a dictionary with the following key-value pairs:
 - `width` : a number describing the width in pixels of the video data
 - `height` : a number describing the height in pixels of the video data
 - `orientation` : a string describing the natural orientation of the video data, either `'portrait'` or `'landscape'`
- `status` : the `PlaybackStatus` of the video; see the [AV documentation](#) for further information.

- `onFullscreenUpdate`

A function to be called when the state of the native iOS fullscreen view changes (controlled via the `presentFullscreenPlayer()` and `dismissFullscreenPlayer()` methods on the `Video`'s `ref`).

The function is passed a dictionary with the following key-value pairs:

- `fullscreenUpdate` : a number taking one of the following values:
 - `Video.FULLSCREEN_UPDATE_PLAYER_WILL_PRESENT` : describing that the fullscreen player is about to present
 - `Video.FULLSCREEN_UPDATE_PLAYER_DID_PRESENT` : describing that the fullscreen player just finished presenting
 - `Video.FULLSCREEN_UPDATE_PLAYER_WILL_DISMISS` : describing that the fullscreen player is about to dismiss
 - `Video.FULLSCREEN_UPDATE_PLAYER_DID_DISMISS` : describing that the fullscreen player just finished dismissing
- `status` : the `PlaybackStatus` of the video; see the [AV documentation](#) for further information.

- `onLoadStart`

A function to be called when the video begins to be loaded into memory. Called without any arguments.

- `onLoad`

A function to be called once the video has been loaded. The data is streamed so all of it may not have been fetched yet, just enough to render the first frame. The function is called with the `PlaybackStatus` of the video as its parameter; see the [AV documentation](#) for further information.

- `onError`

A function to be called if load or playback have encountered a fatal error. The function is passed a single error message string as a parameter. Errors sent here are also set on `playbackStatus.error` that are passed into the `onPlaybackStatusUpdate` callback.

Finally, the following props are available to control the playback of the video, but we recommend that you use the methods available on the `ref` (described below and in the [AV documentation](#)) for finer control.

- `status`

A dictionary setting a new `PlaybackStatusToSet` on the video. See the [AV documentation](#) for more information ON `PlaybackStatusToSet`.

- `progressUpdateIntervalMillis`

A number describing the new minimum interval in milliseconds between calls of `onPlaybackStatusUpdate`. See the [AV documentation](#) for more information.

- `positionMillis`

The desired position of playback in milliseconds. See the [AV documentation](#) for more information.

- `shouldPlay`

A boolean describing if the media is supposed to play. Playback may not start immediately after setting this value for reasons such as buffering. Make sure to update your UI based on the `isPlaying` and `isBuffering` properties of the `PlaybackStatus`. See the [AV documentation](#) for more information.

- `rate`

The desired playback rate of the media. This value must be between `0.0` and `32.0`. Only available on Android API version 23 and later and iOS. See the [AV documentation](#) for more information.

- `shouldCorrectPitch`

A boolean describing if we should correct the pitch for a changed rate. If set to `true`, the pitch of the audio will be corrected (so a rate different than `1.0` will timestretch the audio). See the [AV documentation](#) for more information.

- `volume`

The desired volume of the audio for this media. This value must be between `0.0` (silence) and `1.0` (maximum volume). See the [AV documentation](#) for more information.

- `isMuted`

A boolean describing if the audio of this media should be muted. See the [AV documentation](#) for more information.

- `isLooping`

A boolean describing if the media should play once (`false`) or loop indefinitely (`true`). See the [AV documentation](#) for more information.

The following methods are available on the component's ref:

- `videoRef.presentFullscreenPlayer()`

This presents a fullscreen view of your video component on top of your app's UI. Note that even if `useNativeControls` is set to `false`, native controls will be visible in fullscreen mode.

Returns

A `Promise` that is fulfilled with the `PlaybackStatus` of the video once the fullscreen player has finished presenting, or rejects if there was an error, or if this was called on an Android device.

- `videoRef.dismissFullscreenPlayer()`

This dismisses the fullscreen video view.

Returns

A `Promise` that is fulfilled with the `PlaybackStatus` of the video once the fullscreen player has finished dismissing, or rejects if there was an error, or if this was called on an Android device.

The rest of the API on the `Video` component ref is the same as the API for `Audio.Sound` -- see the [AV documentation](#) for further information:

- `videoRef.loadAsync(source, initialStatus = {}, downloadFirst = true)`
- `videoRef.unloadAsync()`
- `videoRef.getStatusAsync()`
- `videoRef.setOnPlaybackStatusUpdate(onPlaybackStatusUpdate)`
- `videoRef.setStatusAsync(statusToSet)`
- `videoRef.playAsync()`
- `videoRef.replayAsync()`
- `videoRef.pauseAsync()`
- `videoRef.stopAsync()`
- `videoRef.setPositionAsync(millis)`
- `videoRef.setRateAsync(value, shouldCorrectPitch)`
- `videoRef.setVolumeAsync(value)`
- `videoRef.setIsMutedAsync(value)`
- `videoRef.setIsLoopingAsync(value)`
- `videoRef.setProgressUpdateIntervalAsync(millis)`

WebBrowser

Provides access to the system's web browser and supports handling redirects. On iOS, it uses `SFSafariViewController` OR `SFAuthenticationSession`, depending on the method you call, and on Android it uses `ChromeCustomTabs`. As of iOS 11, `SFSafariViewController` no longer shares cookies with the Safari, so if you are using `WebBrowser` for authentication you will want to use `WebBrowser.openAuthSessionAsync`, and if you just want to open a webpage (such as your app privacy policy), then use `WebBrowser.openBrowserAsync`.

Installation

This API is pre-installed in [managed](#) apps. To use it in a [bare](#) React Native app, follow its [installation instructions](#).

Usage

Handling deep links from the WebBrowser

If you are using the `WebBrowser` window for authentication or another use case where you would like to pass information back into your app through a deep link, be sure to add a handler with `Linking.addEventListerner` before opening the browser. When the listener fires, you should call `WebBrowser.dismissBrowser()` -- it will not automatically dismiss when a deep link is handled. Aside from that, redirects from `WebBrowser` work the same as other deep links. [Read more about it in the Linking guide](#).

API

```
// in managed apps:  
import { WebBrowser } from 'expo';  
  
// in bare apps:  
import * as WebBrowser from 'expo-web-browser';
```

WebBrowser.openBrowserAsync(url)

Opens the url with Safari in a modal on iOS using `SFSafariViewController`, and Chrome in a new [custom tab](#) on Android. On iOS, the modal Safari will not share cookies with the system Safari. If you need this, use `WebBrowser.openAuthSessionAsync`.

Arguments

- **url (string)** -- The url to open in the web browser.

Returns

Returns a Promise:

- If the user closed the web browser, the Promise resolves with `{ type: 'cancel' }`.
- If the browser is closed using `WebBrowser.dismissBrowser()`, the Promise resolves with `{ type: 'dismiss' }`.

WebBrowser.openAuthSessionAsync(url, redirectUrl)

Opens the url with Safari in a modal on iOS using `SFAuthenticationSession`, and Chrome in a new [custom tab](#) on Android. On iOS, the user will be asked whether to allow the app to authenticate using the given url.

Arguments

- **url (string)** -- The url to open in the web browser. This should be a login page.
- **redirectUrl (string) -- Optional:** the url to deep link back into your app. By default, this will be [Constants.linkUrl](#)

Returns a Promise:

- If the user does not permit the application to authenticate with the given url, the Promise resolved with `{ type: 'cancel' }`.
- If the user closed the web browser, the Promise resolves with `{ type: 'cancel' }`.
- If the browser is closed using `WebBrowser.dismissBrowser()`, the Promise resolves with `{ type: 'dismiss' }`.

WebBrowser.dismissBrowser()

Dismisses the system's presented web browser. On Android calling this method doesn't actually do anything, the user still has to manually dismiss the browser.

Returns

The promise resolves with `{ type: 'dismiss' }`.

#

Learn the Basics

React Native is like React, but it uses native components instead of web components as building blocks. So to understand the basic structure of a React Native app, you need to understand some of the basic React concepts, like JSX, components, `state`, and `props`. If you already know React, you still need to learn some React-Native-specific stuff, like the native components. This tutorial is aimed at all audiences, whether you have React experience or not.

Let's do this thing.

Hello World

In accordance with the ancient traditions of our people, we must first build an app that does nothing except say "Hello world". Here it is:

```
import React, { Component } from 'react';
import { Text, View } from 'react-native';

export default class HelloWorldApp extends Component {
  render() {
    return (
      <View>
        <Text>Hello world!</Text>
      </View>
    );
  }
}
```

If you are feeling curious, you can play around with sample code directly in the web simulators. You can also paste it into your `App.js` file to create a real app on your local machine.

What's going on here?

Some of the things in here might not look like JavaScript to you. Don't panic. *This is the future.*

First of all, ES2015 (also known as ES6) is a set of improvements to JavaScript that is now part of the official standard, but not yet supported by all browsers, so often it isn't used yet in web development. React Native ships with ES2015 support, so you can use this stuff without worrying about compatibility. `import`, `from`, `class`, and `extends` in the example above are all ES2015 features. If you aren't familiar with ES2015, you can probably pick it up just by reading through sample code like this tutorial has. If you want, [this page](#) has a good overview of ES2015 features.

The other unusual thing in this code example is `<view><text>Hello world!</text></view>`. This is JSX - a syntax for embedding XML within JavaScript. Many frameworks use a special templating language which lets you embed code inside markup language. In React, this is reversed. JSX lets you write your markup language inside code. It looks like HTML on the web, except instead of web things like `<div>` or ``, you use React components. In this case, `<text>` is a built-in component that just displays some text and `view` is like the `<div>` or ``.

Components

So this code is defining `HelloWorldApp`, a new `Component`. When you're building a React Native app, you'll be making new components a lot. Anything you see on the screen is some sort of component. A component can be pretty simple - the only thing that's required is a `render` function which returns some JSX to render.

This app doesn't do very much

Good point. To make components do more interesting things, you need to [learn about Props](#).

Props

Most components can be customized when they are created, with different parameters. These creation parameters are called `props`.

For example, one basic React Native component is the `Image`. When you create an image, you can use a prop named `source` to control what image it shows.

```
import React, { Component } from 'react';
import { AppRegistry, Image } from 'react-native';

export default class Bananas extends Component {
  render() {
    let pic = {
      uri: 'https://upload.wikimedia.org/wikipedia/commons/d/de/Bananavarieties.jpg'
    };
    return (
      <Image source={pic} style={{width: 193, height: 110}}/>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => Bananas);
```

Notice the braces surrounding `{pic}` - these embed the variable `pic` into JSX. You can put any JavaScript expression inside braces in JSX.

Your own components can also use `props`. This lets you make a single component that is used in many different places in your app, with slightly different properties in each place. Just refer to `this.props` in your `render` function. Here's an example:

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

class Greeting extends Component {
  render() {
    return (
      <View style={{alignItems: 'center'}}>
        <Text>Hello {this.props.name}!</Text>
      </View>
    );
  }
}

export default class LotsOfGreetings extends Component {
  render() {
    return (
      <View style={{alignItems: 'center'}}>
        <Greeting name='Rexxar' />
        <Greeting name='Jaina' />
        <Greeting name='Valeera' />
      </View>
    );
  }
}
```

```
// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => LotsOfGreetings);
```

Using `name` as a prop lets us customize the `Greeting` component, so we can reuse that component for each of our greetings. This example also uses the `Greeting` component in JSX, just like the built-in components. The power to do this is what makes React so cool - if you find yourself wishing that you had a different set of UI primitives to work with, you just invent new ones.

The other new thing going on here is the `View` component. A `View` is useful as a container for other components, to help control style and layout.

With `props` and the basic `Text`, `Image`, and `View` components, you can build a wide variety of static screens. To learn how to make your app change over time, you need to [learn about State](#).

State

There are two types of data that control a component: `props` and `state`. `props` are set by the parent and they are fixed throughout the lifetime of a component. For data that is going to change, we have to use `state`.

In general, you should initialize `state` in the constructor, and then call `setState` when you want to change it.

For example, let's say we want to make text that blinks all the time. The text itself gets set once when the blinking component gets created, so the text itself is a `prop`. The "whether the text is currently on or off" changes over time, so that should be kept in `state`.

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

class Blink extends Component {
  constructor(props) {
    super(props);
    this.state = { isShowingText: true };

    // Toggle the state every second
    setInterval(() => (
      this.setState(previousState => (
        { isShowingText: !previousState.isShowingText }
      ))
    ), 1000);
  }

  render() {
    if (!this.state.isShowingText) {
      return null;
    }

    return (
      <Text>{this.props.text}</Text>
    );
  }
}

export default class BlinkApp extends Component {
  render() {
    return (
      <View>
        <Blink text='I love to blink' />
        <Blink text='Yes blinking is so great' />
        <Blink text='Why did they ever take this out of HTML' />
        <Blink text='Look at me look at me look at me' />
      </View>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => BlinkApp);
```

In a real application, you probably won't be setting state with a timer. You might set state when you have new data arrived from the server, or from user input. You can also use a state container like [Redux](#) or [Mobx](#) to control your data flow. In that case you would use Redux or Mobx to modify your state rather than calling `setState`.

directly.

When `setState` is called, `BlinkApp` will re-render its Component. By calling `setState` within the Timer, the component will re-render every time the Timer ticks.

State works the same way as it does in React, so for more details on handling state, you can look at the [React.Component API](#). At this point, you might be annoyed that most of our examples so far use boring default black text. To make things more beautiful, you will have to [learn about Style](#).

Style

With React Native, you don't use a special language or syntax for defining styles. You just style your application using JavaScript. All of the core components accept a prop named `style`. The style names and `values` usually match how CSS works on the web, except names are written using camel casing, e.g `backgroundColor` rather than `background-color`.

The `style` prop can be a plain old JavaScript object. That's the simplest and what we usually use for example code. You can also pass an array of styles - the last style in the array has precedence, so you can use this to inherit styles.

As a component grows in complexity, it is often cleaner to use `StyleSheet.create` to define several styles in one place. Here's an example:

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

const styles = StyleSheet.create({
  bigblue: {
    color: 'blue',
    fontWeight: 'bold',
    fontSize: 30,
  },
  red: {
    color: 'red',
  },
});

export default class LotsOfStyles extends Component {
  render() {
    return (
      <View>
        <Text style={styles.red}>just red</Text>
        <Text style={styles.bigblue}>just bigblue</Text>
        <Text style={[styles.bigblue, styles.red]}>bigblue, then red</Text>
        <Text style={[styles.red, styles.bigblue]}>red, then bigblue</Text>
      </View>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => LotsOfStyles);
```

One common pattern is to make your component accept a `style` prop which in turn is used to style subcomponents. You can use this to make styles "cascade" the way they do in CSS.

There are a lot more ways to customize text style. Check out the [Text component reference](#) for a complete list.

Now you can make your text beautiful. The next step in becoming a style master is to [learn how to control component size](#).

Height and Width

A component's height and width determine its size on the screen.

Fixed Dimensions

The simplest way to set the dimensions of a component is by adding a fixed `width` and `height` to style. All dimensions in React Native are unitless, and represent density-independent pixels.

```
import React, { Component } from 'react';
import { AppRegistry, View } from 'react-native';

export default class FixedDimensionsBasics extends Component {
  render() {
    return (
      <View>
        <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
        <View style={{width: 100, height: 100, backgroundColor: 'skyblue'}} />
        <View style={{width: 150, height: 150, backgroundColor: 'steelblue'}} />
      </View>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => FixedDimensionsBasics);
```

Setting dimensions this way is common for components that should always render at exactly the same size, regardless of screen dimensions.

Flex Dimensions

Use `flex` in a component's style to have the component expand and shrink dynamically based on available space. Normally you will use `flex: 1`, which tells a component to fill all available space, shared evenly amongst other components with the same parent. The larger the `flex` given, the higher the ratio of space a component will take compared to its siblings.

A component can only expand to fill available space if its parent has dimensions greater than 0. If a parent does not have either a fixed `width` and `height` OR `flex`, the parent will have dimensions of 0 and the `flex` children will not be visible.

```
import React, { Component } from 'react';
import { AppRegistry, View } from 'react-native';

export default class FlexDimensionsBasics extends Component {
  render() {
    return (
      // Try removing the `flex: 1` on the parent View.
      // The parent will not have dimensions, so the children can't expand.
      // What if you add `height: 300` instead of `flex: 1`?
      <View style={{flex: 1}}>
```

```
        <View style={{flex: 1, backgroundColor: 'powderblue'}} />
        <View style={{flex: 2, backgroundColor: 'skyblue'}} />
        <View style={{flex: 3, backgroundColor: 'steelblue'}} />
    </View>
);
}
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => FlexDimensionsBasics);
```

After you can control a component's size, the next step is to [learn how to lay it out on the screen](#).

Layout with Flexbox

A component can specify the layout of its children using the flexbox algorithm. Flexbox is designed to provide a consistent layout on different screen sizes.

You will normally use a combination of `flexDirection`, `alignItems`, and `justifyContent` to achieve the right layout.

Flexbox works the same way in React Native as it does in CSS on the web, with a few exceptions. The defaults are different, with `flexDirection` defaulting to `column` instead of `row`, and the `flex` parameter only supporting a single number.

Flex Direction

Adding `flexDirection` to a component's `style` determines the **primary axis** of its layout. Should the children be organized horizontally (`row`) or vertically (`column`)? The default is `column`.

```
import React, { Component } from 'react';
import { AppRegistry, View } from 'react-native';

export default class FlexDirectionBasics extends Component {
  render() {
    return (
      // Try setting `flexDirection` to `column`.
      <View style={{flex: 1, flexDirection: 'row'}>
        <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
        <View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />
        <View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />
      </View>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => FlexDirectionBasics);
```

Justify Content

Adding `justifyContent` to a component's style determines the **distribution** of children along the **primary axis**. Should children be distributed at the start, the center, the end, or spaced evenly? Available options are `flex-start`, `center`, `flex-end`, `space-around`, `space-between` and `space-evenly`.

```
import React, { Component } from 'react';
import { AppRegistry, View } from 'react-native';

export default class JustifyContentBasics extends Component {
  render() {
    return (
      // Try setting `justifyContent` to `center`.
      // Try setting `flexDirection` to `row`.
      <View style={{
        flex: 1,
        flexDirection: 'column',
        justifyContent: 'space-between',
      }}>
    );
  }
}
```

```

        <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
        <View style={{width: 50, height: 50, backgroundColor: 'skyblue'}} />
        <View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />
    </View>
);
}
);

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => JustifyContentBasics);

```

Align Items

Adding `alignItems` to a component's style determines the **alignment** of children along the **secondary axis** (if the primary axis is `row`, then the secondary is `column`, and vice versa). Should children be aligned at the start, the center, the end, or stretched to fill? Available options are `flex-start`, `center`, `flex-end`, and `stretch`.

For `stretch` to have an effect, children must not have a fixed dimension along the secondary axis. In the following example, setting `alignItems: stretch` does nothing until the `width: 50` is removed from the children.

```

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

export default class AlignItemsBasics extends Component {
  render() {
    return (
      // Try setting `alignItems` to 'flex-start'
      // Try setting `justifyContent` to `flex-end`.
      // Try setting `flexDirection` to `row`.
      <View style={{
        flex: 1,
        flexDirection: 'column',
        justifyContent: 'center',
        alignItems: 'stretch',
      }}>
        <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
        <View style={{height: 50, backgroundColor: 'skyblue'}} />
        <View style={{height: 100, backgroundColor: 'steelblue'}} />
      </View>
    );
  }
};

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => AlignItemsBasics);

```

Going Deeper

We've covered the basics, but there are many other styles you may need for layouts. The full list of props that control layout is documented [here](#).

We're getting close to being able to build a real application. One thing we are still missing is a way to take user input, so let's move on to [learn how to handle text input with the TextInput component](#).

Handling Text Input

`TextInput` is a basic component that allows the user to enter text. It has an `onChangeText` prop that takes a function to be called every time the text changed, and an `onSubmitEditing` prop that takes a function to be called when the text is submitted.

For example, let's say that as the user types, you're translating their words into a different language. In this new language, every single word is written the same way: . So the sentence "Hello there Bob" would be translated as "".

```
import React, { Component } from 'react';
import { AppRegistry, Text, TextInput, View } from 'react-native';

export default class PizzaTranslator extends Component {
  constructor(props) {
    super(props);
    this.state = {text: ''};
  }

  render() {
    return (
      <View style={{padding: 10}}>
        <TextInput
          style={{height: 40}}
          placeholder="Type here to translate!"
          onChangeText={(text) => this.setState({text})}
        />
        <Text style={{padding: 10, fontSize: 42}}>
          {this.state.text.split(' ').map((word) => word && '')}.join(' ')
        </Text>
      </View>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => PizzaTranslator);
```

In this example, we store `text` in the state, because it changes over time.

There are a lot more things you might want to do with a text input. For example, you could validate the text inside while the user types. For more detailed examples, see the [React docs on controlled components](#), or the [reference docs for TextInput](#).

Text input is one of the ways the user interacts with the app. Next, let's look at another type of input and [learn how to handle touches](#).

Handling Touches

Users interact with mobile apps mainly through touch. They can use a combination of gestures, such as tapping on a button, scrolling a list, or zooming on a map. React Native provides components to handle all sorts of common gestures, as well as a comprehensive [gesture responder system](#) to allow for more advanced gesture recognition, but the one component you will most likely be interested in is the basic Button.

Displaying a basic button

[Button](#) provides a basic button component that is rendered nicely on all platforms. The minimal example to display a button looks like this:

```
<Button
  onPress={() => {
    Alert.alert('You tapped the button!');
  }}
  title="Press Me"
/>
```

This will render a blue label on iOS, and a blue rounded rectangle with white text on Android. Pressing the button will call the "onPress" function, which in this case displays an alert popup. If you like, you can specify a "color" prop to change the color of your button.



Go ahead and play around with the `Button` component using the example below. You can select which platform your app is previewed in by clicking on the toggle in the bottom right, then click on "Tap to Play" to preview the app.

```
import React, { Component } from 'react';
import { Alert, AppRegistry, Button, StyleSheet, View } from 'react-native';

export default class ButtonBasics extends Component {
  _onPressButton() {
    Alert.alert('You tapped the button!')
  }

  render() {
    return (
      <View style={styles.container}>
        <View style={styles.buttonContainer}>
          <Button
            onPress={this._onPressButton}
            title="Press Me"
          />
        </View>
        <View style={styles.buttonContainer}>
```

```

        <Button
          onPress={this._onPressButton}
          title="Press Me"
          color="#841584"
        />
      </View>
      <View style={styles.alternativeLayoutButtonContainer}>
        <Button
          onPress={this._onPressButton}
          title="This looks great!"
        />
        <Button
          onPress={this._onPressButton}
          title="OK!"
          color="#841584"
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
  },
  buttonContainer: {
    margin: 20
  },
  alternativeLayoutButtonContainer: {
    margin: 20,
    flexDirection: 'row',
    justifyContent: 'space-between'
  }
});

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => ButtonBasics);

```

Touchables

If the basic button doesn't look right for your app, you can build your own button using any of the "Touchable" components provided by React Native. The "Touchable" components provide the capability to capture tapping gestures, and can display feedback when a gesture is recognized. These components do not provide any default styling, however, so you will need to do a bit of work to get them looking nicely in your app.

Which "Touchable" component you use will depend on what kind of feedback you want to provide:

- Generally, you can use **TouchableHighlight** anywhere you would use a button or link on web. The view's background will be darkened when the user presses down on the button.
- You may consider using **TouchableNativeFeedback** on Android to display ink surface reaction ripples that respond to the user's touch.
- **TouchableOpacity** can be used to provide feedback by reducing the opacity of the button, allowing the background to be seen through while the user is pressing down.
- If you need to handle a tap gesture but you don't want any feedback to be displayed, use **TouchableWithoutFeedback**.

In some cases, you may want to detect when a user presses and holds a view for a set amount of time. These long presses can be handled by passing a function to the `onLongPress` props of any of the "Touchable" components.

Let's see all of these in action:

```
import React, { Component } from 'react';
import { Alert, AppRegistry, Platform, StyleSheet, Text, TouchableHighlight, TouchableOpacity, TouchableNativeFeedback, TouchableWithoutFeedback, View } from 'react-native';

export default class Touchables extends Component {
  _onPressButton() {
    Alert.alert('You tapped the button!')
  }

  _onLongPressButton() {
    Alert.alert('You long-pressed the button!')
  }

  render() {
    return (
      <View style={styles.container}>
        <TouchableHighlight onPress={this._onPressButton} underlayColor="white">
          <View style={styles.button}>
            <Text style={styles.buttonText}>TouchableHighlight</Text>
          </View>
        </TouchableHighlight>
        <TouchableOpacity onPress={this._onPressButton}>
          <View style={styles.button}>
            <Text style={styles.buttonText}>TouchableOpacity</Text>
          </View>
        </TouchableOpacity>
        <TouchableNativeFeedback
          onPress={this._onPressButton}
          background={Platform.OS === 'android' ? TouchableNativeFeedback.SelectableBackground() : ''}>
          <View style={styles.button}>
            <Text style={styles.buttonText}>TouchableNativeFeedback</Text>
          </View>
        </TouchableNativeFeedback>
        <TouchableWithoutFeedback
          onPress={this._onPressButton}>
          <View style={styles.button}>
            <Text style={styles.buttonText}>TouchableWithoutFeedback</Text>
          </View>
        </TouchableWithoutFeedback>
        <TouchableHighlight onPress={this._onPressButton} onLongPress={this._onLongPressButton} underlayColor="white">
          <View style={styles.button}>
            <Text style={styles.buttonText}>Touchable with Long Press</Text>
          </View>
        </TouchableHighlight>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    paddingTop: 60,
    alignItems: 'center'
  },
});
```

```
button: {
  marginBottom: 30,
  width: 260,
  alignItems: 'center',
  backgroundColor: '#2196F3'
},
buttonText: {
  padding: 20,
  color: 'white'
});
};

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => Touchables);
```

Scrolling lists, swiping pages, and pinch-to-zoom

Another gesture commonly used in mobile apps is the swipe or pan. This gesture allows the user to scroll through a list of items, or swipe through pages of content. In order to handle these and other gestures, we'll learn [how to use a ScrollView](#) next.

Using a ScrollView

The `ScrollView` is a generic scrolling container that can host multiple components and views. The scrollable items need not be homogeneous, and you can scroll both vertically and horizontally (by setting the `horizontal` property).

This example creates a vertical `ScrollView` with both images and text mixed together.

```
import React, { Component } from 'react';
import { AppRegistry, ScrollView, Image, Text } from 'react-native';

export default class IScrolledDownAndWhatHappenedNextShockedMe extends Component {
  render() {
    return (
      <ScrollView>
        <Text style={{fontSize:96}}>Scroll me plz</Text>
        <Image source={{uri: "https://facebook.github.io/react-native/img/favicon.png", width: 64, height: 64}} />
        <Text style={{fontSize:96}}>If you like</Text>
        <Image source={{uri: "https://facebook.github.io/react-native/img/favicon.png", width: 64, height: 64}} />
        <Text style={{fontSize:96}}>Scrolling down</Text>
        <Image source={{uri: "https://facebook.github.io/react-native/img/favicon.png", width: 64, height: 64}} />
        <Text style={{fontSize:96}}>What's the best</Text>
        <Image source={{uri: "https://facebook.github.io/react-native/img/favicon.png", width: 64, height: 64}} />
        <Text style={{fontSize:96}}>Framework around?</Text>
        <Image source={{uri: "https://facebook.github.io/react-native/img/favicon.png", width: 64, height: 64}} />
        <Text style={{fontSize:80}}>React Native</Text>
      </ScrollView>
    );
  }
}

// skip these lines if using Create React Native App
AppRegistry.registerComponent(
  'AwesomeProject',
  () => IScrolledDownAndWhatHappenedNextShockedMe);
```

ScrollViews can be configured to allow paging through views using swiping gestures by using the `pagingEnabled` props. Swiping horizontally between views can also be implemented on Android using the [ViewPagerAndroid](#) component.

A ScrollView with a single item can be used to allow the user to zoom content. Set up the `maximumZoomScale` and `minimumZoomScale` props and your user will be able to use pinch and expand gestures to zoom in and out.

The ScrollView works best to present a small amount of things of a limited size. All the elements and views of a `ScrollView` are rendered, even if they are not currently shown on the screen. If you have a long list of more items than can fit on the screen, you should use a `FlatList` instead. So let's [learn about list views](#) next.

Using List Views

React Native provides a suite of components for presenting lists of data. Generally, you'll want to use either [FlatList](#) or [SectionList](#).

The `FlatList` component displays a scrolling list of changing, but similarly structured, data. `FlatList` works well for long lists of data, where the number of items might change over time. Unlike the more generic `ScrollView`, the `FlatList` only renders elements that are currently showing on the screen, not all the elements at once.

The `FlatList` component requires two props: `data` and `renderItem`. `data` is the source of information for the list. `renderItem` takes one item from the source and returns a formatted component to render.

This example creates a simple `FlatList` of hardcoded data. Each item in the `data` prop is rendered as a `Text` component. The `FlatListBasics` component then renders the `FlatList` and all `Text` components.

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

export default class FlatListBasics extends Component {
  render() {
    return (
      <View style={styles.container}>
        <FlatList
          data={[
            {key: 'Devin'},
            {key: 'Jackson'},
            {key: 'James'},
            {key: 'Joel'},
            {key: 'John'},
            {key: 'Jillian'},
            {key: 'Jimmy'},
            {key: 'Julie'},
          ]}
          renderItem={({item}) => <Text style={styles.item}>{item.key}</Text>}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 22
  },
  item: {
    padding: 10,
    fontSize: 18,
    height: 44,
  },
})

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => FlatListBasics);
```

If you want to render a set of data broken into logical sections, maybe with section headers, similar to `UITableView`s on iOS, then a [SectionList](#) is the way to go.

```

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

export default class SectionListBasics extends Component {
  render() {
    return (
      <View style={styles.container}>
        <SectionList
          sections={[
            {title: 'D', data: ['Devin']},
            {title: 'J', data: ['Jackson', 'James', 'Jillian', 'Jimmy', 'Joel', 'John', 'Julie']},
          ]}
          renderItem={({item}) => <Text style={styles.item}>{item}</Text>}
          renderSectionHeader={({section}) => <Text style={styles.sectionHeader}>{section.title}</Text>}
          keyExtractor={({item, index}) => index}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    paddingTop: 22
  },
  sectionHeader: {
    paddingTop: 2,
    paddingLeft: 10,
    paddingRight: 10,
    paddingBottom: 2,
    fontSize: 14,
    fontWeight: 'bold',
    backgroundColor: 'rgba(247,247,247,1.0)',
  },
  item: {
    padding: 10,
    fontSize: 18,
    height: 44,
  },
})

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => SectionListBasics);

```

One of the most common uses for a list view is displaying data that you fetch from a server. To do that, you will need to [learn about networking in React Native](#).

Networking

Many mobile apps need to load resources from a remote URL. You may want to make a POST request to a REST API, or you may simply need to fetch a chunk of static content from another server.

Using Fetch

React Native provides the [Fetch API](#) for your networking needs. Fetch will seem familiar if you have used `XMLHttpRequest` or other networking APIs before. You may refer to MDN's guide on [Using Fetch](#) for additional information.

Making requests

In order to fetch content from an arbitrary URL, just pass the URL to `fetch`:

```
fetch('https://mywebsite.com/mydata.json');
```

Fetch also takes an optional second argument that allows you to customize the HTTP request. You may want to specify additional headers, or make a POST request:

```
fetch('https://mywebsite.com/endpoint/' , {
  method: 'POST',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    firstParam: 'yourValue',
    secondParam: 'yourOtherValue',
  }),
});
```

Take a look at the [Fetch Request docs](#) for a full list of properties.

Handling the response

The above examples show how you can make a request. In many cases, you will want to do something with the response.

Networking is an inherently asynchronous operation. Fetch methods will return a [Promise](#) that makes it straightforward to write code that works in an asynchronous manner:

```
function getMoviesFromApiAsync() {
  return fetch('https://facebook.github.io/react-native/movies.json')
    .then((response) => response.json())
    .then((responseJson) => {
      return responseJson.movies;
    })
    .catch((error) => {
```

```
        console.error(error);
    });
}
```

You can also use the proposed ES2017 `async / await` syntax in a React Native app:

```
async function getMoviesFromApi() {
  try {
    let response = await fetch(
      'https://facebook.github.io/react-native/movies.json',
    );
    let responseJson = await response.json();
    return responseJson.movies;
  } catch (error) {
    console.error(error);
  }
}
```

Don't forget to catch any errors that may be thrown by `fetch`, otherwise they will be dropped silently.

```
import React from 'react';
import { FlatList, ActivityIndicator, Text, View } from 'react-native';

export default class FetchExample extends React.Component {

  constructor(props){
    super(props);
    this.state ={ isLoading: true}
  }

  componentDidMount(){
    return fetch('https://facebook.github.io/react-native/movies.json')
      .then((response) => response.json())
      .then((responseJson) => {

        this.setState({
          isLoading: false,
          dataSource: responseJson.movies,
        }, function(){

        }));
      }

      .catch((error) =>{
        console.error(error);
      });
  }

  render(){
    if(this.state.isLoading){
      return(
        <View style={{flex: 1, padding: 20}}>
          <ActivityIndicator/>
        </View>
      )
    }

    return(

```

```

        <View style={{flex: 1, paddingTop:20}}>
          <FlatList
            data={this.state.dataSource}
            renderItem={({item}) => <Text>{item.title}, {item.releaseYear}</Text>}
            keyExtractor={({id}, index) => id}
          />
        </View>
      );
    }
  }
}

```

By default, iOS will block any request that's not encrypted using SSL. If you need to fetch from a cleartext URL (one that begins with `http`) you will first need to [add an App Transport Security exception](#). If you know ahead of time what domains you will need access to, it is more secure to add exceptions just for those domains; if the domains are not known until runtime you can [disable ATS completely](#). Note however that from January 2017, [Apple's App Store review will require reasonable justification for disabling ATS](#).

See [Apple's documentation](#) for more information.

Using Other Networking Libraries

The [XMLHttpRequest API](#) is built in to React Native. This means that you can use third party libraries such as [frisbee](#) or [axios](#) that depend on it, or you can use the XMLHttpRequest API directly if you prefer.

```

var request = new XMLHttpRequest();
request.onreadystatechange = (e) => {
  if (request.readyState !== 4) {
    return;
  }

  if (request.status === 200) {
    console.log('success', request.responseText);
  } else {
    console.warn('error');
  }
};

request.open('GET', 'https://mywebsite.com/endpoint/');
request.send();

```

The security model for XMLHttpRequest is different than on web as there is no concept of [CORS](#) in native apps.

WebSocket Support

React Native also supports [WebSockets](#), a protocol which provides full-duplex communication channels over a single TCP connection.

```

var ws = new WebSocket('ws://host.com/path');

ws.onopen = () => {
  // connection opened
  ws.send('something'); // send a message
};

ws.onmessage = (e) => {

```

```
// a message was received
console.log(e.data);
};

ws.onerror = (e) => {
  // an error occurred
  console.log(e.message);
};

ws.onclose = (e) => {
  // connection closed
  console.log(e.code, e.reason);
};
```

High Five!

If you've gotten here by reading linearly through the tutorial, then you are a pretty impressive human being. Congratulations. Next, you might want to check out [all the cool stuff the community does with React Native](#).

Platform Specific Code

When building a cross-platform app, you'll want to re-use as much code as possible. Scenarios may arise where it makes sense for the code to be different, for example you may want to implement separate visual components for iOS and Android.

React Native provides two ways to easily organize your code and separate it by platform:

- Using the [Platform module](#).
- Using [platform-specific file extensions](#).

Certain components may have properties that work on one platform only. All of these props are annotated with `@platform` and have a small badge next to them on the website.

Platform module

React Native provides a module that detects the platform in which the app is running. You can use the detection logic to implement platform-specific code. Use this option when only small parts of a component are platform-specific.

```
import {Platform, StyleSheet} from 'react-native';

const styles = StyleSheet.create({
  height: Platform.OS === 'ios' ? 200 : 100,
});
```

`Platform.OS` will be `ios` when running on iOS and `android` when running on Android.

There is also a `Platform.select` method available, that given an object containing `Platform.OS` as keys, returns the value for the platform you are currently running on.

```
import {Platform, StyleSheet} from 'react-native';

const styles = StyleSheet.create({
  container: {
    flex: 1,
    ...Platform.select({
      ios: {
        backgroundColor: 'red',
      },
      android: {
        backgroundColor: 'blue',
      },
    }),
  },
});
```

This will result in a container having `flex: 1` on both platforms, a red background color on iOS, and a blue background color on Android.

Since it accepts `any` value, you can also use it to return platform specific component, like below:

```
const Component = Platform.select({
  ios: () => require('ComponentIOS'),
  android: () => require('ComponentAndroid'),
})();

<Component />
```

Detecting the Android version

On Android, the `Platform` module can also be used to detect the version of the Android Platform in which the app is running:

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

if (Platform.Version === 25) {
  console.log('Running on Nougat!');
}
```

Detecting the iOS version

On iOS, the `Version` is a result of `-[UIDevice systemVersion]`, which is a string with the current version of the operating system. An example of the system version is "10.3". For example, to detect the major version number on iOS:

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

const majorVersionIOS = parseInt(Platform.Version, 10);
if (majorVersionIOS <= 9) {
  console.log('Work around a change in behavior');
}
```

Platform-specific extensions

When your platform-specific code is more complex, you should consider splitting the code out into separate files. React Native will detect when a file has a `.ios.` or `.android.` extension and load the relevant platform file when required from other components.

For example, say you have the following files in your project:

```
BigButton.ios.js
BigButton.android.js
```

You can then require the component as follows:

```
const BigButton = require('../BigButton');
```

React Native will automatically pick up the right file based on the running platform.

If you share your React Native code with a website, you might as well use the `BigButton.native.js` so that both iOS and Android will use this file, while the website will use `BigButton.js`.

Navigating Between Screens

Mobile apps are rarely made up of a single screen. Managing the presentation of, and transition between, multiple screens is typically handled by what is known as a navigator.

This guide covers the various navigation components available in React Native. If you are just getting started with navigation, you will probably want to use [React Navigation](#). React Navigation provides an easy to use navigation solution, with the ability to present common stack navigation and tabbed navigation patterns on both iOS and Android.

If you're only targeting iOS, you may want to also check out [NavigatorIOS](#) as a way of providing a native look and feel with minimal configuration, as it provides a wrapper around the native `UINavigationController` class. This component will not work on Android, however.

If you'd like to achieve a native look and feel on both iOS and Android, or you're integrating React Native into an app that already manages navigation natively, the following library provides native navigation on both platforms: [react-native-navigation](#).

React Navigation

The community solution to navigation is a standalone library that allows developers to set up the screens of an app with just a few lines of code.

The first step is to install in your project:

```
npm install --save react-navigation
```

Then you can quickly create an app with a home screen and a profile screen:

```
import {
  createStackNavigator,
} from 'react-navigation';

const App = createStackNavigator({
  Home: { screen: HomeScreen },
  Profile: { screen: ProfileScreen },
});

export default App;
```

Each screen component can set navigation options such as the header title. It can use action creators on the `navigation` prop to link to other screens:

```
class HomeScreen extends React.Component {
  static navigationOptions = {
    title: 'Welcome',
  };
  render() {
    const { navigate } = this.props.navigation;
    return (
      <View>
```

```

        <Button
          title="Go to Jane's profile"
          onPress={() =>
            navigate('Profile', { name: 'Jane' })
          }
        />
      );
    }
}

```

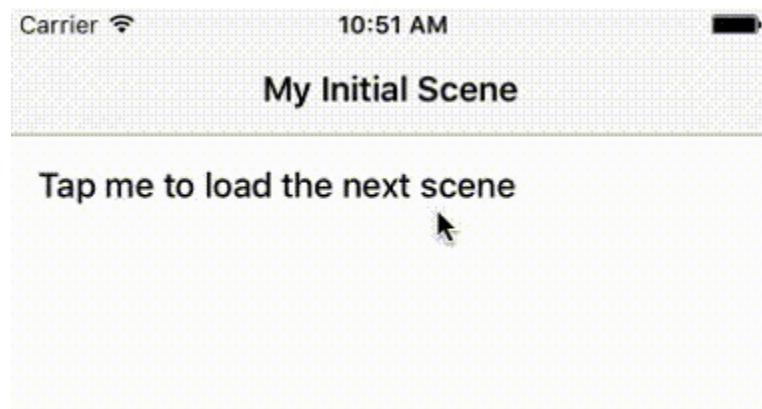
React Navigation routers make it easy to override navigation logic. Because routers can be nested inside each other, developers can override navigation logic for one area of the app without making widespread changes.

The views in React Navigation use native components and the `Animated` library to deliver 60fps animations that are run on the native thread. Plus, the animations and gestures can be easily customized.

For a complete intro to React Navigation, follow the [React Navigation Getting Started Guide](#), or browse other docs such as the [Intro to Navigators](#).

NavigatorIOS

`NavigatorIOS` looks and feels just like `UINavigationController`, because it is actually built on top of it.



```

<NavigatorIOS
  initialRoute={{
    component: MyScene,
    title: 'My Initial Scene',
    passProps: {myProp: 'foo'},
  }}
/>

```

Like other navigation systems, `NavigatorIOS` uses routes to represent screens, with some important differences. The actual component that will be rendered can be specified using the `component` key in the route, and any props that should be passed to this component can be specified in `passProps`. A "navigator" object is automatically passed as a prop to the component, allowing you to call `push` and `pop` as needed.

As `NavigatorIOS` leverages native UIKit navigation, it will automatically render a navigation bar with a back button and title.

```

import React from 'react';
import PropTypes from 'prop-types';

```

```

import {Button, NavigatorIOS, Text, View} from 'react-native';

export default class NavigatorIOSApp extends React.Component {
  render() {
    return (
      <NavigatorIOS
        initialRoute={{
          component: MyScene,
          title: 'My Initial Scene',
          passProps: {index: 1},
        }}
        style={{flex: 1}}
      />
    );
  }
}

class MyScene extends React.Component {
  static propTypes = {
    route: PropTypes.shape({
      title: PropTypes.string.isRequired,
    }),
    navigator: PropTypes.object.isRequired,
  };

  constructor(props, context) {
    super(props, context);
    this._onForward = this._onForward.bind(this);
  }

  _onForward() {
    let nextIndex = ++this.props.index;
    this.props.navigator.push({
      component: MyScene,
      title: 'Scene ' + nextIndex,
      passProps: {index: nextIndex},
    });
  }

  render() {
    return (
      <View>
        <Text>Current Scene: {this.props.title}</Text>
        <Button
          onPress={this._onForward}
          title="Tap me to load the next scene"
        />
      </View>
    );
  }
}

```

Check out the [NavigatorIOS reference docs](#) to learn more about this component.

Images

Static Image Resources

React Native provides a unified way of managing images and other media assets in your iOS and Android apps. To add a static image to your app, place it somewhere in your source code tree and reference it like this:

```
<Image source={require('./my-icon.png')} />
```

The image name is resolved the same way JS modules are resolved. In the example above, the packager will look for `my-icon.png` in the same folder as the component that requires it. Also, if you have `my-icon.ios.png` and `my-icon.android.png`, the packager will pick the correct file for the platform.

You can also use the `@2x` and `@3x` suffixes to provide images for different screen densities. If you have the following file structure:

```
.
├── button.js
└── img
    ├── check.png
    ├── check@2x.png
    └── check@3x.png
```

...and `button.js` code contains:

```
<Image source={require('./img/check.png')} />
```

...the packager will bundle and serve the image corresponding to device's screen density. For example, `check@2x.png`, will be used on an iPhone 7, while `check@3x.png` will be used on an iPhone 7 Plus or a Nexus 5. If there is no image matching the screen density, the closest best option will be selected.

On Windows, you might need to restart the packager if you add new images to your project.

Here are some benefits that you get:

1. Same system on iOS and Android.
2. Images live in the same folder as your JavaScript code. Components are self-contained.
3. No global namespace, i.e. you don't have to worry about name collisions.
4. Only the images that are actually used will be packaged into your app.
5. Adding and changing images doesn't require app recompilation, just refresh the simulator as you normally do.
6. The packager knows the image dimensions, no need to duplicate it in the code.
7. Images can be distributed via `npm` packages.

In order for this to work, the image name in `require` has to be known statically.

```
// GOOD
```

```

<Image source={require('./my-icon.png')} />

// BAD
var icon = this.props.active ? 'my-icon-active' : 'my-icon-inactive';
<Image source={require('./' + icon + '.png')} />

// GOOD
var icon = this.props.active
  ? require('./my-icon-active.png')
  : require('./my-icon-inactive.png');
<Image source={icon} />

```

Note that image sources required this way include size (width, height) info for the Image. If you need to scale the image dynamically (i.e. via flex), you may need to manually set `{ width: undefined, height: undefined }` on the style attribute.

Static Non-Image Resources

The `require` syntax described above can be used to statically include audio, video or document files in your project as well. Most common file types are supported including `.mp3`, `.wav`, `.mp4`, `.mov`, `.html` and `.pdf`. See [packager defaults](#) for the full list.

You can add support for other types by creating a packager config file (see the [packager config file](#) for the full list of configuration options).

A caveat is that videos must use absolute positioning instead of `flexGrow`, since size info is not currently passed for non-image assets. This limitation doesn't occur for videos that are linked directly into Xcode or the Assets folder for Android.

Images From Hybrid App's Resources

If you are building a hybrid app (some UIs in React Native, some UIs in platform code) you can still use images that are already bundled into the app.

For images included via Xcode asset catalogs or in the Android drawable folder, use the image name without the extension:

```
<Image source={{uri: 'app_icon'}} style={{width: 40, height: 40}} />
```

For images in the Android assets folder, use the `asset:/` scheme:

```
<Image source={{uri: 'asset:/app_icon.png'}} style={{width: 40, height: 40}} />
```

These approaches provide no safety checks. It's up to you to guarantee that those images are available in the application. Also you have to specify image dimensions manually.

Network Images

Many of the images you will display in your app will not be available at compile time, or you will want to load some dynamically to keep the binary size down. Unlike with static resources, *you will need to manually specify the dimensions of your image*. It's highly recommended that you use https as well in order to satisfy [App Transport Security](#) requirements on iOS.

```
// GOOD
<Image source={{uri: 'https://facebook.github.io/react/logo-og.png'}}
      style={{width: 400, height: 400}} />

// BAD
<Image source={{uri: 'https://facebook.github.io/react/logo-og.png'}} />
```

Network Requests for Images

If you would like to set such things as the HTTP-Verb, Headers or a Body along with the image request, you may do this by defining these properties on the source object:

```
<Image
  source={{
    uri: 'https://facebook.github.io/react/logo-og.png',
    method: 'POST',
    headers: {
      Pragma: 'no-cache',
    },
    body: 'Your Body goes here',
  }}
  style={{width: 400, height: 400}} />
```

Uri Data Images

Sometimes, you might be getting encoded image data from a REST API call. You can use the `'data:'` uri scheme to use these images. Same as for network resources, *you will need to manually specify the dimensions of your image*.

This is recommended for very small and dynamic images only, like icons in a list from a DB.

```
// include at least width and height!
<Image
  style={{
    width: 51,
    height: 51,
    resizeMode: 'contain',
  }}
  source={{
    uri:
      'data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAADMAAAzCAYAAA6oTAqAAAAEXRFWHTbZ20d2FyZQBwbmdjcnVzaEB1SfMAAABQ
      SURVGje7dSxCQBACARB+2/ab8BEeQNhFi6WSYzYLydDQYGBgYGBgYGBgYGBgZmcvDqYGBgmhivGQYGBgYGBgYGBgYGBgYGBg
      AAAAABJRU5ErkJgg==',
  }}
/>
```

Cache Control (iOS Only)

In some cases you might only want to display an image if it is already in the local cache, i.e. a low resolution placeholder until a higher resolution is available. In other cases you do not care if the image is outdated and are willing to display an outdated image to save bandwidth. The `cache` source property gives you control over how the network layer interacts with the cache.

- `default` : Use the native platforms default strategy.
- `reload` : The data for the URL will be loaded from the originating source. No existing cache data should be used to satisfy a URL load request.
- `force-cache` : The existing cached data will be used to satisfy the request, regardless of its age or expiration date. If there is no existing data in the cache corresponding the request, the data is loaded from the originating source.
- `only-if-cached` : The existing cache data will be used to satisfy a request, regardless of its age or expiration date. If there is no existing data in the cache corresponding to a URL load request, no attempt is made to load the data from the originating source, and the load is considered to have failed.

```
<Image
  source={{
    uri: 'https://facebook.github.io/react/logo-og.png',
    cache: 'only-if-cached',
  }}
  style={{width: 400, height: 400}}
/>
```

Local Filesystem Images

See [CameraRoll](#) for an example of using local resources that are outside of `Images.xcassets`.

Best Camera Roll Image

iOS saves multiple sizes for the same image in your Camera Roll, it is very important to pick the one that's as close as possible for performance reasons. You wouldn't want to use the full quality 3264x2448 image as source when displaying a 200x200 thumbnail. If there's an exact match, React Native will pick it, otherwise it's going to use the first one that's at least 50% bigger in order to avoid blur when resizing from a close size. All of this is done by default so you don't have to worry about writing the tedious (and error prone) code to do it yourself.

Why Not Automatically Size Everything?

In the browser if you don't give a size to an image, the browser is going to render a 0x0 element, download the image, and then render the image based with the correct size. The big issue with this behavior is that your UI is going to jump all around as images load, this makes for a very bad user experience.

In React Native this behavior is intentionally not implemented. It is more work for the developer to know the dimensions (or aspect ratio) of the remote image in advance, but we believe that it leads to a better user experience. Static images loaded from the app bundle via the `require('./my-icon.png')` syntax *can be automatically sized* because their dimensions are available immediately at the time of mounting.

For example, the result of `require('./my-icon.png')` might be:

```
{"__packager_asset":true,"uri":"my-icon.png","width":591,"height":573}
```

Source as an object

In React Native, one interesting decision is that the `src` attribute is named `source` and doesn't take a string but an object with a `uri` attribute.

```
<Image source={{uri: 'something.jpg'}} />
```

On the infrastructure side, the reason is that it allows us to attach metadata to this object. For example if you are using `require('./my-icon.png')`, then we add information about its actual location and size (don't rely on this fact, it might change in the future!). This is also future proofing, for example we may want to support sprites at some point, instead of outputting `{uri: ...}`, we can output `{uri: ..., crop: {left: 10, top: 50, width: 20, height: 40}}` and transparently support spriteing on all the existing call sites.

On the user side, this lets you annotate the object with useful attributes such as the dimension of the image in order to compute the size it's going to be displayed in. Feel free to use it as your data structure to store more information about your image.

Background Image via Nesting

A common feature request from developers familiar with the web is `background-image`. To handle this use case, you can use the `<ImageBackground>` component, which has the same props as `<Image>`, and add whatever children to it you would like to layer on top of it.

You might not want to use `<ImageBackground>` in some cases, since the implementation is very simple. Refer to `<ImageBackground>`'s [documentation](#) for more insight, and create your own custom component when needed.

```
return (
  <ImageBackground source={...} style={{width: '100%', height: '100%'}}>
    <Text>Inside</Text>
  </ImageBackground>
);
```

Note that you must specify some width and height style attributes.

iOS Border Radius Styles

Please note that the following corner specific, border radius style properties are currently ignored by iOS's image component:

- `borderTopLeftRadius`
- `borderTopRightRadius`
- `borderBottomLeftRadius`
- `borderBottomRightRadius`

Off-thread Decoding

Image decoding can take more than a frame-worth of time. This is one of the major sources of frame drops on the web because decoding is done in the main thread. In React Native, image decoding is done in a different thread. In practice, you already need to handle the case when the image is not downloaded yet, so displaying the placeholder for a few more frames while it is decoding does not require any code change.

Animations

Animations are very important to create a great user experience. Stationary objects must overcome inertia as they start moving. Objects in motion have momentum and rarely come to a stop immediately. Animations allow you to convey physically believable motion in your interface.

React Native provides two complementary animation systems: `Animated` for granular and interactive control of specific values, and `LayoutAnimation` for animated global layout transactions.

Animated API

The `Animated` API is designed to make it very easy to concisely express a wide variety of interesting animation and interaction patterns in a very performant way. `Animated` focuses on declarative relationships between inputs and outputs, with configurable transforms in between, and simple `start` / `stop` methods to control time-based animation execution.

`Animated` exports four animatable component types: `View`, `Text`, `Image`, and `ScrollView`, but you can also create your own using `Animated.createAnimatedComponent()`.

For example, a container view that fades in when it is mounted may look like this:

```
import React from 'react';
import { Animated, Text, View } from 'react-native';

class FadeInView extends React.Component {
  state = {
    fadeAnim: new Animated.Value(0), // Initial value for opacity: 0
  }

  componentDidMount() {
    Animated.timing(
      this.state.fadeAnim, // The animated value to drive
      {
        toValue: 1, // Animate to opacity: 1 (opaque)
        duration: 10000, // Make it take a while
      }
    ).start(); // Starts the animation
  }

  render() {
    let { fadeAnim } = this.state;

    return (
      <Animated.View // Special animatable View
        style={{
          ...this.props.style,
          opacity: fadeAnim, // Bind opacity to animated value
        }}
      >
        {this.props.children}
      </Animated.View>
    );
  }
}

// You can then use your `FadeInView` in place of a `View` in your components:
```

```

export default class App extends React.Component {
  render() {
    return (
      <View style={{flex: 1, alignItems: 'center', justifyContent: 'center'}}>
        <FadeInView style={{width: 250, height: 50, backgroundColor: 'powderblue'}}>
          <Text style={{fontSize: 28, textAlign: 'center', margin: 10}}>Fading in</Text>
        </FadeInView>
      </View>
    )
  }
}

```

Let's break down what's happening here. In the `FadeInView` constructor, a new `Animated.Value` called `fadeAnim` is initialized as part of `state`. The opacity property on the `view` is mapped to this animated value. Behind the scenes, the numeric value is extracted and used to set opacity.

When the component mounts, the opacity is set to 0. Then, an easing animation is started on the `fadeAnim` animated value, which will update all of its dependent mappings (in this case, just the opacity) on each frame as the value animates to the final value of 1.

This is done in an optimized way that is faster than calling `setState` and re-rendering.

Because the entire configuration is declarative, we will be able to implement further optimizations that serialize the configuration and runs the animation on a high-priority thread.

Configuring animations

Animations are heavily configurable. Custom and predefined easing functions, delays, durations, decay factors, spring constants, and more can all be tweaked depending on the type of animation.

`Animated` provides several animation types, the most commonly used one being `Animated.timing()`. It supports animating a value over time using one of various predefined easing functions, or you can use your own. Easing functions are typically used in animation to convey gradual acceleration and deceleration of objects.

By default, `timing` will use a `easeInOut` curve that conveys gradual acceleration to full speed and concludes by gradually decelerating to a stop. You can specify a different easing function by passing a `easing` parameter. Custom `duration` or even a `delay` before the animation starts is also supported.

For example, if we want to create a 2-second long animation of an object that slightly backs up before moving to its final position:

```

Animated.timing(this.state.xPosition, {
  toValue: 100,
  easing: Easing.back(),
  duration: 2000,
}).start();

```

Take a look at the [Configuring animations](#) section of the `Animated` API reference to learn more about all the config parameters supported by the built-in animations.

Composing animations

Animations can be combined and played in sequence or in parallel. Sequential animations can play immediately after the previous animation has finished, or they can start after a specified delay. The `Animated` API provides several methods, such as `sequence()` and `delay()`, each of which simply take an array of animations to execute

and automatically calls `start()` / `stop()` as needed.

For example, the following animation coasts to a stop, then it springs back while twirling in parallel:

```
Animated.sequence([
  // decay, then spring to start and twirl
  Animated.decay(position, {
    // coast to a stop
    velocity: {x: gestureState.vx, y: gestureState.vy}, // velocity from gesture release
    deceleration: 0.997,
  }),
  Animated.parallel([
    // after decay, in parallel:
    Animated.spring(position, {
      toValue: {x: 0, y: 0}, // return to start
    }),
    Animated.timing(twirl, {
      // and twirl
      toValue: 360,
    }),
  ]),
]).start(); // start the sequence group
```

If one animation is stopped or interrupted, then all other animations in the group are also stopped.

`Animated.parallel` has a `stopTogether` option that can be set to `false` to disable this.

You can find a full list of composition methods in the [Composing animations](#) section of the `Animated` API reference.

Combining animated values

You can [combine two animated values](#) via addition, multiplication, division, or modulo to make a new animated value.

There are some cases where an animated value needs to invert another animated value for calculation. An example is inverting a scale (2x --> 0.5x):

```
const a = new Animated.Value(1);
const b = Animated.divide(1, a);

Animated.spring(a, {
  toValue: 2,
}).start();
```

Interpolation

Each property can be run through an interpolation first. An interpolation maps input ranges to output ranges, typically using a linear interpolation but also supports easing functions. By default, it will extrapolate the curve beyond the ranges given, but you can also have it clamp the output value.

A simple mapping to convert a 0-1 range to a 0-100 range would be:

```
value.interpolate({
  inputRange: [0, 1],
  outputRange: [0, 100],
```

```
});
```

For example, you may want to think about your `Animated.Value` as going from 0 to 1, but animate the position from 150px to 0px and the opacity from 0 to 1. This can easily be done by modifying `style` from the example above like so:

```
style={{
  opacity: this.state.fadeAnim, // Binds directly
  transform: [
    translateY: this.state.fadeAnim.interpolate({
      inputRange: [0, 1],
      outputRange: [150, 0] // 0 : 150, 0.5 : 75, 1 : 0
    })
  ]
}}
```

`interpolate()` supports multiple range segments as well, which is handy for defining dead zones and other handy tricks. For example, to get a negation relationship at -300 that goes to 0 at -100, then back up to 1 at 0, and then back down to zero at 100 followed by a dead-zone that remains at 0 for everything beyond that, you could do:

```
value.interpolate({
  inputRange: [-300, -100, 0, 100, 101],
  outputRange: [300, 0, 1, 0, 0],
});
```

Which would map like so:

Input	Output
-400	450
-300	300
-200	150
-100	0
-50	0.5
0	1
50	0.5
100	0
101	0
200	0

`interpolate()` also supports mapping to strings, allowing you to animate colors as well as values with units. For example, if you wanted to animate a rotation you could do:

```
value.interpolate({
  inputRange: [0, 360],
  outputRange: ['0deg', '360deg'],
});
```

`interpolate()` also supports arbitrary easing functions, many of which are already implemented in the [Easing](#) module. `interpolate()` also has configurable behavior for extrapolating the `outputRange`. You can set the extrapolation by setting the `extrapolate`, `extrapolateLeft`, or `extrapolateRight` options. The default value is `extend` but you can use `clamp` to prevent the output value from exceeding `outputRange`.

Tracking dynamic values

Animated values can also track other values. Just set the `toValue` of an animation to another animated value instead of a plain number. For example, a "Chat Heads" animation like the one used by Messenger on Android could be implemented with a `spring()` pinned on another animated value, or with `timing()` and a `duration` of 0 for rigid tracking. They can also be composed with interpolations:

```
Animated.spring(follower, {toValue: leader}).start();
Animated.timing(opacity, {
  toValue: pan.x.interpolate({
    inputRange: [0, 300],
    outputRange: [1, 0],
  }),
}.start();
```

The `leader` and `follower` animated values would be implemented using `Animated.ValueXY()`. `ValueXY` is a handy way to deal with 2D interactions, such as panning or dragging. It is a simple wrapper that basically contains two `Animated.Value` instances and some helper functions that call through to them, making `ValueXY` a drop-in replacement for `Value` in many cases. It allows us to track both x and y values in the example above.

Tracking gestures

Gestures, like panning or scrolling, and other events can map directly to animated values using `Animated.event`. This is done with a structured map syntax so that values can be extracted from complex event objects. The first level is an array to allow mapping across multiple args, and that array contains nested objects.

For example, when working with horizontal scrolling gestures, you would do the following in order to map

```
event.nativeEvent.contentOffset.x to scrollX (an Animated.Value):
```

```
onScroll={Animated.event(
  // scrollX = e.nativeEvent.contentOffset.x
  [{ nativeEvent: {
    contentOffset: {
      x: scrollX
    }
  }]
)}
```

When using `PanResponder`, you could use the following code to extract the x and y positions from `gestureState.dx` and `gestureState.dy`. We use a `null` in the first position of the array, as we are only interested in the second argument passed to the `PanResponder` handler, which is the `gestureState`.

```
onPanResponderMove={Animated.event(
  [null, // ignore the native event
  // extract dx and dy from gestureState
  // like 'pan.x = gestureState.dx, pan.y = gestureState.dy'
  {dx: pan.x, dy: pan.y}
])}
```

Responding to the current animation value

You may notice that there is no obvious way to read the current value while animating. This is because the value may only be known in the native runtime due to optimizations. If you need to run JavaScript in response to the current value, there are two approaches:

- `spring.stopAnimation(callback)` will stop the animation and invoke `callback` with the final value. This is useful when making gesture transitions.
- `spring.addListener(callback)` will invoke `callback` asynchronously while the animation is running, providing a recent value. This is useful for triggering state changes, for example snapping a bobble to a new option as the user drags it closer, because these larger state changes are less sensitive to a few frames of lag compared to continuous gestures like panning which need to run at 60 fps.

`Animated` is designed to be fully serializable so that animations can be run in a high performance way, independent of the normal JavaScript event loop. This does influence the API, so keep that in mind when it seems a little trickier to do something compared to a fully synchronous system. Check out `Animated.Value.addListener` as a way to work around some of these limitations, but use it sparingly since it might have performance implications in the future.

Using the native driver

The `Animated` API is designed to be serializable. By using the [native driver](#), we send everything about the animation to native before starting the animation, allowing native code to perform the animation on the UI thread without having to go through the bridge on every frame. Once the animation has started, the JS thread can be blocked without affecting the animation.

Using the native driver for normal animations is quite simple. Just add `useNativeDriver: true` to the animation config when starting it.

```
Animated.timing(this.state.animatedValue, {
  toValue: 1,
  duration: 500,
  useNativeDriver: true, // <-- Add this
}).start();
```

Animated values are only compatible with one driver so if you use native driver when starting an animation on a value, make sure every animation on that value also uses the native driver.

The native driver also works with `Animated.event`. This is especially useful for animations that follow the scroll position as without the native driver, the animation will always run a frame behind the gesture due to the async nature of React Native.

```
<AnimatedScrollView // <-- Use the Animated ScrollView wrapper
scrollEventThrottle={1} // <-- Use 1 here to make sure no events are ever missed
onScroll={Animated.event(
  [
    {
      nativeEvent: {
        contentOffset: {y: this.state.animatedValue},
      },
    ],
  ],
  {useNativeDriver: true}, // <-- Add this
)}>
{content}
</AnimatedScrollView>
```

You can see the native driver in action by running the [RNTester app](#), then loading the Native Animated Example. You can also take a look at the [source code](#) to learn how these examples were produced.

Caveats

Not everything you can do with `Animated` is currently supported by the native driver. The main limitation is that you can only animate non-layout properties: things like `transform` and `opacity` will work, but flexbox and position properties will not. When using `Animated.event`, it will only work with direct events and not bubbling events. This means it does not work with `PanResponder` but does work with things like `ScrollView#onScroll`.

When an animation is running, it can prevent `VirtualizedList` components from rendering more rows. If you need to run a long or looping animation while the user is scrolling through a list, you can use `isInteraction: false` in your animation's config to prevent this issue.

Bear in mind

While using transform styles such as `rotateY`, `rotateX`, and others ensure the transform style `perspective` is in place. At this time some animations may not render on Android without it. Example below.

```
<Animated.View
  style={{
    transform: [
      {scale: this.state.scale},
      {rotateY: this.state.rotateY},
      {perspective: 1000}, // without this line this Animation will not render on Android while working fine on iOS
    ],
  }}
/>
```

Additional examples

The RNTester app has various examples of `Animated` in use:

- [AnimatedGratuitousApp](#)
- [NativeAnimationsExample](#)

LayoutAnimation API

`LayoutAnimation` allows you to globally configure `create` and `update` animations that will be used for all views in the next render/layout cycle. This is useful for doing flexbox layout updates without bothering to measure or calculate specific properties in order to animate them directly, and is especially useful when layout changes may affect ancestors, for example a "see more" expansion that also increases the size of the parent and pushes down the row below which would otherwise require explicit coordination between the components in order to animate them all in sync.

Note that although `LayoutAnimation` is very powerful and can be quite useful, it provides much less control than `Animated` and other animation libraries, so you may need to use another approach if you can't get `LayoutAnimation` to do what you want.

Note that in order to get this to work on **Android** you need to set the following flags via `UIManager`:

```
UIManager.setLayoutAnimationEnabledExperimental &&
  UIManager.setLayoutAnimationEnabledExperimental(true);
```

```
import React from 'react';
import {
  NativeModules,
  LayoutAnimation,
  Text,
  TouchableOpacity,
  StyleSheet,
  View,
} from 'react-native';

const { UIManager } = NativeModules;

UIManager.setLayoutAnimationEnabledExperimental &&
  UIManager.setLayoutAnimationEnabledExperimental(true);

export default class App extends React.Component {
  state = {
    w: 100,
    h: 100,
  };

  _onPress = () => {
    // Animate the update
    LayoutAnimation.spring();
    this.setState({w: this.state.w + 15, h: this.state.h + 15})
  }

  render() {
    return (
      <View style={styles.container}>
        <View style={[styles.box, {width: this.state.w, height: this.state.h}]} />
        <TouchableOpacity onPress={this._onPress}>
          <View style={styles.button}>
            <Text style={styles.buttonText}>Press me!</Text>
          </View>
        </TouchableOpacity>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  box: {
    width: 200,
    height: 200,
    backgroundColor: 'red',
  },
  button: {
    backgroundColor: 'black',
    paddingHorizontal: 20,
    paddingVertical: 15,
    marginTop: 15,
  },
  buttonText: {
```

```
    color: '#fff',
    fontWeight: 'bold',
  },
});
```

This example uses a preset value, you can customize the animations as you need, see [LayoutAnimation.js](#) for more information.

Additional notes

requestAnimationFrame

`requestAnimationFrame` is a polyfill from the browser that you might be familiar with. It accepts a function as its only argument and calls that function before the next repaint. It is an essential building block for animations that underlies all of the JavaScript-based animation APIs. In general, you shouldn't need to call this yourself - the animation APIs will manage frame updates for you.

setNativeProps

As mentioned [in the Direct Manipulation section](#), `setNativeProps` allows us to modify properties of native-backed components (components that are actually backed by native views, unlike composite components) directly, without having to `setState` and re-render the component hierarchy.

We could use this in the Rebound example to update the scale - this might be helpful if the component that we are updating is deeply nested and hasn't been optimized with `shouldComponentUpdate`.

If you find your animations with dropping frames (performing below 60 frames per second), look into using `setNativeProps` OR `shouldComponentUpdate` to optimize them. Or you could run the animations on the UI thread rather than the JavaScript thread [with the useNativeDriver option](#). You may also want to defer any computationally intensive work until after animations are complete, using the [InteractionManager](#). You can monitor the frame rate by using the In-App Developer Menu "FPS Monitor" tool.

Accessibility

Native App Accessibility (iOS and Android)

Both iOS and Android provide APIs for making apps accessible to people with disabilities. In addition, both platforms provide bundled assistive technologies, like the screen readers VoiceOver (iOS) and TalkBack (Android) for the visually impaired. Similarly, in React Native we have included APIs designed to provide developers with support for making apps more accessible. Take note, iOS and Android differ slightly in their approaches, and thus the React Native implementations may vary by platform.

In addition to this documentation, you might find [this blog post](#) about React Native accessibility to be useful.

Making Apps Accessible

Accessibility properties

accessible (iOS, Android)

When `true`, indicates that the view is an accessibility element. When a view is an accessibility element, it groups its children into a single selectable component. By default, all touchable elements are accessible.

On Android, `accessible={true}` property for a react-native View will be translated into native `focusable={true}`.

```
<View accessible={true}>
  <Text>text one</Text>
  <Text>text two</Text>
</View>
```

In the above example, we can't get accessibility focus separately on 'text one' and 'text two'. Instead we get focus on a parent view with 'accessible' property.

accessibilityLabel (iOS, Android)

When a view is marked as accessible, it is a good practice to set an accessibilityLabel on the view, so that people who use VoiceOver know what element they have selected. VoiceOver will read this string when a user selects the associated element.

To use, set the `accessibilityLabel` property to a custom string on your View, Text or Touchable:

```
<TouchableOpacity
  accessible={true}
  accessibilityLabel="Tap me!"
  onPress={this._onPress}>
  <View style={styles.button}>
    <Text style={styles.buttonText}>Press me!</Text>
  </View>
</TouchableOpacity>
```

In the above example, the `accessibilityLabel` on the `TouchableOpacity` element would default to "Press me!". The label is constructed by concatenating all `Text` node children separated by spaces.

accessibilityHint (iOS, Android)

An accessibility hint helps users understand what will happen when they perform an action on the accessibility element when that result is not obvious from the accessibility label.

To use, set the `accessibilityHint` property to a custom string on your View, Text or Touchable:

```
<TouchableOpacity
  accessible={true}
  accessibilityLabel="Go back"
  accessibilityHint="Navigates to the previous screen"
  onPress={this._onPress}>
  <View style={styles.button}>
    <Text style={styles.buttonText}>Back</Text>
  </View>
</TouchableOpacity>
```

iOS In the above example, VoiceOver will read the hint after the label, if the user has hints enabled in the device's VoiceOver settings. Read more about guidelines for `accessibilityHint` in the [iOS Developer Docs](#)

Android In the above example, Talkback will read the hint after the label. At this time, hints cannot be turned off on Android.

accessibilityIgnoresInvertColors(iOS)

Inverting screen colors is an Accessibility feature that makes the iPhone and iPad easier on the eyes for some people with a sensitivity to brightness, easier to distinguish for some people with color blindness, and easier to make out for some people with low vision. However, sometimes you have views such as photos that you don't want to be inverted. In this case, you can set this property to be false so that these specific views won't have their colors inverted.

accessibilityRole (iOS, Android)

Note: Accessibility Role and Accessibility States are meant to be a cross-platform solution to replace `accessibilityTraits` and `accessibilityComponentType`, which will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead of `accessibilityTraits` and `accessibilityComponentType`.

Accessibility Role tells a person using either VoiceOver on iOS or TalkBack on Android the type of element that is focused on. To use, set the `accessibilityRole` property to one of the following strings:

- **none** Used when the element has no role.
- **button** Used when the element should be treated as a button.
- **link** Used when the element should be treated as a link.
- **search** Used when the text field element should also be treated as a search field.
- **image** Used when the element should be treated as an image. Can be combined with button or link, for example.
- **keyboardkey** Used when the element acts as a keyboard key.
- **text** Used when the element should be treated as static text that cannot change.
- **adjustable** Used when an element can be "adjusted" (e.g. a slider).

- **imagebutton** Used when the element should be treated as a button and is also an image.
- **header** Used when an element acts as a header for a content section (e.g. the title of a navigation bar).
- **summary** Used when an element can be used to provide a quick summary of current conditions in the app when the app first launches.

accessibilityStates (iOS, Android)

Note: > `accessibilityRole` and `accessibilityStates` are meant to be a cross-platform solution to replace `accessibilityTraits` and `accessibilityComponentType`, which will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead of `accessibilityTraits` and `accessibilityComponentType`.

Accessibility State tells a person using either VoiceOver on iOS or TalkBack on Android the state of the element currently focused on. The state of the element can be set either to `selected` or `disabled` or both:

- **selected** Used when the element is in a selected state. For example, a button is selected.
- **disabled** Used when the element is disabled and cannot be interacted with.

To use, set the `accessibilityStates` to an array containing either `selected`, `disabled`, or both.

accessibilityTraits (iOS)

Note: `accessibilityTraits` will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead of `accessibilityTraits` and `accessibilityComponentType`.

Accessibility traits tell a person using VoiceOver what kind of element they have selected. Is this element a label? A button? A header? These questions are answered by `accessibilityTraits`.

To use, set the `accessibilityTraits` property to one of (or an array of) accessibility trait strings:

- **none** Used when the element has no traits.
- **button** Used when the element should be treated as a button.
- **link** Used when the element should be treated as a link.
- **header** Used when an element acts as a header for a content section (e.g. the title of a navigation bar).
- **search** Used when the text field element should also be treated as a search field.
- **image** Used when the element should be treated as an image. Can be combined with button or link, for example.
- **selected** Used when the element is selected. For example, a selected row in a table or a selected button within a segmented control.
- **plays** Used when the element plays its own sound when activated.
- **key** Used when the element acts as a keyboard key.
- **text** Used when the element should be treated as static text that cannot change.
- **summary** Used when an element can be used to provide a quick summary of current conditions in the app when the app first launches. For example, when Weather first launches, the element with today's weather conditions is marked with this trait.
- **disabled** Used when the control is not enabled and does not respond to user input.
- **frequentUpdates** Used when the element frequently updates its label or value, but too often to send notifications. Allows an accessibility client to poll for changes. A stopwatch would be an example.
- **startsMedia** Used when activating an element starts a media session (e.g. playing a movie, recording audio) that should not be interrupted by output from an assistive technology, like VoiceOver.
- **adjustable** Used when an element can be "adjusted" (e.g. a slider).
- **allowsDirectInteraction** Used when an element allows direct touch interaction for VoiceOver users (for example, a view representing a piano keyboard).

- **pageTurn** Informs VoiceOver that it should scroll to the next page when it finishes reading the contents of the element.

accessibilityViewIsModal (iOS)

A Boolean value indicating whether VoiceOver should ignore the elements within views that are siblings of the receiver.

For example, in a window that contains sibling views `A` and `B`, setting `accessibilityViewIsModal` to `true` on view `B` causes VoiceOver to ignore the elements in the view `A`. On the other hand, if view `B` contains a child view `C` and you set `accessibilityViewIsModal` to `true` on view `C`, VoiceOver does not ignore the elements in view `A`.

accessibilityElementsHidden (iOS)

A Boolean value indicating whether the accessibility elements contained within this accessibility element are hidden.

For example, in a window that contains sibling views `A` and `B`, setting `accessibilityElementsHidden` to `true` on view `B` causes VoiceOver to ignore the elements in the view `B`. This is similar to the Android property `importantForAccessibility="no-hide-descendants"`.

onAccessibilityTap (iOS)

Use this property to assign a custom function to be called when someone activates an accessible element by double tapping on it while it's selected.

onMagicTap (iOS)

Assign this property to a custom function which will be called when someone performs the "magic tap" gesture, which is a double-tap with two fingers. A magic tap function should perform the most relevant action a user could take on a component. In the Phone app on iPhone, a magic tap answers a phone call, or ends the current one. If the selected element does not have an `onMagicTap` function, the system will traverse up the view hierarchy until it finds a view that does.

accessibilityComponentType (Android)

Note: > `accessibilityComponentType` will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead of `accessibilityTraits` and `accessibilityComponentType`.

In some cases, we also want to alert the end user of the type of selected component (i.e., that it is a "button"). If we were using native buttons, this would work automatically. Since we are using javascript, we need to provide a bit more context for TalkBack. To do so, you must specify the 'accessibilityComponentType' property for any UI component. We support 'none', 'button', 'radiobutton_checked' and 'radiobutton_unchecked'.

```
<TouchableWithoutFeedback accessibilityComponentType="button"
  onPress={this._onPress}>
  <View style={styles.button}>
    <Text style={styles.buttonText}>Press me!</Text>
  </View>
</TouchableWithoutFeedback>
```

In the above example, the TouchableWithoutFeedback is being announced by TalkBack as a native Button.

accessibilityLiveRegion (Android)

When components dynamically change, we want TalkBack to alert the end user. This is made possible by the 'accessibilityLiveRegion' property. It can be set to 'none', 'polite' and 'assertive':

- **none** Accessibility services should not announce changes to this view.
- **polite** Accessibility services should announce changes to this view.
- **assertive** Accessibility services should interrupt ongoing speech to immediately announce changes to this view.

```
<TouchableWithoutFeedback onPress={this._addOne}>
  <View style={styles.embedded}>
    <Text>Click me</Text>
  </View>
</TouchableWithoutFeedback>
<Text accessibilityLiveRegion="polite">
  Clicked {this.state.count} times
</Text>
```

In the above example method _addOne changes the state.count variable. As soon as an end user clicks the TouchableWithoutFeedback, TalkBack reads text in the Text view because of its 'accessibilityLiveRegion="polite"' property.

importantForAccessibility (Android)

In the case of two overlapping UI components with the same parent, default accessibility focus can have unpredictable behavior. The 'importantForAccessibility' property will resolve this by controlling if a view fires accessibility events and if it is reported to accessibility services. It can be set to 'auto', 'yes', 'no' and 'no-hide-descendants' (the last value will force accessibility services to ignore the component and all of its children).

```
<View style={styles.container}>
  <View style={{position: 'absolute', left: 10, top: 10, right: 10, height: 100,
    backgroundColor: 'green'}} importantForAccessibility="yes">
    <Text> First layout </Text>
  </View>
  <View style={{position: 'absolute', left: 10, top: 10, right: 10, height: 100,
    backgroundColor: 'yellow'}} importantForAccessibility="no-hide-descendants">
    <Text> Second layout </Text>
  </View>
</View>
```

In the above example, the yellow layout and its descendants are completely invisible to TalkBack and all other accessibility services. So we can easily use overlapping views with the same parent without confusing TalkBack.

Checking if a Screen Reader is Enabled

The `AccessibilityInfo` API allows you to determine whether or not a screen reader is currently active. See the [AccessibilityInfo documentation](#) for details.

Sending Accessibility Events (Android)

Sometimes it is useful to trigger an accessibility event on a UI component (i.e. when a custom view appears on a screen or a custom radio button has been selected). Native UIManager module exposes a method 'sendAccessibilityEvent' for this purpose. It takes two arguments: view tag and a type of an event.

```
import { UIManager, findNodeHandle } from 'react-native';

_onPress: function() {
  const radioButton = this.state.radioButton === 'radiobutton_checked' ?
    'radiobutton_unchecked' : 'radiobutton_checked'

  this.setState({
    radioButton: radioButton
  });

  if (radioButton === 'radiobutton_checked') {
    UIManager.sendAccessibilityEvent(
      findNodeHandle(this),
      UIManager.AccessibilityEventTypes.typeViewClicked);
  }
}

<CustomRadioButton
  accessibilityComponentType={this.state.radioButton}
  onPress={this._onPress}/>
```

In the above example we've created a custom radio button that now behaves like a native one. More specifically, TalkBack now correctly announces changes to the radio button selection.

Testing VoiceOver Support (iOS)

To enable VoiceOver, go to the Settings app on your iOS device. Tap General, then Accessibility. There you will find many tools that people use to make their devices more usable, such as bolder text, increased contrast, and VoiceOver.

To enable VoiceOver, tap on VoiceOver under "Vision" and toggle the switch that appears at the top.

At the very bottom of the Accessibility settings, there is an "Accessibility Shortcut". You can use this to toggle VoiceOver by triple clicking the Home button.

Timers

Timers are an important part of an application and React Native implements the [browser timers](#).

Timers

- `setTimeout`, `clearTimeout`
- `setInterval`, `clearInterval`
- `setImmediate`, `clearImmediate`
- `requestAnimationFrame`, `cancelAnimationFrame`

`requestAnimationFrame(fn)` is not the same as `setTimeout(fn, 0)` - the former will fire after all the frame has flushed, whereas the latter will fire as quickly as possible (over 1000x per second on a iPhone 5S).

`setImmediate` is executed at the end of the current JavaScript execution block, right before sending the batched response back to native. Note that if you call `setImmediate` within a `setImmediate` callback, it will be executed right away, it won't yield back to native in between.

The `Promise` implementation uses `setImmediate` as its asynchronicity primitive.

InteractionManager

One reason why well-built native apps feel so smooth is by avoiding expensive operations during interactions and animations. In React Native, we currently have a limitation that there is only a single JS execution thread, but you can use `InteractionManager` to make sure long-running work is scheduled to start after any interactions/animations have completed.

Applications can schedule tasks to run after interactions with the following:

```
InteractionManager.runAfterInteractions(() => {
  // ...long-running synchronous task...
});
```

Compare this to other scheduling alternatives:

- `requestAnimationFrame()`: for code that animates a view over time.
- `setImmediate/setTimeout/setInterval()`: run code later, note this may delay animations.
- `runAfterInteractions()`: run code later, without delaying active animations.

The touch handling system considers one or more active touches to be an 'interaction' and will delay `runAfterInteractions()` callbacks until all touches have ended or been cancelled.

`InteractionManager` also allows applications to register animations by creating an interaction 'handle' on animation start, and clearing it upon completion:

```
var handle = InteractionManager.createInteractionHandle();
// run animation... (`runAfterInteractions` tasks are queued)
// later, on animation completion:
InteractionManager.clearInteractionHandle(handle);
```

```
// queued tasks run if all handles were cleared
```

TimerMixin

We found out that the primary cause of fatsals in apps created with React Native was due to timers firing after a component was unmounted. To solve this recurring issue, we introduced `TimerMixin`. If you include `TimerMixin`, then you can replace your calls to `setTimeout(fn, 500)` with `this.setTimeout(fn, 500)` (just prepend `this.`) and everything will be properly cleaned up for you when the component unmounts.

This library does not ship with React Native - in order to use it on your project, you will need to install it with `npm i react-timer-mixin --save` from your project directory.

```
import TimerMixin from 'react-timer-mixin';

var Component = createReactClass({
  mixins: [TimerMixin],
  componentDidMount: function() {
    this.setTimeout(() => {
      console.log('I do not leak!');
    }, 500);
  },
});
```

This will eliminate a lot of hard work tracking down bugs, such as crashes caused by timeouts firing after a component has been unmounted.

Keep in mind that if you use ES6 classes for your React components [there is no built-in API for mixins](#). To use `TimerMixin` with ES6 classes, we recommend [react-mixin](#).

Performance

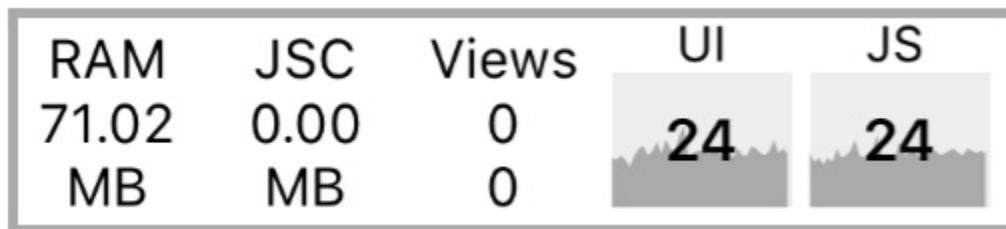
A compelling reason for using React Native instead of WebView-based tools is to achieve 60 frames per second and a native look and feel to your apps. Where possible, we would like for React Native to do the right thing and help you to focus on your app instead of performance optimization, but there are areas where we're not quite there yet, and others where React Native (similar to writing native code directly) cannot possibly determine the best way to optimize for you and so manual intervention will be necessary. We try our best to deliver buttery-smooth UI performance by default, but sometimes that just isn't possible.

This guide is intended to teach you some basics to help you to [troubleshoot performance issues](#), as well as discuss [common sources of problems and their suggested solutions](#).

What you need to know about frames

Your grandparents' generation called movies "[moving pictures](#)" for a reason: realistic motion in video is an illusion created by quickly changing static images at a consistent speed. We refer to each of these images as frames. The number of frames that is displayed each second has a direct impact on how smooth and ultimately life-like a video (or user interface) seems to be. iOS devices display 60 frames per second, which gives you and the UI system about 16.67ms to do all of the work needed to generate the static image (frame) that the user will see on the screen for that interval. If you are unable to do the work necessary to generate that frame within the allotted 16.67ms, then you will "drop a frame" and the UI will appear unresponsive.

Now to confuse the matter a little bit, open up the developer menu in your app and toggle [Show Perf Monitor](#). You will notice that there are two different frame rates.



JS frame rate (JavaScript thread)

For most React Native applications, your business logic will run on the JavaScript thread. This is where your React application lives, API calls are made, touch events are processed, etc... Updates to native-backed views are batched and sent over to the native side at the end of each iteration of the event loop, before the frame deadline (if all goes well). If the JavaScript thread is unresponsive for a frame, it will be considered a dropped frame. For example, if you were to call `this.setState` on the root component of a complex application and it resulted in re-rendering computationally expensive component subtrees, it's conceivable that this might take 200ms and result in 12 frames being dropped. Any animations controlled by JavaScript would appear to freeze during that time. If anything takes longer than 100ms, the user will feel it.

This often happens during `Navigator` transitions: when you push a new route, the JavaScript thread needs to render all of the components necessary for the scene in order to send over the proper commands to the native side to create the backing views. It's common for the work being done here to take a few frames and cause [jank](#).

because the transition is controlled by the JavaScript thread. Sometimes components will do additional work on `componentDidMount`, which might result in a second stutter in the transition.

Another example is responding to touches: if you are doing work across multiple frames on the JavaScript thread, you might notice a delay in responding to `TouchableOpacity`, for example. This is because the JavaScript thread is busy and cannot process the raw touch events sent over from the main thread. As a result, `TouchableOpacity` cannot react to the touch events and command the native view to adjust its opacity.

UI frame rate (main thread)

Many people have noticed that performance of `NavigatorIOS` is better out of the box than `Navigator`. The reason for this is that the animations for the transitions are done entirely on the main thread, and so they are not interrupted by frame drops on the JavaScript thread.

Similarly, you can happily scroll up and down through a `ScrollView` when the JavaScript thread is locked up because the `ScrollView` lives on the main thread. The scroll events are dispatched to the JS thread, but their receipt is not necessary for the scroll to occur.

Common sources of performance problems

Running in development mode (`dev=true`)

JavaScript thread performance suffers greatly when running in dev mode. This is unavoidable: a lot more work needs to be done at runtime to provide you with good warnings and error messages, such as validating `propTypes` and various other assertions. Always make sure to test performance in [release builds](#).

Using `console.log` statements

When running a bundled app, these statements can cause a big bottleneck in the JavaScript thread. This includes calls from debugging libraries such as [redux-logger](#), so make sure to remove them before bundling. You can also use this [babel plugin](#) that removes all the `console.*` calls. You need to install it first with `npm i babel-plugin-transform-remove-console --save-dev`, and then edit the `.babelrc` file under your project directory like this:

```
{  
  "env": {  
    "production": {  
      "plugins": ["transform-remove-console"]  
    }  
  }  
}
```

This will automatically remove all `console.*` calls in the release (production) versions of your project.

`ListView` initial rendering is too slow or scroll performance is bad for large lists

Use the new `FlatList` or `SectionList` component instead. Besides simplifying the API, the new list components also have significant performance enhancements, the main one being nearly constant memory usage for any number of rows.

If your `FlatList` is rendering slow, be sure that you've implemented `getLayout` to optimize rendering speed by skipping measurement of the rendered items.

JS FPS plunges when re-rendering a view that hardly changes

If you are using a `ListView`, you must provide a `rowHasChanged` function that can reduce a lot of work by quickly determining whether or not a row needs to be re-rendered. If you are using immutable data structures, this would be as simple as a reference equality check.

Similarly, you can implement `shouldComponentUpdate` and indicate the exact conditions under which you would like the component to re-render. If you write pure components (where the return value of the render function is entirely dependent on props and state), you can leverage `PureComponent` to do this for you. Once again, immutable data structures are useful to keep this fast -- if you have to do a deep comparison of a large list of objects, it may be that re-rendering your entire component would be quicker, and it would certainly require less code.

Dropping JS thread FPS because of doing a lot of work on the JavaScript thread at the same time

"Slow Navigator transitions" is the most common manifestation of this, but there are other times this can happen. Using `InteractionManager` can be a good approach, but if the user experience cost is too high to delay work during an animation, then you might want to consider `LayoutAnimation`.

The `Animated` API currently calculates each keyframe on-demand on the JavaScript thread unless you set `useNativeDriver: true`, while `LayoutAnimation` leverages Core Animation and is unaffected by JS thread and main thread frame drops.

One case where I have used this is for animating in a modal (sliding down from top and fading in a translucent overlay) while initializing and perhaps receiving responses for several network requests, rendering the contents of the modal, and updating the view where the modal was opened from. See the Animations guide for more information about how to use `LayoutAnimation`.

Caveats:

- `LayoutAnimation` only works for fire-and-forget animations ("static" animations) -- if it must be interruptible, you will need to use `Animated`.

Moving a view on the screen (scrolling, translating, rotating) drops UI thread FPS

This is especially true when you have text with a transparent background positioned on top of an image, or any other situation where alpha compositing would be required to re-draw the view on each frame. You will find that enabling `shouldRasterizeIOS` OR `renderToHardwareTextureAndroid` can help with this significantly.

Be careful not to overuse this or your memory usage could go through the roof. Profile your performance and memory usage when using these props. If you don't plan to move a view anymore, turn this property off.

Animating the size of an image drops UI thread FPS

On iOS, each time you adjust the width or height of an `Image` component it is re-cropped and scaled from the original image. This can be very expensive, especially for large images. Instead, use the `transform: [{scale}]` style property to animate the size. An example of when you might do this is when you tap an image and zoom it

in to full screen.

My TouchableX view isn't very responsive

Sometimes, if we do an action in the same frame that we are adjusting the opacity or highlight of a component that is responding to a touch, we won't see that effect until after the `onPress` function has returned. If `onPress` does a `setState` that results in a lot of work and a few frames dropped, this may occur. A solution to this is to wrap any action inside of your `onPress` handler in `requestAnimationFrame`:

```
handleOnPress() {
  // Always use TimerMixin with requestAnimationFrame, setTimeout and
  // setInterval
  this.requestAnimationFrame(() => {
    this.doExpensiveAction();
  });
}
```

Slow navigator transitions

As mentioned above, `Navigator` animations are controlled by the JavaScript thread. Imagine the "push from right" scene transition: each frame, the new scene is moved from the right to left, starting offscreen (let's say at an x-offset of 320) and ultimately settling when the scene sits at an x-offset of

1. Each frame during this transition, the JavaScript thread needs to send a new x-offset to the main thread. If the JavaScript thread is locked up, it cannot do this and so no update occurs on that frame and the animation stutters.

One solution to this is to allow for JavaScript-based animations to be offloaded to the main thread. If we were to do the same thing as in the above example with this approach, we might calculate a list of all x-offsets for the new scene when we are starting the transition and send them to the main thread to execute in an optimized way. Now that the JavaScript thread is freed of this responsibility, it's not a big deal if it drops a few frames while rendering the scene -- you probably won't even notice because you will be too distracted by the pretty transition.

Solving this is one of the main goals behind the new [React Navigation](#) library. The views in React Navigation use native components and the `Animated` library to deliver 60 FPS animations that are run on the native thread.

Profiling

Use the built-in profiler to get detailed information about work done in the JavaScript thread and main thread side-by-side. Access it by selecting Perf Monitor from the Debug menu.

For iOS, Instruments is an invaluable tool, and on Android you should learn to use `systrace`.

But first, **make sure that Development Mode is OFF!** You should see `__DEV__ === false`, development-level warning are OFF, performance optimizations are ON in your application logs.

Another way to profile JavaScript is to use the Chrome profiler while debugging. This won't give you accurate results as the code is running in Chrome but will give you a general idea of where bottlenecks might be. Run the profiler under Chrome's `Performance` tab. A flame graph will appear under `User Timing`. To view more details in tabular format, click at the `Bottom Up` tab below and then select `DedicatedWorker Thread` at the top left menu.

Profiling Android UI Performance with `systrace`

Android supports 10k+ different phones and is generalized to support software rendering: the framework architecture and need to generalize across many hardware targets unfortunately means you get less for free relative to iOS. But sometimes, there are things you can improve -- and many times it's not native code's fault at all!

The first step for debugging this jank is to answer the fundamental question of where your time is being spent during each 16ms frame. For that, we'll be using a standard Android profiling tool called `systrace`.

`systrace` is a standard Android marker-based profiling tool (and is installed when you install the Android platform-tools package). Profiled code blocks are surrounded by start/end markers which are then visualized in a colorful chart format. Both the Android SDK and React Native framework provide standard markers that you can visualize.

1. Collecting a trace

First, connect a device that exhibits the stuttering you want to investigate to your computer via USB and get it to the point right before the navigation/animation you want to profile. Run `systrace` as follows:

```
$ <path_to_android_sdk>/platform-tools/systrace/systrace.py --time=10 -o trace.html sched gfx view -a <your_package_name>
```

A quick breakdown of this command:

- `time` is the length of time the trace will be collected in seconds
- `sched`, `gfx`, and `view` are the android SDK tags (collections of markers) we care about: `sched` gives you information about what's running on each core of your phone, `gfx` gives you graphics info such as frame boundaries, and `view` gives you information about measure, layout, and draw passes
- `-a <your_package_name>` enables app-specific markers, specifically the ones built into the React Native framework. `your_package_name` can be found in the `AndroidManifest.xml` of your app and looks like `com.example.app`

Once the trace starts collecting, perform the animation or interaction you care about. At the end of the trace, `systrace` will give you a link to the trace which you can open in your browser.

2. Reading the trace

After opening the trace in your browser (preferably Chrome), you should see something like this:



HINT: Use the WASD keys to strafe and zoom

If your trace .html file isn't opening correctly, check your browser console for the following:

Uncaught TypeError: Object.observe is not a function

Since `Object.observe` was deprecated in recent browsers, you may have to open the file from the Google Chrome Tracing tool. You can do so by:

- Opening tab in chrome chrome://tracing
- Selecting load
- Selecting the html file generated from the previous command.

Enable VSync highlighting

Check this checkbox at the top right of the screen to highlight the 16ms frame boundaries:



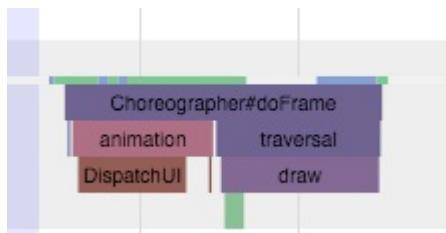
You should see zebra stripes as in the screenshot above. If you don't, try profiling on a different device: Samsung has been known to have issues displaying vsyncs while the Nexus series is generally pretty reliable.

3. Find your process

Scroll until you see (part of) the name of your package. In this case, I was profiling `com.facebook.adsmanager`, which shows up as `book.adsmanager` because of silly thread name limits in the kernel.

On the left side, you'll see a set of threads which correspond to the timeline rows on the right. There are a few threads we care about for our purposes: the UI thread (which has your package name or the name UI Thread), `mqt_js`, and `mqt_native_modules`. If you're running on Android 5+, we also care about the Render Thread.

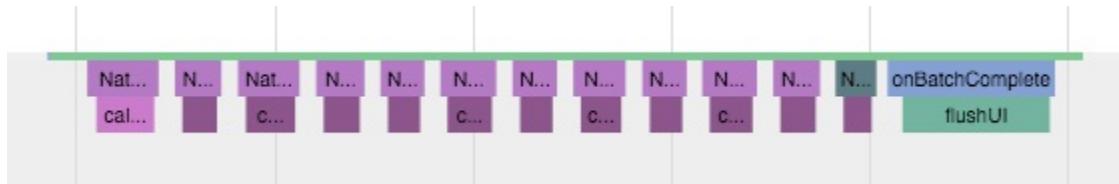
- **UI Thread.** This is where standard android measure/layout/draw happens. The thread name on the right will be your package name (in my case book.adsmanager) or UI Thread. The events that you see on this thread should look something like this and have to do with `Choreographer`, `traversals`, and `DispatchUI`:



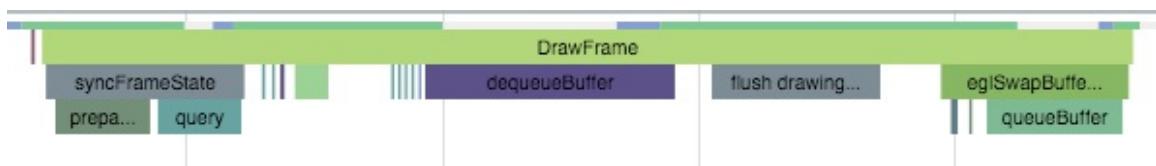
- **JS Thread.** This is where JavaScript is executed. The thread name will be either `mqt_js` or `<....>` depending on how cooperative the kernel on your device is being. To identify it if it doesn't have a name, look for things like `JSCall`, `Bridge.executeJSCall`, etc:



- **Native Modules Thread.** This is where native module calls (e.g. the `UIManager`) are executed. The thread name will be either `mqt_native_modules` or `<....>`. To identify it in the latter case, look for things like `NativeCall`, `callJavaModuleMethod`, and `onBatchComplete`:

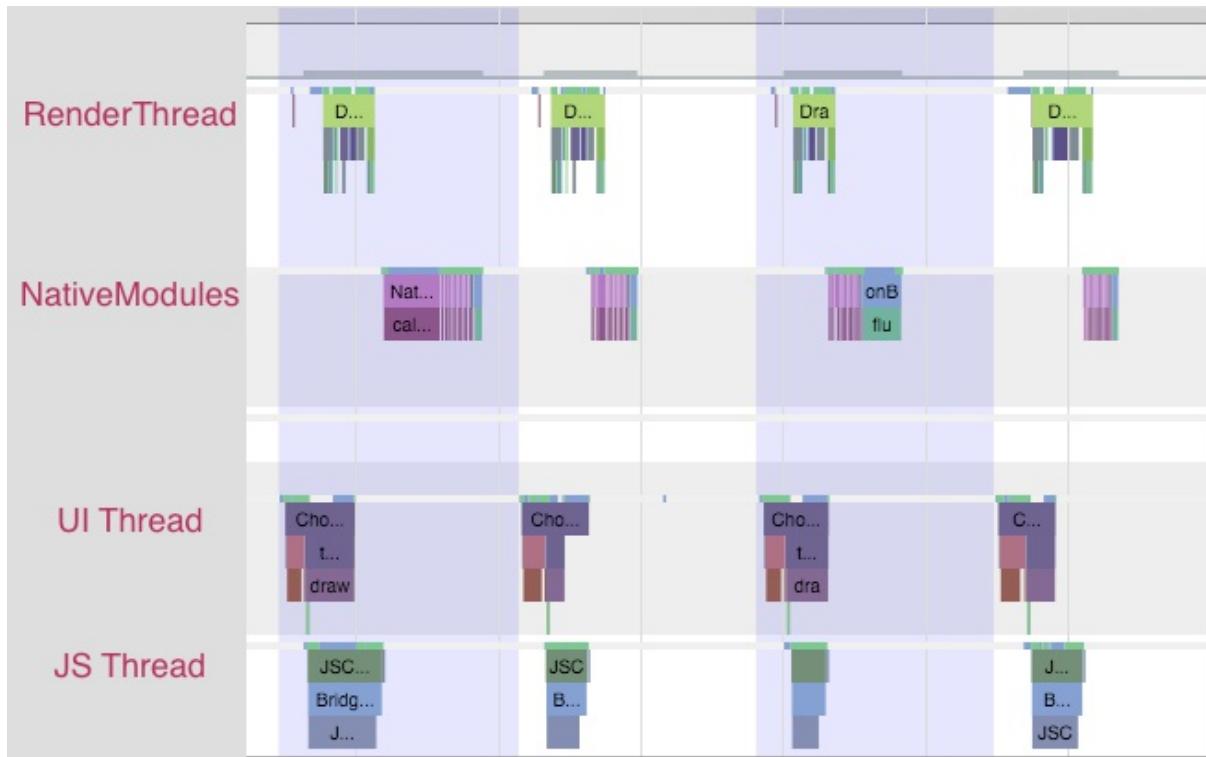


- **Bonus: Render Thread.** If you're using Android L (5.0) and up, you will also have a render thread in your application. This thread generates the actual OpenGL commands used to draw your UI. The thread name will be either `RenderThread` or `<....>`. To identify it in the latter case, look for things like `DrawFrame` and `queueBuffer`:



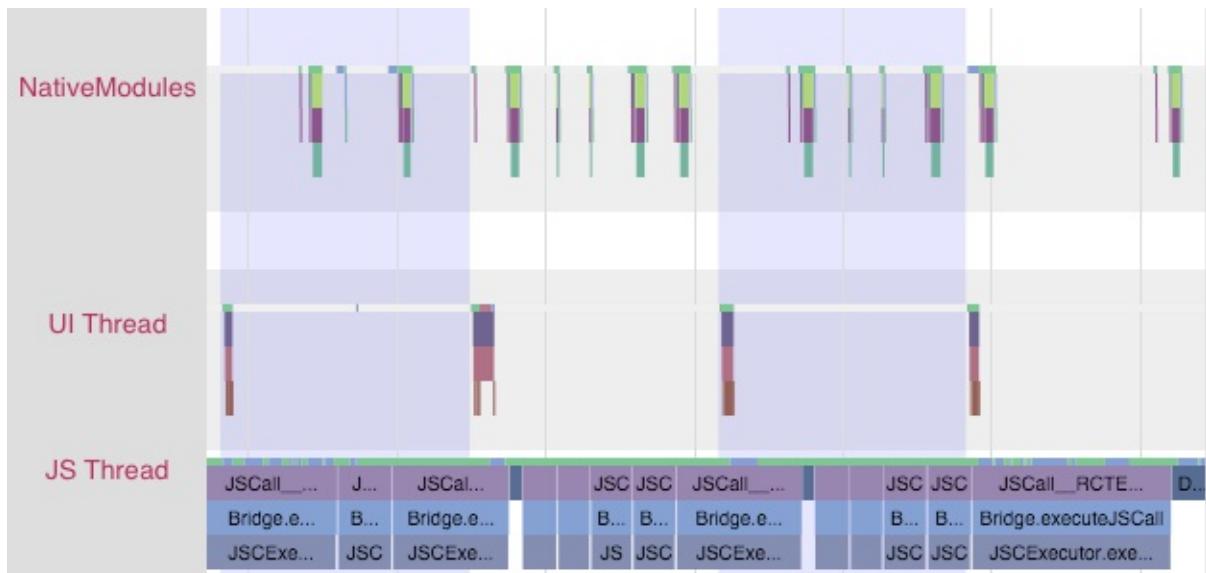
Identifying a culprit

A smooth animation should look something like the following:



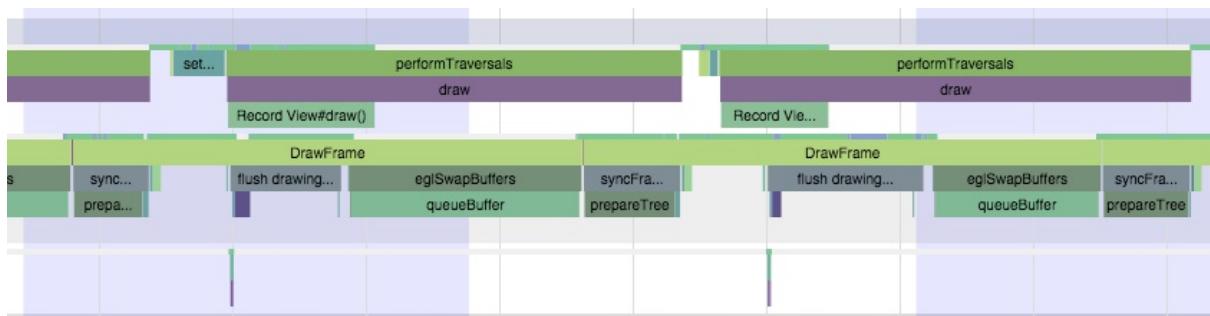
Each change in color is a frame -- remember that in order to display a frame, all our UI work needs to be done by the end of that 16ms period. Notice that no thread is working close to the frame boundary. An application rendering like this is rendering at 60 FPS.

If you noticed chop, however, you might see something like this:



Notice that the JS thread is executing basically all the time, and across frame boundaries! This app is not rendering at 60 FPS. In this case, **the problem lies in JS**.

You might also see something like this:

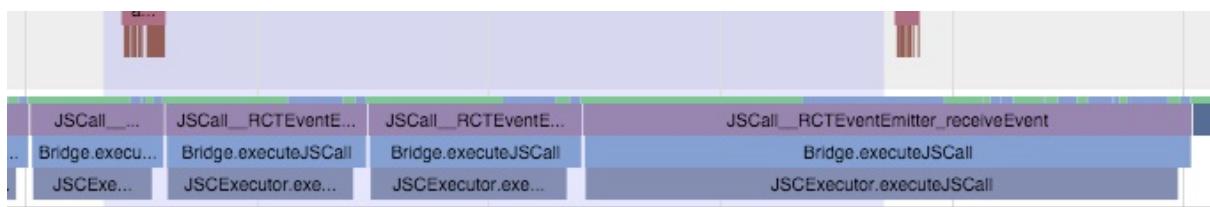


In this case, the UI and render threads are the ones that have work crossing frame boundaries. The UI that we're trying to render on each frame is requiring too much work to be done. In this case, **the problem lies in the native views being rendered.**

At this point, you'll have some very helpful information to inform your next steps.

Resolving JavaScript issues

If you identified a JS problem, look for clues in the specific JS that you're executing. In the scenario above, we see `RCTEventEmitter` being called multiple times per frame. Here's a zoom-in of the JS thread from the trace above:



This doesn't seem right. Why is it being called so often? Are they actually different events? The answers to these questions will probably depend on your product code. And many times, you'll want to look into `shouldComponentUpdate`.

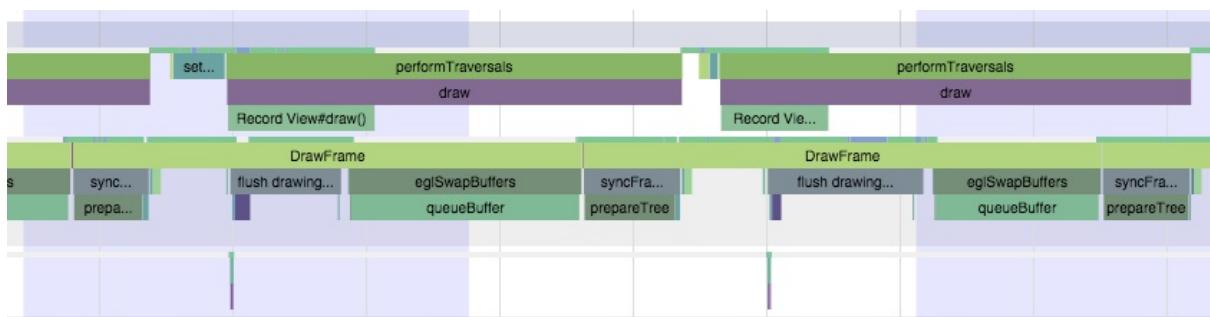
Resolving native UI Issues

If you identified a native UI problem, there are usually two scenarios:

1. the UI you're trying to draw each frame involves too much work on the GPU, or
2. You're constructing new UI during the animation/interaction (e.g. loading in new content during a scroll).

Too much GPU work

In the first scenario, you'll see a trace that has the UI thread and/or Render Thread looking like this:



Notice the long amount of time spent in `DrawFrame` that crosses frame boundaries. This is time spent waiting for the GPU to drain its command buffer from the previous frame.

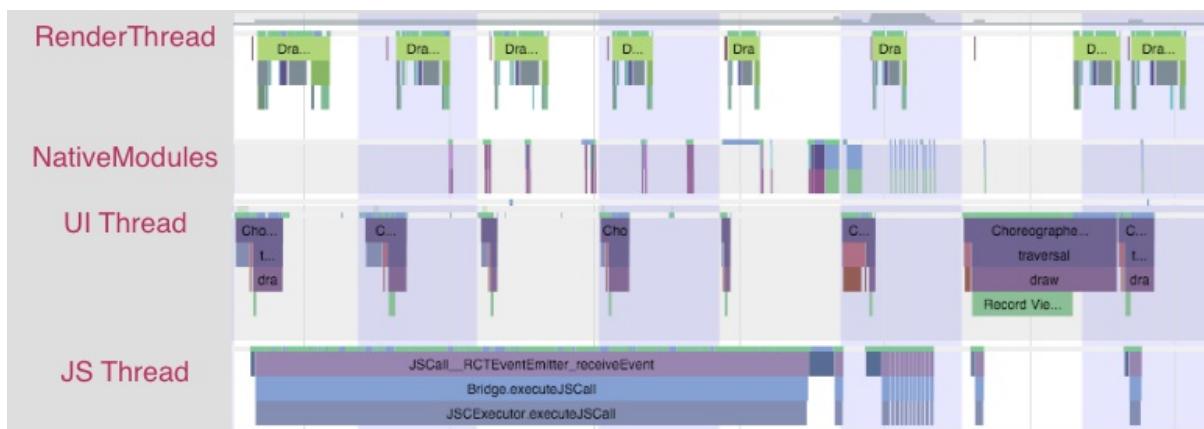
To mitigate this, you should:

- investigate using `renderToHardwareTextureAndroid` for complex, static content that is being animated/transformed (e.g. the `Navigator` slide/alpha animations)
- make sure that you are **not** using `needsOffscreenAlphaCompositing`, which is disabled by default, as it greatly increases the per-frame load on the GPU in most cases.

If these don't help and you want to dig deeper into what the GPU is actually doing, you can check out [Tracer for OpenGL ES](#).

Creating new views on the UI thread

In the second scenario, you'll see something more like this:



Notice that first the JS thread thinks for a bit, then you see some work done on the native modules thread, followed by an expensive traversal on the UI thread.

There isn't an easy way to mitigate this unless you're able to postpone creating new UI until after the interaction, or you are able to simplify the UI you're creating. The react native team is working on an infrastructure level solution for this that will allow new UI to be created and configured off the main thread, allowing the interaction to continue smoothly.

RAM bundles + inline requires

If you have a large app you may want to consider the Random Access Modules (RAM) bundle format, and using inline requires. This is useful for apps that have a large number of screens which may not ever be opened during a typical usage of the app. Generally it is useful to apps that have large amounts of code that are not needed for a while after startup. For instance the app includes complicated profile screens or lesser used features, but most sessions only involve visiting the main screen of the app for updates. We can optimize the loading of the bundle by using the RAM format and requiring those features and screens inline (when they are actually used).

Loading JavaScript

Before react-native can execute JS code, that code must be loaded into memory and parsed. With a standard bundle if you load a 50mb bundle, all 50mb must be loaded and parsed before any of it can be executed. The optimization behind RAM bundles is that you can load only the portion of the 50mb that you actually need at startup, and progressively load more of the bundle as those sections are needed.

Inline Requires

Inline requires delay the requiring of a module or file until that file is actually needed. A basic example would look like this:

VeryExpensive.js

```
import React, { Component } from 'react';
import { Text } from 'react-native';
// ... import some very expensive modules

// You may want to log at the file level to verify when this is happening
console.log('VeryExpensive component loaded');

export default class VeryExpensive extends Component {
  // lots and lots of code
  render() {
    return <Text>Very Expensive Component</Text>;
  }
}
```

Optimized.js

```
import React, { Component } from 'react';
import { TouchableOpacity, View, Text } from 'react-native';

let VeryExpensive = null;

export default class Optimized extends Component {
  state = { needsExpensive: false };

  didPress = () => {
    if (VeryExpensive == null) {
      VeryExpensive = require('./VeryExpensive').default;
    }
  }

  this.setState(() => ({
    needsExpensive: true,
  }));
};

render() {
  return (
    <View style={{ marginTop: 20 }}>
      <TouchableOpacity onPress={this.didPress}>
        <Text>Load</Text>
      </TouchableOpacity>
      {this.state.needsExpensive ? <VeryExpensive /> : null}
    </View>
  );
}
```

Even without the RAM format, inline requires can lead to startup time improvements, because the code within VeryExpensive.js will only execute once it is required for the first time.

Enable the RAM format

On iOS using the RAM format will create a single indexed file that react native will load one module at a time. On Android, by default it will create a set of files for each module. You can force Android to create a single file, like iOS, but using multiple files can be more performant and requires less memory.

Enable the RAM format in Xcode by editing the build phase "Bundle React Native code and images". Before

```
../node_modules/react-native/scripts/react-native-xcode.sh add export BUNDLE_COMMAND="ram-bundle" :
```

```
export BUNDLE_COMMAND="ram-bundle"
export NODE_BINARY=node
../node_modules/react-native/scripts/react-native-xcode.sh
```

On Android enable the RAM format by editing your `android/app/build.gradle` file. Before the line `apply from:`

"`.../node_modules/react-native/react.gradle`" add or amend the `project.ext.react` block:

```
project.ext.react = [
  bundleCommand: "ram-bundle",
]
```

Use the following lines on Android if you want to use a single indexed file:

```
project.ext.react = [
  bundleCommand: "ram-bundle",
  extraPackagerArgs: ["--indexed-ram-bundle"]
]
```

Configure Preloading and Inline Requires

Now that we have a RAM bundle, there is overhead for calling `require`. `require` now needs to send a message over the bridge when it encounters a module it has not loaded yet. This will impact startup the most, because that is where the largest number of require calls are likely to take place while the app loads the initial module. Luckily we can configure a portion of the modules to be preloaded. In order to do this, you will need to implement some form of inline require.

Adding a packager config file

Create a folder in your project called packager, and create a single file named config.js. Add the following:

```
const config = {
  transformer: {
    getTransformOptions: () => {
      return {
        transform: { inlineRequires: true },
      };
    },
  },
};

module.exports = config;
```

In Xcode, in the build phase, include `export BUNDLE_CONFIG="packager/config.js"`.

```
export BUNDLE_COMMAND="ram-bundle"
export BUNDLE_CONFIG="packager/config.js"
export NODE_BINARY=node
../node_modules/react-native/scripts/react-native-xcode.sh
```

Edit your android/app/build.gradle file to include `bundleConfig: "packager/config.js", .`

```
project.ext.react = [
  bundleCommand: "ram-bundle",
  bundleConfig: "packager/config.js"
]
```

Finally, you can update "start" under "scripts" on your package.json to use the config:

```
"start": "node node_modules/react-native/local-cli/cli.js start --config ../../../../../../packager/config.js",
```

Start your package server with `npm start`. Note that when the dev packager is automatically launched via xcode and `react-native run-android`, etc, it does not use `npm start`, so it won't use the config.

Investigating the Loaded Modules

In your root file (index.(ios|android).js) you can add the following after the initial imports:

```
const modules = require.getModules();
const moduleIds = Object.keys(modules);
const loadedModuleNames = moduleIds
  .filter(moduleId => modules[moduleId].isInitialized)
  .map(moduleId => modules[moduleId].verboseName);
const waitingModuleNames = moduleIds
  .filter(moduleId => !modules[moduleId].isInitialized)
  .map(moduleId => modules[moduleId].verboseName);

// make sure that the modules you expect to be waiting are actually waiting
console.log(
  'loaded:',
  loadedModuleNames.length,
  'waiting:',
  waitingModuleNames.length
);

// grab this text blob, and put it in a file named packager/modulePaths.js
console.log(`module.exports = ${JSON.stringify(loadedModuleNames.sort())};`);
```

When you run your app, you can look in the console and see how many modules have been loaded, and how many are waiting. You may want to read the moduleNames and see if there are any surprises. Note that inline requires are invoked the first time the imports are referenced. You may need to investigate and refactor to ensure only the modules you want are loaded on startup. Note that you can change the Systrace object on require to help debug problematic requires.

```
require.Systrace.beginEvent = (message) => {
  if(message.includes(problematicModule)) {
    throw new Error();
  }
}
```

Every app is different, but it may make sense to only load the modules you need for the very first screen. When you are satisfied, put the output of the loadedModuleNames into a file named `packager/modulePaths.js`.

Updating the config.js

Returning to packager/config.js we should update it to use our newly generated modulePaths.js file.

```
const modulePaths = require('./modulePaths');
const resolve = require('path').resolve;
const fs = require('fs');

// Update the following line if the root folder of your app is somewhere else.
const ROOT_FOLDER = path.resolve(__dirname, '..');

const config = {
  transformer: {
    getTransformOptions: () => {
      const moduleMap = {};
      modulePaths.forEach(path => {
        if (fs.existsSync(path)) {
          moduleMap[resolve(path)] = true;
        }
      });
      return {
        preloadedModules: moduleMap,
        transform: { inlineRequires: { blacklist: moduleMap } },
      };
    },
  },
};

module.exports = config;
```

The `preloadedModules` entry in the config indicates which modules should be marked as preloaded when building a RAM bundle. When the bundle is loaded, those modules are immediately loaded, before any requires have even executed. The `blacklist` entry indicates that those modules should not be required inline. Because they are preloaded, there is no performance benefit from using an inline require. In fact the javascript spends extra time resolving the inline require every time the imports are referenced.

Test and Measure Improvements

You should now be ready to build your app using the RAM format and inline requires. Make sure you measure the before and after startup times.

Gesture Responder System

The gesture responder system manages the lifecycle of gestures in your app. A touch can go through several phases as the app determines what the user's intention is. For example, the app needs to determine if the touch is scrolling, sliding on a widget, or tapping. This can even change during the duration of a touch. There can also be multiple simultaneous touches.

The touch responder system is needed to allow components to negotiate these touch interactions without any additional knowledge about their parent or child components.

Best Practices

To make your app feel great, every action should have the following attributes:

- Feedback/highlighting- show the user what is handling their touch, and what will happen when they release the gesture
- Cancel-ability- when making an action, the user should be able to abort it mid-touch by dragging their finger away

These features make users more comfortable while using an app, because it allows people to experiment and interact without fear of making mistakes.

TouchableHighlight and Touchable*

The responder system can be complicated to use. So we have provided an abstract `Touchable` implementation for things that should be "tappable". This uses the responder system and allows you to easily configure tap interactions declaratively. Use `TouchableHighlight` anywhere where you would use a button or link on web.

Responder Lifecycle

A view can become the touch responder by implementing the correct negotiation methods. There are two methods to ask the view if it wants to become responder:

- `View.props.onStartShouldSetResponder: (evt) => true`, - Does this view want to become responder on the start of a touch?
- `View.props.onMoveShouldSetResponder: (evt) => true`, - Called for every touch move on the View when it is not the responder: does this view want to "claim" touch responsiveness?

If the View returns true and attempts to become the responder, one of the following will happen:

- `View.props.onResponderGrant: (evt) => {}` - The View is now responding for touch events. This is the time to highlight and show the user what is happening
- `View.props.onResponderReject: (evt) => {}` - Something else is the responder right now and will not release it

If the view is responding, the following handlers can be called:

- `View.props.onResponderMove: (evt) => {}` - The user is moving their finger
- `View.props.onResponderRelease: (evt) => {}` - Fired at the end of the touch, ie "touchUp"
- `View.props.onResponderTerminationRequest: (evt) => true` - Something else wants to become responder. Should this view release the responder? Returning true allows release

- `View.props.onResponderTerminate: (evt) => {}` - The responder has been taken from the View. Might be taken by other views after a call to `onResponderTerminationRequest`, or might be taken by the OS without asking (happens with control center/ notification center on iOS)

`evt` is a synthetic touch event with the following form:

- `nativeEvent`
 - `changedTouches` - Array of all touch events that have changed since the last event
 - `identifier` - The ID of the touch
 - `locationX` - The X position of the touch, relative to the element
 - `locationY` - The Y position of the touch, relative to the element
 - `pageX` - The X position of the touch, relative to the root element
 - `pageY` - The Y position of the touch, relative to the root element
 - `target` - The node id of the element receiving the touch event
 - `timestamp` - A time identifier for the touch, useful for velocity calculation
 - `touches` - Array of all current touches on the screen

Capture ShouldSet Handlers

`onStartShouldSetResponder` and `onMoveShouldSetResponder` are called with a bubbling pattern, where the deepest node is called first. That means that the deepest component will become responder when multiple Views return true for `*ShouldSetResponder` handlers. This is desirable in most cases, because it makes sure all controls and buttons are usable.

However, sometimes a parent will want to make sure that it becomes responder. This can be handled by using the capture phase. Before the responder system bubbles up from the deepest component, it will do a capture phase, firing `on*ShouldSetResponderCapture`. So if a parent View wants to prevent the child from becoming responder on a touch start, it should have a `onStartShouldSetResponderCapture` handler which returns true.

- `View.props.onStartShouldSetResponderCapture: (evt) => true,`
- `View.props.onMoveShouldSetResponderCapture: (evt) => true,`

PanResponder

For higher-level gesture interpretation, check out [PanResponder](#).

JavaScript Environment

JavaScript Runtime

When using React Native, you're going to be running your JavaScript code in two environments:

- In most cases, React Native will use [JavaScriptCore](#), the JavaScript engine that powers Safari. Note that on iOS, JavaScriptCore does not use JIT due to the absence of writable executable memory in iOS apps.
- When using Chrome debugging, all JavaScript code runs within Chrome itself, communicating with native code via WebSockets. Chrome uses [V8](#) as its JavaScript engine.

While both environments are very similar, you may end up hitting some inconsistencies. We're likely going to experiment with other JavaScript engines in the future, so it's best to avoid relying on specifics of any runtime.

JavaScript Syntax Transformers

Syntax transformers make writing code more enjoyable by allowing you to use new JavaScript syntax without having to wait for support on all interpreters.

React Native ships with the [Babel JavaScript compiler](#). Check [Babel documentation](#) on its supported transformations for more details.

Here's a full list of React Native's [enabled transformations](#).

ES5

- [Reserved Words](#): `promise.catch(function() { })`;

ES6

- [Arrow functions](#): `<C onPress={() => this.setState({pressed: true})}>`
- [Block scoping](#): `let greeting = 'hi';`
- [Call spread](#): `Math.max(...array);`
- [Classes](#): `class C extends React.Component { render() { return <View />; } }`
- [Constants](#): `const answer = 42;`
- [Destructuring](#): `var {isActive, style} = this.props;`
- [for...of](#): `for (var num of [1, 2, 3]) {}`
- [Modules](#): `import React, { Component } from 'react';`
- [Computed Properties](#): `var key = 'abc'; var obj = {[key]: 10};`
- [Object Concise Method](#): `var obj = { method() { return 10; } };`
- [Object Short Notation](#): `var name = 'vjeux'; var obj = { name };`
- [Rest Params](#): `function(type, ...args) { }`
- [Template Literals](#): `var who = 'world'; var str = `Hello ${who}`;`

ES8

- [Function Trailing Comma](#): `function f(a, b, c,) { }`
- [Async Functions](#): `async function doStuffAsync() { const foo = await doOtherStuffAsync(); } ;`

Stage 3

- **Object Spread**: `var extended = { ...obj, a: 10 };`

Specific

- **JSX**: `<View style={{color: 'red'}} />`
- **Flow**: `function foo(x: ?number): string {}`

Polyfills

Many standards functions are also available on all the supported JavaScript runtimes.

Browser

- `console.{log, warn, error, info, trace, table, group, groupEnd}`
- CommonJS require
- XMLHttpRequest, fetch
- `{set, clear}{Timeout, Interval, Immediate}, {request, cancel}AnimationFrame`
- `navigator.geolocation`

ES6

- `Object.assign`
- `String.prototype.{startsWith, endsWith, repeat, includes}`
- `Array.from`
- `Array.prototype.{find, findIndex}`

ES7

- `Array.prototype.{includes}`

ES8

- `Object.{entries, values}`

Specific

- `__DEV__`

Direct Manipulation

It is sometimes necessary to make changes directly to a component without using state/props to trigger a re-render of the entire subtree. When using React in the browser for example, you sometimes need to directly modify a DOM node, and the same is true for views in mobile apps. `setNativeProps` is the React Native equivalent to setting properties directly on a DOM node.

Use `setNativeProps` when frequent re-rendering creates a performance bottleneck

Direct manipulation will not be a tool that you reach for frequently; you will typically only be using it for creating continuous animations to avoid the overhead of rendering the component hierarchy and reconciling many views. `setNativeProps` is imperative and stores state in the native layer (DOM, UIView, etc.) and not within your React components, which makes your code more difficult to reason about. Before you use it, try to solve your problem with `setState` and `shouldComponentUpdate`.

setNativeProps with TouchableOpacity

`TTouchableOpacity` uses `setNativeProps` internally to update the opacity of its child component:

```
setOpacityTo(value) {
  // Redacted: animation related code
  this.refs[CHILD_REF].setNativeProps({
    opacity: value
  });
},
```

This allows us to write the following code and know that the child will have its opacity updated in response to taps, without the child having any knowledge of that fact or requiring any changes to its implementation:

```
<TouchableOpacity onPress={this._handlePress}>
  <View style={styles.button}>
    <Text>Press me!</Text>
  </View>
</TouchableOpacity>
```

Let's imagine that `setNativeProps` was not available. One way that we might implement it with that constraint is to store the opacity value in the state, then update that value whenever `onPress` is fired:

```
constructor(props) {
  super(props);
  this.state = { myButtonOpacity: 1, };
}

render() {
  return (
    <TouchableOpacity onPress={() => this.setState({myButtonOpacity: 0.5})}
      onPressOut={() => this.setState({myButtonOpacity: 1})}>
      <View style={[styles.button, {opacity: this.state.myButtonOpacity}]}>
        <Text>Press me!</Text>
      </View>
    </TouchableOpacity>
  );
}
```

```
        </TouchableOpacity>
    )
}
```

This is computationally intensive compared to the original example - React needs to re-render the component hierarchy each time the opacity changes, even though other properties of the view and its children haven't changed. Usually this overhead isn't a concern but when performing continuous animations and responding to gestures, judiciously optimizing your components can improve your animations' fidelity.

If you look at the implementation of `setNativeProps` in [NativeMethodsMixin](#) you will notice that it is a wrapper around `RCTUIManager.updateView` - this is the exact same function call that results from re-rendering - see [receiveComponent](#) in [ReactNativeBaseComponent.js](#).

Composite components and `setNativeProps`

Composite components are not backed by a native view, so you cannot call `setNativeProps` on them. Consider this example:

```
import React from 'react';
import { Text, TouchableOpacity, View } from 'react-native';

class MyButton extends React.Component {
  render() {
    return (
      <View>
        <Text>{this.props.label}</Text>
      </View>
    )
  }
}

export default class App extends React.Component {
  render() {
    return (
      <TouchableOpacity>
        <MyButton label="Press me!" />
      </TouchableOpacity>
    )
  }
}
```

If you run this you will immediately see this error: `Touchable child must either be native or forward setNativeProps to a native component`. This occurs because `MyButton` isn't directly backed by a native view whose opacity should be set. You can think about it like this: if you define a component with `createClass` you would not expect to be able to set a style prop on it and have that work - you would need to pass the style prop down to a child, unless you are wrapping a native component. Similarly, we are going to forward `setNativeProps` to a native-backed child component.

Forward `setNativeProps` to a child

All we need to do is provide a `setNativeProps` method on our component that calls `setNativeProps` on the appropriate child with the given arguments.

```

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

class MyButton extends React.Component {
  setNativeProps = (nativeProps) => {
    this._root.setNativeProps(nativeProps);
  }

  render() {
    return (
      <View ref={component => this._root = component} {...this.props}>
        <Text>{this.props.label}</Text>
      </View>
    )
  }
}

export default class App extends React.Component {
  render() {
    return (
      <TouchableOpacity>
        <MyButton label="Press me!" />
      </TouchableOpacity>
    )
  }
}

```

You can now use `MyButton` inside of `TouchableOpacity`! A sidenote for clarity: we used the [ref callback](#) syntax here, rather than the traditional string-based ref.

You may have noticed that we passed all of the props down to the child view using `{...this.props}`. The reason for this is that `TouchableOpacity` is actually a composite component, and so in addition to depending on `setNativeProps` on its child, it also requires that the child perform touch handling. To do this, it passes on [various props](#) that call back to the `TouchableOpacity` component. `TouchableHighlight`, in contrast, is backed by a native view and only requires that we implement `setNativeProps`.

setNativeProps to clear TextInput value

Another very common use case of `setNativeProps` is to clear the value of a `TextInput`. The `controlled` prop of `TextInput` can sometimes drop characters when the `bufferDelay` is low and the user types very quickly. Some developers prefer to skip this prop entirely and instead use `setNativeProps` to directly manipulate the `TextInput` value when necessary. For example, the following code demonstrates clearing the input when you tap a button:

```

import React from 'react';
import { TextInput, Text, TouchableOpacity, View } from 'react-native';

export default class App extends React.Component {
  clearText = () => {
    this._textInput.setNativeProps({text: ''});
  }

  render() {
    return (
      <View style={{flex: 1}}>
        <TextInput
          ref={component => this._textInput = component}
          style={{height: 50, flex: 1, marginHorizontal: 20, borderWidth: 1, borderColor: '#ccc'}}/>
      </View>
    )
  }
}

```

```
        <TouchableOpacity onPress={this.clearText}>
          <Text>Clear text</Text>
        </TouchableOpacity>
      </View>
    );
}
}
```

Avoiding conflicts with the render function

If you update a property that is also managed by the render function, you might end up with some unpredictable and confusing bugs because anytime the component re-renders and that property changes, whatever value was previously set from `setNativeProps` will be completely ignored and overridden.

setNativeProps & shouldComponentUpdate

By intelligently applying `shouldComponentUpdate` you can avoid the unnecessary overhead involved in reconciling unchanged component subtrees, to the point where it may be performant enough to use `setState` instead of `setNativeProps`.

Other native methods

The methods described here are available on most of the default components provided by React Native. Note, however, that they are *not* available on composite components that aren't directly backed by a native view. This will generally include most components that you define in your own app.

measure(callback)

Determines the location on screen, width, and height of the given view and returns the values via an async callback. If successful, the callback will be called with the following arguments:

- x
- y
- width
- height
- pageX
- pageY

Note that these measurements are not available until after the rendering has been completed in native. If you need the measurements as soon as possible, consider using the `onLayout` prop instead.

measureInWindow(callback)

Determines the location of the given view in the window and returns the values via an async callback. If the React root view is embedded in another native view, this will give you the absolute coordinates. If successful, the callback will be called with the following arguments:

- x
- y
- width

- height

measureLayout(relativeToNativeNode, onSuccess, onFailure)

Like `measure()`, but measures the view relative an ancestor, specified as `relativeToNativeNode`. This means that the returned x, y are relative to the origin x, y of the ancestor view.

As always, to obtain a native node handle for a component, you can use `ReactNative.findNodeHandle(component)`.

focus()

Requests focus for the given input or view. The exact behavior triggered will depend on the platform and type of view.

blur()

Removes focus from an input or view. This is the opposite of `focus()`.

Color Reference

Components in React Native are [styled using JavaScript](#). Color properties usually match how [CSS works on the web](#).

Red-green-blue

React Native supports `rgb()` and `rgba()` in both hexadecimal and functional notation:

- `'#f0f' (#rgb)`
- `'#ff00ff' (#rrggbba)`
- `'rgb(255, 0, 255)'`
- `'rgba(255, 255, 255, 1.0)'`
- `'#f0ff' (#rgba)`
- `'#ff00ffff' (#rrgbbbaaa)`

Hue-saturation-lightness

`hsl()` and `hsla()` is supported in functional notation:

- `'hsl(360, 100%, 100%)'`
- `'hsla(360, 100%, 100%, 1.0)'`

transparent

This is a shortcut for `rgba(0,0,0,0)`:

- `'transparent'`

Named colors

You can also use color names as values. React Native follows the [CSS3 specification](#):

- `aliceblue (#f0f8ff)`
- `antiquewhite (#faebd7)`
- `aqua (#00ffff)`
- `aquamarine (#7fffdd)`
- `azure (#f0ffff)`
- `beige (#f5f5dc)`
- `bisque (#ffe4c4)`
- `black (#000000)`
- `blanchedalmond (#ffebcd)`
- `blue (#0000ff)`
- `blueviolet (#8a2be2)`
- `brown (#a52a2a)`
- `burlywood (#deb887)`
- `cadetblue (#5f9ea0)`

- chartreuse (#7fff00)
- chocolate (#d2691e)
- coral (#ff7f50)
- cornflowerblue (#6495ed)
- cornsilk (#fff8dc)
- crimson (#dc143c)
- cyan (#00ffff)
- darkblue (#00008b)
- darkcyan (#008b8b)
- darkgoldenrod (#b8860b)
- darkgray (#a9a9a9)
- darkgreen (#006400)
- darkgrey (#a9a9a9)
- darkkhaki (#bdb76b)
- darkmagenta (#8b008b)
- darkolivegreen (#556b2f)
- darkorange (#ff8c00)
- darkorchid (#9932cc)
- darkred (#8b0000)
- darksalmon (#e9967a)
- darkseagreen (#8fbcb8)
- darkslateblue (#483d8b)
- darkslategray (#2f4f4f)
- darkturquoise (#00ced1)
- darkviolet (#9400d3)
- deeppink (#ff1493)
- deepskyblue (#00bfff)
- dimgray (#696969)
- dimgrey (#696969)
- dodgerblue (#1e90ff)
- firebrick (#b22222)
- floralwhite (#ffffaf0)
- forestgreen (#228b22)
- fuchsia (#ff00ff)
- gainsboro (#dcdcdc)
- ghostwhite (#f8f8ff)
- gold (#ffd700)
- goldenrod (#daa520)
- gray (#808080)
- green (#008000)
- greenyellow (#adff2f)
- grey (#808080)
- honeydew (#f0ffff0)
- hotpink (#ff69b4)
- indianred (#cd5c5c)
- indigo (#4b0082)
- ivory (#fffff0)
- khaki (#f0e68c)
- lavender (#e6e6fa)

- lavenderblush (#ffff0f5)
- lawngreen (#7cfc00)
- lemonchiffon (#ffffacd)
- lightblue (#add8e6)
- lightcoral (#f08080)
- lightcyan (#e0ffff)
- lightgoldenrodyellow (#fafad2)
- lightgray (#d3d3d3)
- lightgreen (#90ee90)
- lightgrey (#d3d3d3)
- lightpink (#ffb6c1)
- lightsalmon (#ffa07a)
- lightseagreen (#20b2aa)
- lightskyblue (#87cefa)
- lightslategrey (#778899)
- lightsteelblue (#b0c4de)
- lightyellow (#ffffe0)
- lime (#00ff00)
- limegreen (#32cd32)
- linen (#faf0e6)
- magenta (#ff00ff)
- maroon (#800000)
- mediumaquamarine (#66cdaa)
- mediumblue (#0000cd)
- mediumorchid (#ba55d3)
- mediumpurple (#9370db)
- mediumseagreen (#3cb371)
- mediumslateblue (#7b68ee)
- mediumspringgreen (#00fa9a)
- mediumturquoise (#48d1cc)
- mediumvioletred (#c71585)
- midnightblue (#191970)
- mintcream (#f5ffff)
- mistyrose (#ffe4e1)
- moccasin (#ffe4b5)
- navajowhite (#ffddead)
- navy (#000080)
- oldlace (#fdf5e6)
- olive (#808000)
- olivedrab (#6b8e23)
- orange (#ffa500)
- orangered (#ff4500)
- orchid (#da70d6)
- palegoldenrod (#eee8aa)
- palegreen (#98fb98)
- paleturquoise (#afeeee)
- palevioletred (#db7093)
- papayawhip (#ffefd5)
- peachpuff (#ffdab9)

- peru (#cd853f)
- pink (#ffc0cb)
- plum (#dda0dd)
- powderblue (#b0e0e6)
- purple (#800080)
- rebeccapurple (#663399)
- red (#ff0000)
- rosybrown (#bc8f8f)
- royalblue (#4169e1)
- saddlebrown (#8b4513)
- salmon (#fa8072)
- sandybrown (#f4a460)
- seagreen (#2e8b57)
- seashell (#fff5ee)
- sienna (#a0522d)
- silver (#c0c0c0)
- skyblue (#87ceeb)
- slateblue (#6a5acd)
- slategray (#708090)
- snow (#fffffa)
- springgreen (#00ff7f)
- steelblue (#4682b4)
- tan (#d2b48c)
- teal (#008080)
- thistle (#d8bfd8)
- tomato (#ff6347)
- turquoise (#40e0d0)
- violet (#ee82ee)
- wheat (#f5deb3)
- white (#ffffff)
- whitesmoke (#f5f5f5)
- yellow (#ffff00)
- yellowgreen (#9acd32)

ActivityIndicator

Displays a circular loading indicator.

Example

```
import React, { Component } from 'react'
import {
  ActivityIndicator,
  AppRegistry,
  StyleSheet,
  Text,
  View,
} from 'react-native'

export default class App extends Component {
  render() {
    return (
      <View style={[styles.container, styles.horizontal]}>
        <ActivityIndicator size="large" color="#0000ff" />
        <ActivityIndicator size="small" color="#00ff00" />
        <ActivityIndicator size="large" color="#0000ff" />
        <ActivityIndicator size="small" color="#00ff00" />
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center'
  },
  horizontal: {
    flexDirection: 'row',
    justifyContent: 'space-around',
    padding: 10
  }
})

AppRegistry.registerComponent('App', () => App)
```

Props

- [View props...](#)
- `animating`
- `color`
- `size`
- `hidesWhenStopped`

Reference

Props

animating

Whether to show the indicator (true, the default) or hide it (false).

Type	Required
bool	No

color

The foreground color of the spinner (default is gray).

Type	Required
color	No

size

Size of the indicator (default is 'small'). Passing a number to the size prop is only supported on Android.

Type	Required
enum('small', 'large'), ,number	No

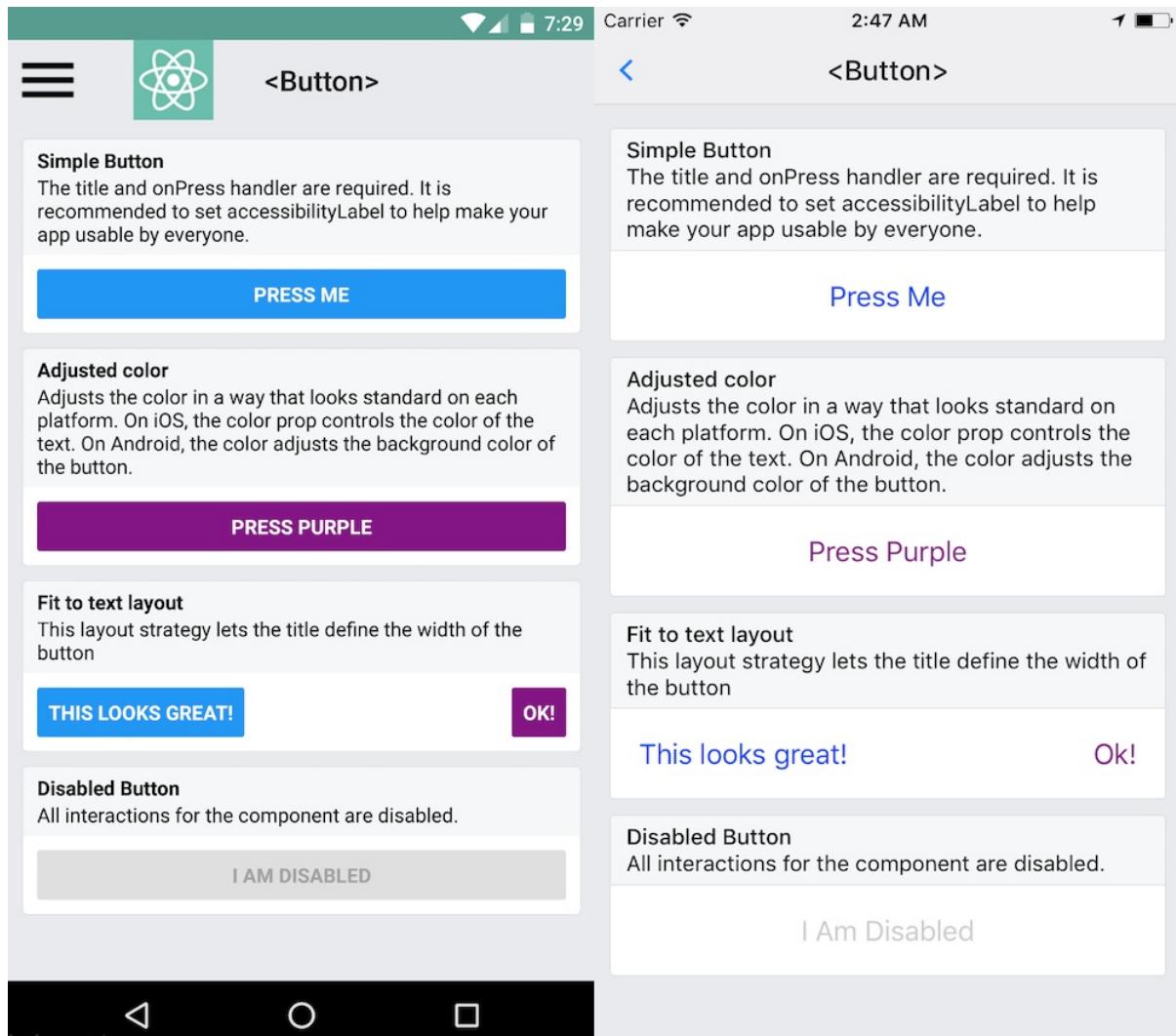
hidesWhenStopped

Whether the indicator should hide when not animating (true by default).

Type	Required	Platform
bool	No	iOS

Button

A basic button component that should render nicely on any platform. Supports a minimal level of customization.



If this button doesn't look right for your app, you can build your own button using [TouchableOpacity](#) or [TouchableNativeFeedback](#). For inspiration, look at the [source code for this button component](#). Or, take a look at the [wide variety of button components built by the community](#).

Example usage:

```
import { Button } from 'react-native';
...
<Button
  onPress={onPressLearnMore}
  title="Learn More"
  color="#841584"
  accessibilityLabel="Learn more about this purple button"
/>
```

Props

- `onPress`
 - `title`
 - `accessibilityLabel`
 - `color`
 - `disabled`
 - `testID`
 - `hasTVPreferredFocus`
-

Reference

Props

`onPress`

Handler to be called when the user taps the button

Type	Required
function	Yes

`title`

Text to display inside the button

Type	Required
string	Yes

`accessibilityLabel`

Text to display for blindness accessibility features

Type	Required
string	No

`color`

Color of the text (iOS), or background color of the button (Android)

Type	Required
<code>color</code>	No

disabled

If true, disable all interactions for this component.

Type	Required
bool	No

testID

Used to locate this view in end-to-end tests.

Type	Required
string	No

hasTVPREFERREDFocus

(*Apple TV only*) TV preferred focus (see documentation for the View component).

Type	Required	Platform
bool	No	iOS

DatePickerIOS

Use `DatePickerIOS` to render a date/time picker (selector) on iOS. This is a controlled component, so you must hook in to the `onDateChange` callback and update the `date` prop in order for the component to update, otherwise the user's change will be reverted immediately to reflect `props.date` as the source of truth.

Example

```
import React, { Component } from 'react'
import {
  DatePickerIOS,
  View,
  StyleSheet,
} from 'react-native'

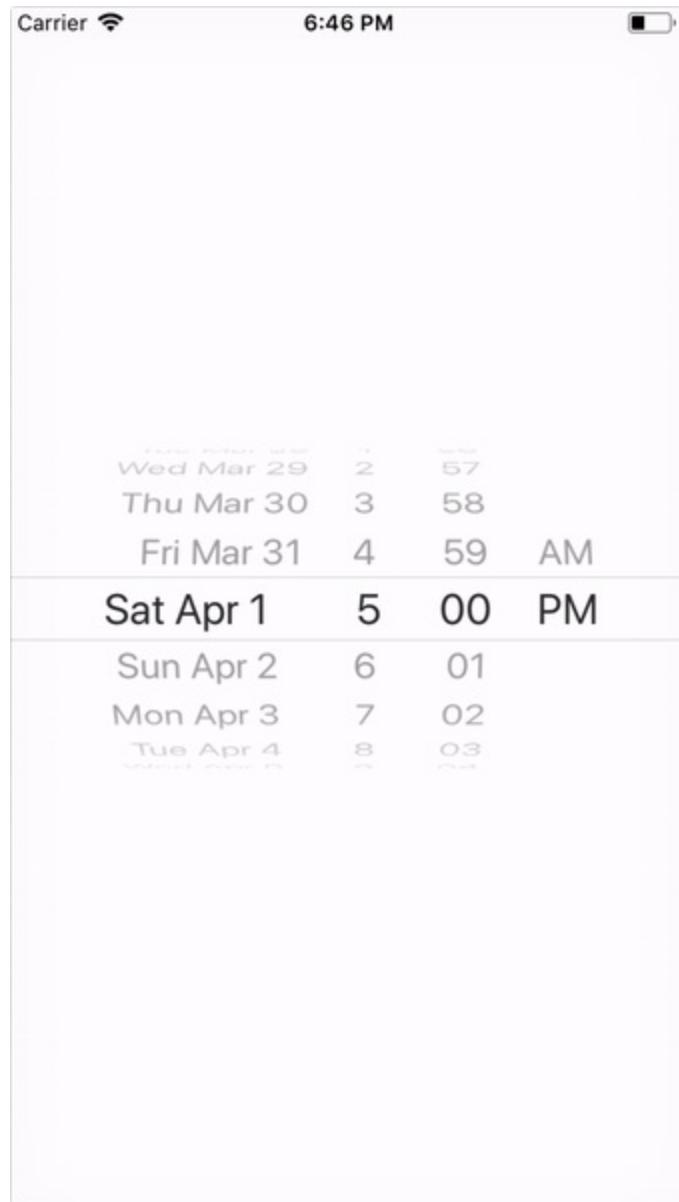
export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = { chosenDate: new Date() };

    this.setDate = this.setDate.bind(this);
  }

  setDate(newDate) {
    this.setState({chosenDate: newDate})
  }

  render() {
    return (
      <View style={styles.container}>
        <DatePickerIOS
          date={this.state.chosenDate}
          onDateChange={this.setDate}
        />
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center'
  },
})
```



Props

- [View props...](#)
- `date`
- `onDateChange`
- `maximumDate`
- `minimumDate`
- `minuteInterval`
- `mode`
- `locale`
- `timeZoneOffsetInMinutes`

Reference

Props

date

The currently selected date.

Type	Required
Date	Yes

onDateChange

Date change handler.

This is called when the user changes the date or time in the UI. The first and only argument is a Date object representing the new date and time.

Type	Required
function	Yes

maximumDate

Maximum date.

Restricts the range of possible date/time values.

Type	Required
Date	No

Example with `maximumDate` set to December 31, 2017:



minimumDate

Minimum date.

Restricts the range of possible date/time values.

--	--	--

Type	Required
Date	No

See `maximumDate` for an example image.

minuteInterval

The interval at which minutes can be selected.

Type	Required
enum(1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30)	No

Example with `minuteInterval` set to `10`:

Fri Dec 1	3	20	
Sat Dec 2	4	30	
Sun Dec 3	5	40	AM
Today	6	50	PM
Tue Dec 5	7	00	
Wed Dec 6	8	10	
Thu Dec 7	9	20	

mode

The date picker mode.

Type	Required
enum('date', 'time', 'datetime')	No

Example with `mode` set to `date`, `time`, and `datetime`:

September	1	2014	4	07		Fri Dec 1	4	07	
October	2	2015	5	08		Sat Dec 2	5	08	
November	3	2016	6	09	AM	Sun Dec 3	6	09	AM
December	4	2017	7	10	PM	Today	7	10	PM
January	5	2018	8	11		Tue Dec 5	8	11	
February	6	2019	9	12		Wed Dec 6	9	12	
March	7	2020	10	13		Thu Dec 7	10	13	

locale

The locale for the date picker. Value needs to be a [Locale ID](#).

Type	Required
String	No

timeZoneOffsetInMinutes

Timezone offset in minutes.

By default, the date picker will use the device's timezone. With this parameter, it is possible to force a certain timezone offset. For instance, to show times in Pacific Standard Time, pass `-7 * 60`.

Type	Required
number	No

DrawerLayoutAndroid

React component that wraps the platform `DrawerLayout` (Android only). The Drawer (typically used for navigation) is rendered with `renderNavigationView` and direct children are the main view (where your content goes). The navigation view is initially not visible on the screen, but can be pulled in from the side of the window specified by the `drawerPosition` prop and its width can be set by the `drawerWidth` prop.

Example:

```
render: function() {
  var navigationView = (
    <View style={{flex: 1, backgroundColor: '#fff'}}>
      <Text style={{margin: 10, fontSize: 15, textAlign: 'left'}}>I'm in the Drawer!</Text>
    </View>
  );
  return (
    <DrawerLayoutAndroid
      drawerWidth={300}
      drawerPosition={DrawerLayoutAndroid.positions.Left}
      renderNavigationView={() => navigationView}>
      <View style={{flex: 1, alignItems: 'center'}}>
        <Text style={{margin: 10, fontSize: 15, textAlign: 'right'}}>Hello</Text>
        <Text style={{margin: 10, fontSize: 15, textAlign: 'right'}}>World!</Text>
      </View>
    </DrawerLayoutAndroid>
  );
},
```

Props

- View props...
- `renderNavigationView`
- `onDrawerClose`
- `drawerPosition`
- `drawerWidth`
- `keyboardDismissMode`
- `drawerLockMode`
- `onDrawerOpen`
- `onDrawerSlide`
- `onDrawerStateChanged`
- `drawerBackgroundColor`
- `statusBarBackgroundColor`

Methods

- `openDrawer`
- `closeDrawer`

Reference

Props

renderNavigationView

The navigation view that will be rendered to the side of the screen and can be pulled in.

Type	Required
function	Yes

onDrawerClose

Function called whenever the navigation view has been closed.

Type	Required
function	No

drawerPosition

Specifies the side of the screen from which the drawer will slide in.

Type	Required
enum(DrawerConsts.DrawerPosition.Left, DrawerConsts.DrawerPosition.Right)	No

drawerWidth

Specifies the width of the drawer, more precisely the width of the view that be pulled in from the edge of the window.

Type	Required
number	No

keyboardDismissMode

Determines whether the keyboard gets dismissed in response to a drag.

- 'none' (the default), drags do not dismiss the keyboard.
- 'on-drag', the keyboard is dismissed when a drag begins.

Type	Required
enum('none', 'on-drag')	No

drawerLockMode

Specifies the lock mode of the drawer. The drawer can be locked in 3 states:

- unlocked (default), meaning that the drawer will respond (open/close) to touch gestures.
- locked-closed, meaning that the drawer will stay closed and not respond to gestures.
- locked-open, meaning that the drawer will stay opened and not respond to gestures. The drawer may still be opened and closed programmatically (`openDrawer` / `closeDrawer`).

Type	Required
<code>enum('unlocked', 'locked-closed', 'locked-open')</code>	No

onDrawerOpen

Function called whenever the navigation view has been opened.

Type	Required
<code>function</code>	No

onDrawerSlide

Function called whenever there is an interaction with the navigation view.

Type	Required
<code>function</code>	No

onDrawerStateChanged

Function called when the drawer state has changed. The drawer can be in 3 states:

- idle, meaning there is no interaction with the navigation view happening at the time
- dragging, meaning there is currently an interaction with the navigation view
- settling, meaning that there was an interaction with the navigation view, and the navigation view is now finishing its closing or opening animation

Type	Required
<code>function</code>	No

drawerBackgroundColor

Specifies the background color of the drawer. The default value is white. If you want to set the opacity of the drawer, use `rgba`. Example:

```
return (
<DrawerLayoutAndroid drawerBackgroundColor="rgba(0,0,0,0.5)">
</DrawerLayoutAndroid>
);
```

Type	Required
color	No

statusBarBackgroundColor

Make the drawer take the entire screen and draw the background of the status bar to allow it to open over the status bar. It will only have an effect on API 21+.

Type	Required
color	No

Methods

openDrawer()

```
openDrawer();
```

Opens the drawer.

closeDrawer()

```
closeDrawer();
```

Closes the drawer.

FlatList

A performant interface for rendering simple, flat lists, supporting the most handy features:

- Fully cross-platform.
- Optional horizontal mode.
- Configurable viewability callbacks.
- Header support.
- Footer support.
- Separator support.
- Pull to Refresh.
- Scroll loading.
- ScrollToIndex support.

If you need section support, use `<SectionList>`.

Minimal Example:

```
<FlatList
  data={[{key: 'a'}, {key: 'b'}]}
  renderItem={({item}) => <Text>{item.key}</Text>}
/>
```

More complex, multi-select example demonstrating `PureComponent` usage for perf optimization and avoiding bugs.

- By binding the `onPressItem` handler, the props will remain `==` and `PureComponent` will prevent wasteful re-renders unless the actual `id`, `selected`, or `title` props change, even if the components rendered in `MyListItem` did not have such optimizations.
- By passing `extraData={this.state}` to `FlatList` we make sure `FlatList` itself will re-render when the `state.selected` changes. Without setting this prop, `FlatList` would not know it needs to re-render any items because it is also a `PureComponent` and the prop comparison will not show any changes.
- `keyExtractor` tells the list to use the `id`s for the react keys instead of the default `key` property.

```
class MyListItem extends React.PureComponent {
  _onPress = () => {
    this.props.onPressItem(this.props.id);
  };

  render() {
    const textColor = this.props.selected ? "red" : "black";
    return (
      <TouchableOpacity onPress={this._onPress}>
        <View>
          <Text style={{ color: textColor }}>
            {this.props.title}
          </Text>
        </View>
      </TouchableOpacity>
    );
  }
}

class MultiSelectList extends React.PureComponent {
  state = {selected: (new Map()): Map<string, boolean>};
```

```

    _keyExtractor = (item, index) => item.id;

    _onPressItem = (id: string) => {
      // updater functions are preferred for transactional updates
      this.setState((state) => {
        // copy the map rather than modifying state.
        const selected = new Map(state.selected);
        selected.set(id, !selected.get(id)); // toggle
        return {selected};
      });
    };

    _renderItem = ({item}) => (
      <MyListItem
        id={item.id}
        onPressItem={this._onPressItem}
        selected={!!this.state.selected.get(item.id)}
        title={item.title}
      />
    );

    render() {
      return (
        <FlatList
          data={this.props.data}
          extraData={this.state}
          keyExtractor={this._keyExtractor}
          renderItem={this._renderItem}
        />
      );
    }
  }
}

```

This is a convenience wrapper around `<VirtualizedList>`, and thus inherits its props (as well as those of `<ScrollView>`) that aren't explicitly listed here, along with the following caveats:

- Internal state is not preserved when content scrolls out of the render window. Make sure all your data is captured in the item data or external stores like Flux, Redux, or Relay.
- This is a `PureComponent` which means that it will not re-render if `props` remain shallow- equal. Make sure that everything your `renderItem` function depends on is passed as a prop (e.g. `extraData`) that is not `==` after updates, otherwise your UI may not update on changes. This includes the `data` prop and parent component state.
- In order to constrain memory and enable smooth scrolling, content is rendered asynchronously offscreen. This means it's possible to scroll faster than the fill rate and momentarily see blank content. This is a tradeoff that can be adjusted to suit the needs of each application, and we are working on improving it behind the scenes.
- By default, the list looks for a `key` prop on each item and uses that for the React key. Alternatively, you can provide a custom `keyExtractor` prop.

Also inherits [ScrollView Props](#), unless it is nested in another FlatList of same orientation.

Props

- `ScrollView` props...
- `VirtualizedList` props...
- `renderItem`
- `data`

- `ItemSeparatorComponent`
- `ListEmptyComponent`
- `ListFooterComponent`
- `ListHeaderComponent`
- `columnWrapperStyle`
- `extraData`
- `getItemLayout`
- `horizontal`
- `initialNumToRender`
- `initialScrollIndex`
- `inverted`
- `keyExtractor`
- `numColumns`
- `onEndReached`
- `onEndReachedThreshold`
- `onRefresh`
- `onViewableItemsChanged`
- `progressViewOffset`
- `legacyImplementation`
- `refreshing`
- `removeClippedSubviews`
- `viewabilityConfig`
- `viewabilityConfigCallbackPairs`

Methods

- `scrollToEnd`
- `scrollToIndex`
- `scrollToItem`
- `scrollToOffset`
- `recordInteraction`
- `flashScrollIndicators`

Reference

Props

`renderItem`

```
renderItem({ item: Object, index: number, separators: { highlight: Function, unhighlight: Function, updateProps: Function(select: string, newProps: Object) } }) => ?React.Element
```

Takes an item from `data` and renders it into the list.

Provides additional metadata like `index` if you need it, as well as a more generic `separators.updateProps` function which let you set whatever props you want to change the rendering of either the leading separator or trailing separator in case the more common `highlight` and `unhighlight` (which set the `highlighted: boolean` prop) are insufficient for your use case.

Type	Required
function	Yes

Example usage:

```
<FlatList
  ItemSeparatorComponent={Platform.OS !== 'android' && ({highlighted}) => (
    <View style={[style.separator, highlighted && {marginLeft: 0}]} />
  )}
  data={[{title: 'Title Text', key: 'item1'}]}
  renderItem={({item, separators}) => (
    <TouchableHighlight
      onPress={() => this._onPress(item)}
      onShowUnderlay={separators.highlight}
      onHideUnderlay={separators.unhighlight}>
      <View style={{backgroundColor: 'white'}}>
        <Text>{item.title}</Text>
      </View>
    </TouchableHighlight>
  )}
/>
```

data

For simplicity, data is just a plain array. If you want to use something else, like an immutable list, use the underlying `VirtualizedList` directly.

Type	Required
array	Yes

ItemSeparatorComponent

Rendered in between each item, but not at the top or bottom. By default, `highlighted` and `leadingItem` props are provided. `renderItem` provides `separators.highlight` / `unhighlight` which will update the `highlighted` prop, but you can also add custom props with `separators.updateProps`.

Type	Required
component	No

ListEmptyComponent

Rendered when the list is empty. Can be a React Component Class, a render function, or a rendered element.

Type	Required
component, function, element	No

ListFooterComponent

Rendered at the bottom of all the items. Can be a React Component Class, a render function, or a rendered element.

Type	Required
component, function, element	No

ListHeaderComponent

Rendered at the top of all the items. Can be a React Component Class, a render function, or a rendered element.

Type	Required
component, function, element	No

columnWrapperStyle

Optional custom style for multi-item rows generated when `numColumns > 1`.

Type	Required
style object	No

extraData

A marker property for telling the list to re-render (since it implements `PureComponent`). If any of your `renderItem`, Header, Footer, etc. functions depend on anything outside of the `data` prop, stick it here and treat it immutably.

Type	Required
any	No

getItemLayout

```
(data, index) => {length: number, offset: number, index: number}
```

`getItemLayout` is an optional optimization that let us skip the measurement of dynamic content if you know the height of items ahead of time. `getItemLayout` is both efficient and easy to use if you have fixed height items, for example:

```
getItemLayout={(data, index) => (
  {length: ITEM_HEIGHT, offset: ITEM_HEIGHT * index, index}
)}
```

Adding `getItemLayout` can be a great performance boost for lists of several hundred items. Remember to include separator length (height or width) in your offset calculation if you specify `ItemSeparatorComponent`.

Type	Required
function	No

horizontal

If true, renders items next to each other horizontally instead of stacked vertically.

Type	Required
boolean	No

initialNumToRender

How many items to render in the initial batch. This should be enough to fill the screen but not much more. Note these items will never be unmounted as part of the windowed rendering in order to improve perceived performance of scroll-to-top actions.

Type	Required
number	No

initialScrollIndex

Instead of starting at the top with the first item, start at `initialScrollIndex`. This disables the "scroll to top" optimization that keeps the first `initialNumToRender` items always rendered and immediately renders the items starting at this initial index. Requires `getItemLayout` to be implemented.

Type	Required
number	No

inverted

Reverses the direction of scroll. Uses scale transforms of `-1`.

Type	Required
boolean	No

keyExtractor

```
(item: object, index: number) => string;
```

Used to extract a unique key for a given item at the specified index. Key is used for caching and as the react key to track item re-ordering. The default extractor checks `item.key`, then falls back to using the index, like React does.

Type	Required
function	No

numColumns

Multiple columns can only be rendered with `horizontal={false}` and will zig-zag like a `flexWrap` layout. Items should all be the same height - masonry layouts are not supported.

Type	Required
number	No

onEndReached

```
(info: {distanceFromEnd: number}) => void
```

Called once when the scroll position gets within `onEndReachedThreshold` of the rendered content.

Type	Required
function	No

onEndReachedThreshold

How far from the end (in units of visible length of the list) the bottom edge of the list must be from the end of the content to trigger the `onEndReached` callback. Thus a value of 0.5 will trigger `onEndReached` when the end of the content is within half the visible length of the list.

Type	Required
number	No

onRefresh

```
() => void
```

If provided, a standard RefreshControl will be added for "Pull to Refresh" functionality. Make sure to also set the `refreshing` prop correctly.

Type	Required
function	No

onViewableItemsChanged

```
(info: {
  viewableItems: array,
  changed: array,
}) => void
```

Called when the viewability of rows changes, as defined by the `viewabilityConfig` prop.

Type	Required
function	No

progressViewOffset

Set this when offset is needed for the loading indicator to show correctly.

Type	Required	Platform
number	No	Android

legacyImplementation

May not have full feature parity and is meant for debugging and performance comparison.

Type	Required
boolean	No

refreshing

Set this true while waiting for new data from a refresh.

Type	Required
boolean	No

removeClippedSubviews

This may improve scroll performance for large lists.

Note: May have bugs (missing content) in some circumstances - use at your own risk.

Type	Required
boolean	No

viewabilityConfig

See `ViewabilityHelper.js` for flow type and further documentation.

Type	Required
<code>ViewabilityConfig</code>	No

`viewabilityConfig` takes a type `ViewabilityConfig` an object with following properties

Property	Required	Type
<code>minimumViewTime</code>	No	number
<code>viewAreaCoveragePercentThreshold</code>	No	number
<code>itemVisiblePercentThreshold</code>	No	number
<code>waitForInteraction</code>	No	boolean

At least one of the `viewAreaCoveragePercentThreshold` OR `itemVisiblePercentThreshold` is required. This needs to be done in the `constructor` to avoid following error ([ref](#)):

Error: Changing `viewabilityConfig` on the fly is not supported`

```
constructor (props) {
  super(props)

  this.viewabilityConfig = {
    waitForInteraction: true,
    viewAreaCoveragePercentThreshold: 95
  }
}
```

```
<FlatList
  viewabilityConfig={this.viewabilityConfig}
  ...
```

minimumViewTime

Minimum amount of time (in milliseconds) that an item must be physically viewable before the viewability callback will be fired. A high number means that scrolling through content without stopping will not mark the content as viewable.

viewAreaCoveragePercentThreshold

Percent of viewport that must be covered for a partially occluded item to count as "viewable", 0-100. Fully visible items are always considered viewable. A value of 0 means that a single pixel in the viewport makes the item viewable, and a value of 100 means that an item must be either entirely visible or cover the entire viewport to count as viewable.

itemVisiblePercentThreshold

Similar to `viewAreaPercentThreshold`, but considers the percent of the item that is visible, rather than the fraction of the viewable area it covers.

waitForInteraction

Nothing is considered viewable until the user scrolls or `recordInteraction` is called after render.

viewabilityConfigCallbackPairs

List of `ViewabilityConfig / onViewableItemsChanged` pairs. A specific `onViewableItemsChanged` will be called when its corresponding `ViewabilityConfig`'s conditions are met. See `ViewabilityHelper.js` for flow type and further documentation.

Type	Required
array of ViewabilityConfigCallbackPair	No

Methods

scrollToEnd()

```
scrollToEnd([params]);
```

Scrolls to the end of the content. May be janky without `getItemLayout` prop.

Parameters:

Name	Type	Required	Description
params	object	No	See below.

Valid `params` keys are:

- 'animated' (boolean) - Whether the list should do an animation while scrolling. Defaults to `true`.

scrollToIndex()

```
scrollToIndex(params);
```

Scrolls to the item at the specified index such that it is positioned in the viewable area such that `viewPosition` 0 places it at the top, 1 at the bottom, and 0.5 centered in the middle.

Note: Cannot scroll to locations outside the render window without specifying the `getItemLayout` prop.

Parameters:

Name	Type	Required	Description
params	object	Yes	See below.

Valid `params` keys are:

- 'animated' (boolean) - Whether the list should do an animation while scrolling. Defaults to `true`.
- 'index' (number) - The index to scroll to. Required.
- 'viewOffset' (number) - A fixed number of pixels to offset the final target position. Required.
- 'viewPosition' (number) - A value of `0` places the item specified by index at the top, `1` at the bottom, and `0.5` centered in the middle.

scrollToItem()

```
scrollToItem(params);
```

Requires linear scan through data - use `scrollToIndex` instead if possible.

Note: Cannot scroll to locations outside the render window without specifying the `getItemLayout` prop.

Parameters:

Name	Type	Required	Description
params	object	Yes	See below.

Valid `params` keys are:

- 'animated' (boolean) - Whether the list should do an animation while scrolling. Defaults to `true`.
- 'item' (object) - The item to scroll to. Required.
- 'viewPosition' (number)

scrollToOffset()

```
scrollToOffset(params);
```

Scroll to a specific content pixel offset in the list.

Parameters:

Name	Type	Required	Description
params	object	Yes	See below.

Valid `params` keys are:

- 'offset' (number) - The offset to scroll to. In case of `horizontal` being true, the offset is the x-value, in any other case the offset is the y-value. Required.
 - 'animated' (boolean) - Whether the list should do an animation while scrolling. Defaults to `true`.
-

recordInteraction()

```
recordInteraction();
```

Tells the list an interaction has occurred, which should trigger viewability calculations, e.g. if `waitForInteractions` is true and the user has not scrolled. This is typically called by taps on items or by navigation actions.

flashScrollIndicators()

```
flashScrollIndicators();
```

Displays the scroll indicators momentarily.

Image

A React component for displaying different types of images, including network images, static resources, temporary local images, and images from local disk, such as the camera roll.

This example shows fetching and displaying an image from local storage as well as one from network and even from data provided in the `'data:'` uri scheme.

Note that for network and data images, you will need to manually specify the dimensions of your image!

```
import React, { Component } from 'react';
import { AppRegistry, View, Image } from 'react-native';

export default class DisplayAnImage extends Component {
  render() {
    return (
      <View>
        <Image
          source={require('/react-native/img/favicon.png')}
        />
        <Image
          style={{width: 50, height: 50}}
          source={{uri: 'https://facebook.github.io/react-native/docs/assets/favicon.png'}}
        />
        <Image
          style={{width: 66, height: 58}}
          source={{uri: 'data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAADMAAAAzCAYAAAA6oTAqAAAEXRFWHTb2Z0d2FyZQBwbm
djcncVzaEB1SfMAABQSURBVGje7dSxCQBACARB+2/ab8BEEQNhFi6WSYzLYudDQYGBgYGBgYGBgYGBgZmcvDqYGBgmhivGQYGBgYGBgYGBgYGBgY
GBgbmQw+P/eMrC5UTVAAAABJRU5ErkJggg='}}
        />
      </View>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('DisplayAnImage', () => DisplayAnImage);
```

You can also add `style` to an image:

```
import React, { Component } from 'react';
import { AppRegistry, View, Image, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  stretch: {
    width: 50,
    height: 200
  }
});

export default class DisplayAnImageWithStyle extends Component {
  render() {
    return (
      <View>
        <Image
          style={styles.stretch}
          source={require('/react-native/img/favicon.png')}
        />
      </View>
    );
  }
}
```

```

        />
      </View>
    );
}
}

// skip these lines if using Create React Native App
AppRegistry.registerComponent(
  'DisplayAnImageWithStyle',
  () => DisplayAnImageWithStyle
);

```

GIF and WebP support on Android

When building your own native code, GIF and WebP are not supported by default on Android.

You will need to add some optional modules in `android/app/build.gradle`, depending on the needs of your app.

```

dependencies {
  // If your app supports Android versions before Ice Cream Sandwich (API level 14)
  compile 'com.facebook.fresco:animated-base-support:1.10.0'

  // For animated GIF support
  compile 'com.facebook.fresco:animated-gif:1.10.0'

  // For WebP support, including animated WebP
  compile 'com.facebook.fresco:animated-webp:1.10.0'
  compile 'com.facebook.fresco:webpsupport:1.10.0'

  // For WebP support, without animations
  compile 'com.facebook.fresco:webpsupport:1.10.0'
}

```

Props

- `style`
- `blurRadius`
- `onLayout`
- `onLoad`
- `onLoadEnd`
- `onLoadStart`
- `resizeMode`
- `source`
- `loadingIndicatorSource`
- `onError`
- `testID`
- `resizeMethod`
- `accessibilityLabel`
- `accessible`
- `capInsets`
- `defaultSource`
- `onPartialLoad`
- `onProgress`
- `fadeDuration`

Methods

- `getSize`
 - `prefetch`
 - `abortPrefetch`
 - `queryCache`
 - `resolveAssetSource`
-

Reference

Props

style

`ImageResizeMode` is an `Enum` for different image resizing modes, set via the `resizeMode` style property on `Image` components. The values are `contain`, `cover`, `stretch`, `center`, `repeat`.

Type	Required
<code>style</code>	No

- [Layout Props...](#)
- [Shadow Props...](#)
- [Transforms...](#)
- `borderTopRightRadius` : number
- `backfaceVisibility` : enum('visible', 'hidden')
- `borderBottomLeftRadius` : number
- `borderBottomRightRadius` : number
- `borderColor` : color
- `borderRadius` : number
- `borderTopLeftRadius` : number
- `backgroundColor` : color
- `borderWidth` : number
- `opacity` : number
- `overflow` : enum('visible', 'hidden')
- `resizeMode` : Object.keys(ImageResizeMode)
- `tintColor` : color

Changes the color of all the non-transparent pixels to the tintColor.

- `overlayColor` : string (*Android*)

When the image has rounded corners, specifying an `overlayColor` will cause the remaining space in the corners to be filled with a solid color. This is useful in cases which are not supported by the Android implementation of rounded corners:

- Certain resize modes, such as 'contain'
- Animated GIFs

A typical way to use this prop is with images displayed on a solid background and setting the `overlayColor` to the same color as the background.

For details of how this works under the hood, see <http://frescolib.org/docs/rounded-corners-and-circles.html>

blurRadius

`blurRadius`: the blur radius of the blur filter added to the image

Type	Required
number	No

onLayout

Invoked on mount and layout changes with `{nativeEvent: {layout: {x, y, width, height}}}`.

Type	Required
function	No

onLoad

Invoked when load completes successfully.

Type	Required
function	No

onLoadEnd

Invoked when load either succeeds or fails.

Type	Required
function	No

onLoadStart

Invoked on load start.

e.g., `onLoadStart={(e) => this.setState({loading: true})}`

Type	Required
function	No

resizeMode

Determines how to resize the image when the frame doesn't match the raw image dimensions.

- `cover` : Scale the image uniformly (maintain the image's aspect ratio) so that both dimensions (width and height) of the image will be equal to or larger than the corresponding dimension of the view (minus padding).
- `contain` : Scale the image uniformly (maintain the image's aspect ratio) so that both dimensions (width and height) of the image will be equal to or less than the corresponding dimension of the view (minus padding).
- `stretch` : Scale width and height independently. This may change the aspect ratio of the src.
- `repeat` : Repeat the image to cover the frame of the view. The image will keep its size and aspect ratio, unless it is larger than the view, in which case it will be scaled down uniformly so that it is contained in the view.
- `center` : Center the image in the view along both dimensions. If the image is larger than the view, scale it down uniformly so that it is contained in the view.

Type	Required
<code>enum('cover', 'contain', 'stretch', 'repeat', 'center')</code>	No

source

The image source (either a remote URL or a local file resource).

This prop can also contain several remote URLs, specified together with their width and height and potentially with scale/other URI arguments. The native side will then choose the best `uri` to display based on the measured size of the image container. A `cache` property can be added to control how networked request interacts with the local cache.

The currently supported formats are `png` , `jpg` , `jpeg` , `bmp` , `gif` , `webp` (Android only), `psd` (iOS only).

Type	Required
<code>ImageSourcePropType</code>	No

loadingIndicatorSource

Similarly to `source` , this property represents the resource used to render the loading indicator for the image, displayed until image is ready to be displayed, typically after when it got downloaded from network.

Type	Required
array of ImageSourcePropTypes, number	No

Can accept a number as returned by `require('./image.jpg')`

onError

Invoked on load error with `{nativeEvent: {error}}`.

Type	Required
function	No

testID

A unique identifier for this element to be used in UI Automation testing scripts.

Type	Required
string	No

resizeMethod

The mechanism that should be used to resize the image when the image's dimensions differ from the image view's dimensions. Defaults to `auto`.

- `auto` : Use heuristics to pick between `resize` and `scale`.
- `resize` : A software operation which changes the encoded image in memory before it gets decoded. This should be used instead of `scale` when the image is much larger than the view.
- `scale` : The image gets drawn downscaled or upscaled. Compared to `resize`, `scale` is faster (usually hardware accelerated) and produces higher quality images. This should be used if the image is smaller than the view. It should also be used if the image is slightly bigger than the view.

More details about `resize` and `scale` can be found at <http://frescolib.org/docs/resizing.html>.

Type	Required	Platform
enum('auto', 'resize', 'scale')	No	Android

accessibilityLabel

The text that's read by the screen reader when the user interacts with the image.

Type	Required	Platform
node	No	iOS

accessible

When true, indicates the image is an accessibility element.

Type	Required	Platform
bool	No	iOS

capInsets

When the image is resized, the corners of the size specified by `capInsets` will stay a fixed size, but the center content and borders of the image will be stretched. This is useful for creating resizable rounded buttons, shadows, and other resizable assets. More info in the [official Apple documentation](#).

Type	Required	Platform
object: {top: number, left: number, bottom: number, right: number}	No	iOS

defaultSource

A static image to display while loading the image source.

Type	Required	Platform
object, number	No	iOS
number	No	Android

If passing an object, the general shape is `{uri: string, width: number, height: number, scale: number}`:

- `uri` - a string representing the resource identifier for the image, which should be either a local file path or the name of a static image resource (which should be wrapped in the `require('./path/to/image.png')` function).
- `width`, `height` - can be specified if known at build time, in which case these will be used to set the default `<Image/>` component dimensions.
- `scale` - used to indicate the scale factor of the image. Defaults to 1.0 if unspecified, meaning that one image pixel equates to one display point / DIP.

If passing a number:

- `number` - Opaque type returned by something like `require('./image.jpg')`.

Note: On Android, the default source prop is ignored on debug builds.

onPartialLoad

Invoked when a partial load of the image is complete. The definition of what constitutes a "partial load" is loader specific though this is meant for progressive JPEG loads.

Type	Required	Platform

function	No	iOS
----------	----	-----

onProgress

Invoked on download progress with `{nativeEvent: {loaded, total}}`.

Type	Required	Platform
function	No	iOS

fadeDuration

Android only. By default, it is 300ms.

Type	Required	Platform
number	No	Android

Methods

getSize()

```
Image.getSize(uri, success, [failure]);
```

Retrieve the width and height (in pixels) of an image prior to displaying it. This method can fail if the image cannot be found, or fails to download.

In order to retrieve the image dimensions, the image may first need to be loaded or downloaded, after which it will be cached. This means that in principle you could use this method to preload images, however it is not optimized for that purpose, and may in future be implemented in a way that does not fully load/download the image data. A proper, supported way to preload images will be provided as a separate API.

Does not work for static image resources.

Parameters:

Name	Type	Required	Description
uri	string	Yes	The location of the image.
success	function	Yes	The function that will be called if the image was successfully found and width and height retrieved.
failure	function	No	The function that will be called if there was an error, such as failing to retrieve the image.

prefetch()

```
Image.prefetch(url);
```

Prefetches a remote image for later use by downloading it to the disk cache

Parameters:

Name	Type	Required	Description
url	string	Yes	The remote location of the image.

abortPrefetch()

```
Image.abortPrefetch(requestId);
```

Abort prefetch request. Android-only.

Parameters:

Name	Type	Required	Description
requestId	number	Yes	Id as returned by prefetch()

queryCache()

```
Image.queryCache(urls);
```

Perform cache interrogation. Returns a mapping from URL to cache status, such as "disk" or "memory". If a requested URL is not in the mapping, it means it's not in the cache.

Parameters:

Name	Type	Required	Description
urls	array	Yes	List of image URLs to check the cache for.

resolveAssetSource()

```
Image.resolveAssetSource(source);
```

Resolves an asset reference into an object which has the properties `uri`, `width`, and `height`.

Parameters:

Name	Type	Required	Description
source	number,	Yes	A number (opaque type returned by require('./foo.png')) or an

source	object	Yes	ImageSource .
--------	--------	-----	---------------

ImageSource is an object like { uri: '<http location || file path>' }

InputAccessoryView

A component which enables customization of the keyboard input accessory view on iOS. The input accessory view is displayed above the keyboard whenever a `TextInput` has focus. This component can be used to create custom toolbars.

To use this component wrap your custom toolbar with the `InputAccessoryView` component, and set a `nativeID`. Then, pass that `nativeID` as the `inputAccessoryViewID` of whatever `TextInput` you desire. A simple example:

```
import React, { Component } from 'react';
import { View, ScrollView, AppRegistry, TextInput, InputAccessoryView, Button } from 'react-native';

export default class UselessTextInput extends Component {
  constructor(props) {
    super(props);
    this.state = {text: 'Placeholder Text'};
  }

  render() {
    const inputAccessoryViewID = "uniqueID";
    return (
      <View>
        <ScrollView keyboardDismissMode="interactive">
          <TextInput
            style={{padding: 10, paddingTop: 50}}
            inputAccessoryViewID={inputAccessoryViewID}
            onChangeText={text => this.setState({text})}
            value={this.state.text}
          />
        </ScrollView>
        <InputAccessoryView nativeID={inputAccessoryViewID}>
          <Button
            onPress={() => this.setState({text: 'Placeholder Text'})}
            title="Reset Text"
          />
        </InputAccessoryView>
      </View>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => UselessTextInput);
```

This component can also be used to create sticky text inputs (text inputs which are anchored to the top of the keyboard). To do this, wrap a `TextInput` with the `InputAccessoryView` component, and don't set a `nativeID`. For an example, look at [InputAccessoryViewExample.js](#).

Props

- `backgroundColor`
- `nativeID`
- `style`

Reference

Props

backgroundColor

Type	Required
color	No

nativeID

An ID which is used to associate this `InputAccessoryView` to specified `TextInput`(s).

Type	Required
string	No

style

Type	Required
style	No

Known issues

- react-native#18997: Doesn't support multiline `TextInput`s
- react-native#20157: Can't use with a bottom tab bar

KeyboardAvoidingView

It is a component to solve the common problem of views that need to move out of the way of the virtual keyboard. It can automatically adjust either its position or bottom padding based on the position of the keyboard.

Example usage:

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

<KeyboardAvoidingView style={styles.container} behavior="padding" enabled>
  ... your UI ...
</KeyboardAvoidingView>
```

Example



Props

- [View props...](#)
 - `keyboardVerticalOffset`
 - `behavior`
 - `contentContainerStyle`
 - `enabled`
-

Reference

Props

keyboardVerticalOffset

This is the distance between the top of the user screen and the react native view, may be non-zero in some use cases.

Type	Required
number	No

behavior

Note: Android and iOS both interact with this prop differently. Android may behave better when given no behavior prop at all, whereas iOS is the opposite.

Type	Required
enum('height', 'position', 'padding')	No

contentContainerStyle

The style of the content container(View) when behavior is 'position'.

Type	Required
View.style	No

enabled

Enabled or disabled KeyboardAvoidingView. The default is `true`.

Type	Required

boolean

No

ListView

DEPRECATED - use one of the new list components, such as `FlatList` or `SectionList` for bounded memory use, fewer bugs, better performance, an easier to use API, and more features. Check out this [blog post](#) for more details.

`ListView` - A core component designed for efficient display of vertically scrolling lists of changing data. The minimal API is to create a `ListView.DataSource`, populate it with a simple array of data blobs, and instantiate a `ListView` component with that data source and a `renderRow` callback which takes a blob from the data array and returns a renderable component.

Minimal example:

```
class MyComponent extends Component {
  constructor() {
    super();
    const ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
    this.state = {
      dataSource: ds.cloneWithRows(['row 1', 'row 2']),
    };
  }

  render() {
    return (
      <ListView
        dataSource={this.state.dataSource}
        renderRow={(rowData) => <Text>{rowData}</Text>}
      />
    );
  }
}
```

`ListView` also supports more advanced features, including sections with sticky section headers, header and footer support, callbacks on reaching the end of the available data (`onEndReached`) and on the set of rows that are visible in the device viewport change (`onChangeVisibleRows`), and several performance optimizations.

There are a few performance operations designed to make `ListView` scroll smoothly while dynamically loading potentially very large (or conceptually infinite) data sets:

- Only re-render changed rows - the `rowHasChanged` function provided to the data source tells the `ListView` if it needs to re-render a row because the source data has changed - see `ListViewDataSource` for more details.
- Rate-limited row rendering - By default, only one row is rendered per event-loop (customizable with the `pageSize` prop). This breaks up the work into smaller chunks to reduce the chance of dropping frames while rendering rows.

Props

- [ScrollView props...](#)
- `dataSource`
- `initialListSize`
- `onEndReachedThreshold`

- `pageSize`
- `renderRow`
- `renderScrollIndicator`
- `scrollRenderAheadDistance`
- `stickyHeaderIndices`
- `enableEmptySections`
- `renderHeader`
- `onEndReached`
- `stickySectionHeadersEnabled`
- `renderSectionHeader`
- `renderSeparator`
- `onChangeVisibleRows`
- `removeClippedSubviews`
- `renderFooter`

Methods

- `getMetrics`
 - `scrollTo`
 - `scrollToEnd`
 - `flashScrollIndicator`
-

Reference

Props

`dataSource`

An instance of [ListView.DataSource](#) to use

Type	Required
<code>ListViewDataSource</code>	Yes

`initialListSize`

How many rows to render on initial component mount. Use this so that the first screen worth of data appears at one time instead of over the course of multiple frames.

Type	Required
<code>number</code>	Yes

`onEndReachedThreshold`

Threshold in pixels (virtual, not physical) for calling onEndReached.

Type	Required
number	Yes

pageSize

Number of rows to render per event loop. Note: if your 'rows' are actually cells, i.e. they don't span the full width of your view (as in the ListViewGridLayoutExample), you should set the pageSize to be a multiple of the number of cells per row, otherwise you're likely to see gaps at the edge of the ListView as new pages are loaded.

Type	Required
number	Yes

renderRow

(rowData, sectionID, rowID, highlightRow) => renderable

Takes a data entry from the data source and its ids and should return a renderable component to be rendered as the row. By default the data is exactly what was put into the data source, but it's also possible to provide custom extractors. ListView can be notified when a row is being highlighted by calling `highlightRow(sectionID, rowID)`. This sets a boolean value of adjacentRowHighlighted in renderSeparator, allowing you to control the separators above and below the highlighted row. The highlighted state of a row can be reset by calling `highlightRow(null)`.

Type	Required
function	Yes

renderScrollIndicator

(props) => renderable

A function that returns the scrollable component in which the list rows are rendered. Defaults to returning a ScrollView with the given props.

Type	Required
function	Yes

scrollRenderAheadDistance

How early to start rendering rows before they come on screen, in pixels.

Type	Required
number	Yes

stickyHeaderIndices

An array of child indices determining which children get docked to the top of the screen when scrolling. For example, passing `stickyHeaderIndices={[0]}` will cause the first child to be fixed to the top of the scroll view. This property is not supported in conjunction with `horizontal={true}`.

Type	Required
array of number	Yes

enableEmptySections

Flag indicating whether empty section headers should be rendered. In the future release empty section headers will be rendered by default, and the flag will be deprecated. If empty sections are not desired to be rendered their indices should be excluded from sectionID object.

Type	Required
bool	No

renderHeader

Type	Required
function	No

onEndReached

Called when all rows have been rendered and the list has been scrolled to within onEndReachedThreshold of the bottom. The native scroll event is provided.

Type	Required
function	No

stickySectionHeadersEnabled

Makes the sections headers sticky. The sticky behavior means that it will scroll with the content at the top of the section until it reaches the top of the screen, at which point it will stick to the top until it is pushed off the screen by the next section header. This property is not supported in conjunction with `horizontal={true}`. Only enabled by default on iOS because of typical platform standards.

Type	Required
bool	No

renderSectionHeader

(sectionData, sectionID) => renderable

If provided, a header is rendered for this section.

Type	Required
function	No

renderSeparator

(sectionID, rowID, adjacentRowHighlighted) => renderable

If provided, a renderable component to be rendered as the separator below each row but not the last row if there is a section header below. Take a sectionID and rowID of the row above and whether its adjacent row is highlighted.

Type	Required
function	No

onChangeVisibleRows

(visibleRows, changedRows) => void

Called when the set of visible rows changes. `visibleRows` maps { sectionID: { rowID: true }} for all the visible rows, and `changedRows` maps { sectionID: { rowID: true | false }} for the rows that have changed their visibility, with true indicating visible, and false indicating the view has moved out of view.

Type	Required
function	No

removeClippedSubviews

A performance optimization for improving scroll perf of large lists, used in conjunction with overflow: 'hidden' on the row containers. This is enabled by default.

Type	Required
bool	No

renderFooter

() => renderable

The header and footer are always rendered (if these props are provided) on every render pass. If they are expensive to re-render, wrap them in StaticContainer or other mechanism as appropriate. Footer is always at the bottom of the list, and header at the top, on every render pass. In a horizontal ListView, the header is rendered on the left and the footer on the right.

Type	Required
function	No

Methods

getMetrics()

```
getMetrics();
```

Exports some data, e.g. for perf investigations or analytics.

scrollTo()

```
scrollTo(...args: Array)
```

Scrolls to a given x, y offset, either immediately or with a smooth animation.

See [ScrollView#scrollTo](#).

scrollToEnd()

```
scrollToEnd(([options]: object));
```

If this is a vertical ListView scrolls to the bottom. If this is a horizontal ListView scrolls to the right.

Use `scrollToEnd({animated: true})` for smooth animated scrolling, `scrollToEnd({animated: false})` for immediate scrolling. If no options are passed, `animated` defaults to true.

See [ScrollView#scrollToEnd](#).

flashScrollIndicators()

```
flashScrollIndicators();
```

Displays the scroll indicators momentarily.

MaskedViewIOS

Renders the child view with a mask specified in the `maskElement` prop.

Example

```
import React from 'react';
import { MaskedViewIOS, Text, View } from 'react-native';

class MyMaskedView extends React.Component {
  render() {
    return (
      // Determines shape of the mask
      <MaskedViewIOS
        style={{ flex: 1, flexDirection: 'row', height: '100%' }}
        maskElement={
          <View style={{
            // Transparent background because mask is based off alpha channel.
            backgroundColor: 'transparent',
            flex: 1,
            justifyContent: 'center',
            alignItems: 'center',
          }}>
            <Text style={{
              fontSize: 60,
              color: 'black',
              fontWeight: 'bold',
            }}>
              Basic Mask
            </Text>
          </View>
        }
      >
        { /* Shows behind the mask, you can put anything here, such as an image */ }
        <View style={{ flex: 1, height: '100%', backgroundColor: '#324376' }} />
        <View style={{ flex: 1, height: '100%', backgroundColor: '#F5DD90' }} />
        <View style={{ flex: 1, height: '100%', backgroundColor: '#F76C5E' }} />
      </MaskedViewIOS>
    );
  }
}
```

The following image demonstrates that you can put almost anything behind the mask. The three examples shown are masked `<View>` , `<Text>` , and `<Image>` .

Basic Mask

ext so I put some text. It's text you like. You need ext in your ext header like this: `put("text", "you like text", so)`.
It's text you like text. It's me ext. You liked text so I put some text? You text. I h

ite. It put some text. It's text you like. You need ext in your ext header like text so. It's text you like text so. It's hard to read exts, sometimes you need exts. I put some text in our ext.

Basic Mask

The alpha channel of the view rendered by the `maskElement` prop determines how much of the view's content and background shows through. Fully or partially opaque pixels allow the underlying content to show through but fully transparent pixels block that content.

Props

- [View props...](#)
 - `maskElement`
-

Reference

Props

maskElement

Type	Required
element	Yes

Modal

The Modal component is a simple way to present content above an enclosing view.

Note: If you need more control over how to present modals over the rest of your app, then consider using a top-level Navigator.

```
import React, {Component} from 'react';
import {Modal, Text, TouchableHighlight, View, Alert} from 'react-native';

class ModalExample extends Component {
  state = {
    modalVisible: false,
  };

  setModalVisible(visible) {
    this.setState({modalVisible: visible});
  }

  render() {
    return (
      <View style={{marginTop: 22}}>
        <Modal
          animationType="slide"
          transparent={false}
          visible={this.state.modalVisible}
          onRequestClose={() => {
            Alert.alert('Modal has been closed.');
          }}>
          <View style={{marginTop: 22}}>
            <View>
              <Text>Hello World!</Text>

              <TouchableHighlight
                onPress={() => {
                  this.setModalVisible(!this.state.modalVisible);
                }}>
                <Text>Hide Modal</Text>
              </TouchableHighlight>
            </View>
          </View>
        </Modal>

        <TouchableHighlight
          onPress={() => {
            this.setModalVisible(true);
          }}>
          <Text>Show Modal</Text>
        </TouchableHighlight>
      </View>
    );
  }
}
```

Props

- `visible`
- `supportedOrientations`

- `onRequestClose`
 - `onShow`
 - `transparent`
 - `animationType`
 - `hardwareAccelerated`
 - `onDismiss`
 - `onOrientationChange`
 - `presentationStyle`
 - `animated`
-

Reference

Props

visible

The `visible` prop determines whether your modal is visible.

Type	Required
bool	No

supportedOrientations

The `supportedOrientations` prop allows the modal to be rotated to any of the specified orientations. On iOS, the modal is still restricted by what's specified in your app's Info.plist's `UIInterfaceOrientation` field. When using `presentationStyle` of `pageSheet` or `formSheet`, this property will be ignored by iOS.

Type	Required	Platform
array of enum('portrait', 'portrait-upside-down', 'landscape', 'landscape-left', 'landscape-right')	No	iOS

onRequestClose

The `onRequestClose` callback is called when the user taps the hardware back button on Android or the menu button on Apple TV. Because of this required prop, be aware that `BackHandler` events will not be emitted as long as the modal is open.

Type	Required	Platform
function	Yes	Android, Platform.isTVOS
function	No	(Others)

onShow

The `onShow` prop allows passing a function that will be called once the modal has been shown.

Type	Required
function	No

transparent

The `transparent` prop determines whether your modal will fill the entire view. Setting this to `true` will render the modal over a transparent background.

Type	Required
bool	No

animationType

The `animationType` prop controls how the modal animates.

- `slide` slides in from the bottom
- `fade` fades into view
- `none` appears without an animation

Default is set to `none`.

Type	Required
enum('none', 'slide', 'fade')	No

hardwareAccelerated

The `hardwareAccelerated` prop controls whether to force hardware acceleration for the underlying window.

Type	Required	Platform
bool	No	Android

onDismiss

The `onDismiss` prop allows passing a function that will be called once the modal has been dismissed.

Type	Required	Platform
function	No	iOS

onOrientationChange

The `onOrientationChange` callback is called when the orientation changes while the modal is being displayed. The orientation provided is only 'portrait' or 'landscape'. This callback is also called on initial render, regardless of the current orientation.

Type	Required	Platform
function	No	iOS

presentationStyle

The `presentationStyle` prop controls how the modal appears (generally on larger devices such as iPad or plus-sized iPhones). See <https://developer.apple.com/reference/uikit/uimodalpresentationstyle> for details.

- `fullScreen` covers the screen completely
- `pageSheet` covers portrait-width view centered (only on larger devices)
- `formSheet` covers narrow-width view centered (only on larger devices)
- `overFullScreen` covers the screen completely, but allows transparency

Default is set to `overFullScreen` OR `fullScreen` depending on `transparent` property.

Type	Required	Platform
<code>enum('fullScreen', 'pageSheet', 'formSheet', 'overFullScreen')</code>	No	iOS

animated

Deprecated. Use the `animationType` prop instead.

NavigatorIOS

`NavigatorIOS` is a wrapper around `UINavigationController`, enabling you to implement a navigation stack. It works exactly the same as it would on a native app using `UINavigationController`, providing the same animations and behavior from UIKit.

As the name implies, it is only available on iOS. Take a look at [React Navigation](#) for a cross-platform solution in JavaScript, or check out either of these components for native solutions: [native-navigation](#), [react-native-navigation](#).

To set up the navigator, provide the `initialRoute` prop with a route object. A route object is used to describe each scene that your app navigates to. `initialRoute` represents the first route in your navigator.

```
import PropTypes from 'prop-types';
import React, { Component } from 'react';
import { NavigatorIOS, Text, TouchableHighlight, View } from 'react-native';

export default class NavigatorIOSApp extends Component {
  render() {
    return (
      <NavigatorIOS
        initialRoute={{
          component: MyScene,
          title: 'My Initial Scene',
        }}
        style={{flex: 1}}
      />
    );
  }
}

class MyScene extends Component {
  static propTypes = {
    title: PropTypes.string.isRequired,
    navigator: PropTypes.object.isRequired,
  }

  _onForward = () => {
    this.props.navigator.push({
      title: 'Scene',
    });
  }

  render() {
    return (
      <View>
        <Text>Current Scene: { this.props.title }</Text>
        <TouchableHighlight onPress={this._onForward}>
          <Text>Tap me to load the next scene</Text>
        </TouchableHighlight>
      </View>
    );
  }
}
```

In this code, the navigator renders the component specified in `initialRoute`, which in this case is `MyScene`. This component will receive a `route` prop and a `navigator` prop representing the navigator. The navigator's navigation bar will render the title for the current scene, "My Initial Scene".

You can optionally pass in a `passProps` property to your `initialRoute`. `NavigatorIOS` passes this in as props to the rendered component:

```
initialRoute={{
  component: MyScene,
  title: 'My Initial Scene',
  passProps: { myProp: 'foo' }
}}
```

You can then access the props passed in via `{this.props.myProp}`.

Handling Navigation

To trigger navigation functionality such as pushing or popping a view, you have access to a `navigator` object. The object is passed in as a prop to any component that is rendered by `NavigatorIOS`. You can then call the relevant methods to perform the navigation action you need:

```
class MyView extends Component {
  _handleBackPress() {
    this.props.navigator.pop();
  }

  _handleNextPress(nextRoute) {
    this.props.navigator.push(nextRoute);
  }

  render() {
    const nextRoute = {
      component: MyView,
      title: 'Bar That',
      passProps: { myProp: 'bar' }
    };
    return(
      <TouchableHighlight onPress={() => this._handleNextPress(nextRoute)}>
        <Text style={{marginTop: 200, alignSelf: 'center'}}>
          See you on the other nav {this.props.myProp}!
        </Text>
      </TouchableHighlight>
    );
  }
}
```

You can also trigger navigator functionality from the `NavigatorIOS` component:

```
class NavvyIOS extends Component {
  _handleNavigationRequest() {
    this.refs.nav.push({
      component: MyView,
      title: 'Genius',
      passProps: { myProp: 'genius' },
    });
  }
}
```

```

render() {
  return (
    <NavigatorIOS
      ref='nav'
      initialRoute={{
        component: MyView,
        title: 'Foo This',
        passProps: { myProp: 'foo' },
        rightButtonTitle: 'Add',
        onRightButtonPress: () => this._handleNavigationRequest(),
      }}
      style={{flex: 1}}
    />
  );
}

```

The code above adds a `_handleNavigationRequest` private method that is invoked from the `NavigatorIOS` component when the right navigation bar item is pressed. To get access to the navigator functionality, a reference to it is saved in the `ref` prop and later referenced to push a new scene into the navigation stack.

Navigation Bar Configuration

Props passed to `NavigatorIOS` will set the default configuration for the navigation bar. Props passed as properties to a route object will set the configuration for that route's navigation bar, overriding any props passed to the `NavigatorIOS` component.

```

_handleNavigationRequest() {
  this.refs.nav.push({
    //...
    passProps: { myProp: 'genius' },
    barTintColor: '#996699',
  });
}

render() {
  return (
    <NavigatorIOS
      //...
      style={{flex: 1}}
      barTintColor='#ffffcc'
    />
  );
}

```

In the example above the navigation bar color is changed when the new route is pushed.

Props

- `initialRoute`
- `barStyle`
- `barTintColor`
- `interactivePopGestureEnabled`
- `itemWrapperStyle`
- `navigationBarHidden`

- `shadowHidden`
- `tintColor`
- `titleTextColor`
- `translucent`

Methods

- `push`
 - `popN`
 - `pop`
 - `replaceAtIndex`
 - `replace`
 - `replacePrevious`
 - `popToTop`
 - `popToRoute`
 - `replacePreviousAndPop`
 - `resetTo`
-

Reference

Props

`initialRoute`

NavigatorIOS uses `route` objects to identify child views, their props, and navigation bar configuration. Navigation operations such as push operations expect routes to look like this the `initialRoute`.

Type	Required
<code>object: {component: function,title: string,titleImage: Image.propTypes.source,passProps: object,backButtonIcon: Image.propTypes.source,backButtonTitles: string,leftButtonIcon: Image.propTypes.source,leftButtonTitles: string,leftButtonSystemIcon: Object.keys(SystemIcons),onLeftButtonPress: function,rightButtonIcon: Image.propTypes.source,rightButtonTitles: string,rightButtonSystemIcon: Object.keys(SystemIcons),onRightButtonPress: function,wrapperStyle: View.style,navigationBarHidden: bool,shadowHidden: bool,tintColor: string,barTintColor: string,barStyle: enum('default', 'black'),titleTextColor: string,translucent: bool}</code>	Yes

`barStyle`

The style of the navigation bar. Supported values are 'default', 'black'. Use 'black' instead of setting `barTintColor` to black. This produces a navigation bar with the native iOS style with higher translucency.

Type	Required
<code>enum('default', 'black')</code>	No

barTintColor

The default background color of the navigation bar.

Type	Required
string	No

interactivePopGestureEnabled

Boolean value that indicates whether the interactive pop gesture is enabled. This is useful for enabling/disabling the back swipe navigation gesture.

If this prop is not provided, the default behavior is for the back swipe gesture to be enabled when the navigation bar is shown and disabled when the navigation bar is hidden. Once you've provided the `interactivePopGestureEnabled` prop, you can never restore the default behavior.

Type	Required
bool	No

itemWrapperStyle

The default wrapper style for components in the navigator. A common use case is to set the `backgroundColor` for every scene.

Type	Required
<code>View.style</code>	No

navigationBarHidden

Boolean value that indicates whether the navigation bar is hidden by default.

Type	Required
bool	No

shadowHidden

Boolean value that indicates whether to hide the 1px hairline shadow by default.

Type	Required
bool	No

tintColor

The default color used for the buttons in the navigation bar.

Type	Required
string	No

titleTextColor

The default text color of the navigation bar title.

Type	Required
string	No

translucent

Boolean value that indicates whether the navigation bar is translucent by default. When true any screens loaded within the navigator will sit below the status bar and underneath the navigation bar. To have screens render below the navigation bar set to false.

Type	Required
bool	No

Methods

push()

```
push((route: object));
```

Navigate forward to a new route.

Parameters:

Name	Type	Required	Description
route	object	Yes	The new route to navigate to.

popN()

```
popN((n: number));
```

Go back N scenes at once. When N=1, behavior matches `pop()`.

Parameters:

Name	Type	Required	Description
n	number	Yes	The number of scenes to pop.

pop()

```
pop();
```

Pop back to the previous scene.

replaceAtIndex()

```
replaceAtIndex((route: object), (index: number));
```

Replace a route in the navigation stack.

Parameters:

Name	Type	Required	Description
route	object	Yes	The new route that will replace the specified one.
index	number	Yes	The route into the stack that should be replaced. If it is negative, it counts from the back of the stack.

replace()

```
replace((route: object));
```

Replace the route for the current scene and immediately load the view for the new route.

Parameters:

Name	Type	Required	Description
route	object	Yes	The new route to navigate to.

replacePrevious()

```
replacePrevious((route: object));
```

Replace the route/view for the previous scene.

Parameters:

Name	Type	Required	Description
route	object	Yes	The new route to will replace the previous scene.

popToTop()

```
popToTop();
```

Go back to the topmost item in the navigation stack.

popToRoute()

```
popToRoute((route: object));
```

Go back to the item for a particular route object.

Parameters:

Name	Type	Required	Description
route	object	Yes	The new route to navigate to.

replacePreviousAndPop()

```
replacePreviousAndPop((route: object));
```

Replaces the previous route/view and transitions back to it.

Parameters:

Name	Type	Required	Description
route	object	Yes	The new route that replaces the previous scene.

resetTo()

```
resetTo((route: object));
```

Replaces the top item and pop to it.

Parameters:

Name	Type	Required	Description
route	object	Yes	The new route that will replace the topmost item.

Picker

Renders the native picker component on iOS and Android. Example:

```
<Picker  
  selectedValue={this.state.language}  
  style={{ height: 50, width: 100 }}  
  onValueChange={(itemValue, itemIndex) => this.setState({language: itemValue})}>  
  <Picker.Item label="Java" value="java" />  
  <Picker.Item label="JavaScript" value="js" />  
</Picker>
```

Props

- [View props...](#)
 - `onValueChange`
 - `selectedValue`
 - `style`
 - `testID`
 - `enabled`
 - `mode`
 - `prompt`
 - `itemStyle`
-

Reference

Props

onValueChange

Callback for when an item is selected. This is called with the following parameters:

- `itemValue` : the `value` prop of the item that was selected
- `itemPosition` : the index of the selected item in this picker

Type	Required
function	No

selectedValue

Value matching value of one of the items. Can be a string or an integer.

Type	Required
------	----------

any	No
-----	----

style

Type	Required
pickerStyleType	No

testID

Used to locate this view in end-to-end tests.

Type	Required
string	No

enabled

If set to false, the picker will be disabled, i.e. the user will not be able to make a selection.

Type	Required	Platform
bool	No	Android

mode

On Android, specifies how to display the selection items when the user taps on the picker:

- 'dialog': Show a modal dialog. This is the default.
- 'dropdown': Shows a dropdown anchored to the picker view

Type	Required	Platform
enum('dialog', 'dropdown')	No	Android

prompt

Prompt string for this picker, used on Android in dialog mode as the title of the dialog.

Type	Required	Platform
string	No	Android

itemStyle

Style to apply to each of the item labels.

Type	Required	Platform
text styles	No	iOS

PickerIOS

Props

- [View props...](#)
 - `itemStyle`
 - `onValueChange`
 - `selectedValue`
-

Reference

Props

`itemStyle`

Type	Required
<code>text styles</code>	No

`onValueChange`

Type	Required
<code>function</code>	No

`selectedValue`

Type	Required
<code>any</code>	No

ProgressBarAndroid

Android-only React component used to indicate that the app is loading or there is some activity in the app.

Example:

```
import React, { Component } from "react";
import {
  ProgressBarAndroid,
  AppRegistry,
  StyleSheet,
  View
} from "react-native";

export default class App extends Component {
  render() {
    return (
      <View style={styles.container}>
        <ProgressBarAndroid />
        <ProgressBarAndroid styleAttr="Horizontal" />
        <ProgressBarAndroid styleAttr="Horizontal" color="#2196F3" />
        <ProgressBarAndroid
          styleAttr="Horizontal"
          indeterminate={false}
          progress={0.5}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "space-evenly",
    padding: 10
  }
});

AppRegistry.registerComponent("App", () => App);
```

Props

- [View props...](#)
- [animating](#)
- [color](#)
- [indeterminate](#)
- [progress](#)
- [styleAttr](#)
- [testID](#)

Reference

Props

animating

Whether to show the ProgressBar (true, the default) or hide it (false).

Type	Required
bool	No

color

Color of the progress bar.

Type	Required
color	No

indeterminate

If the progress bar will show indeterminate progress. Note that this can only be false if styleAttr is Horizontal.

Type	Required
indeterminateType	No

progress

The progress value (between 0 and 1).

Type	Required
number	No

styleAttr

Style of the ProgressBar. One of:

- Horizontal
- Normal (default)
- Small
- Large
- Inverse
- SmallInverse
- LargeInverse

Type	Required

```
enum('Horizontal', 'Normal', 'Small', 'Large', 'Inverse', 'SmallInverse', 'LargeInverse')
```

```
No
```

testID

Used to locate this view in end-to-end tests.

Type	Required
string	No

ProgressViewIOS

Use `ProgressViewIOS` to render a `UIProgressView` on iOS.

Props

- [View props...](#)
 - `progress`
 - `progressImage`
 - `progressTintColor`
 - `progressViewStyle`
 - `trackImage`
 - `trackTintColor`
-

Reference

Props

progress

The progress value (between 0 and 1).

Type	Required
number	No

progressImage

A stretchable image to display as the progress bar.

Type	Required
<code>Image.propTypes.source</code>	No

progressTintColor

The tint color of the progress bar itself.

Type	Required
string	No

progressViewStyle

The progress bar style.

Type	Required
enum('default', 'bar')	No

trackImage

A stretchable image to display behind the progress bar.

Type	Required
Image.propTypes.source	No

trackTintColor

The tint color of the progress bar track.

Type	Required
string	No

RefreshControl

This component is used inside a ScrollView or ListView to add pull to refresh functionality. When the ScrollView is at `scrollY: 0`, swiping down triggers an `onRefresh` event.

Usage example

```
class RefreshableList extends Component {
  constructor(props) {
    super(props);
    this.state = {
      refreshing: false,
    };
  }

  _onRefresh = () => {
    this.setState({refreshing: true});
    fetchData().then(() => {
      this.setState({refreshing: false});
    });
  }

  render() {
    return (
      <ScrollView
        refreshControl={
          <RefreshControl
            refreshing={this.state.refreshing}
            onRefresh={this._onRefresh}
          />
        }
      >
      ...
    );
  }
}
```

Note: `refreshing` is a controlled prop, this is why it needs to be set to true in the `onRefresh` function otherwise the refresh indicator will stop immediately.

Props

- [View props...](#)
- `refreshing`
- `onRefresh`
- `colors`
- `enabled`
- `progressBackgroundColor`
- `progressViewOffset`
- `size`
- `tintColor`

- `title`
 - `titleColor`
-

Reference

Props

refreshing

Whether the view should be indicating an active refresh.

Type	Required
bool	Yes

onRefresh

Called when the view starts refreshing.

Type	Required
function	No

colors

The colors (at least one) that will be used to draw the refresh indicator.

Type	Required	Platform
array of <code>color</code>	No	Android

enabled

Whether the pull to refresh functionality is enabled.

Type	Required	Platform
bool	No	Android

progressBackgroundColor

The background color of the refresh indicator.

Type	Required	Platform

color	No	Android
-------	----	---------

progressViewOffset

Progress view top offset

Type	Required	Platform
number	No	Android

size

Size of the refresh indicator, see RefreshControl.SIZE.

Type	Required	Platform
enum(RefreshLayoutConsts.SIZE.DEFAULT, RefreshLayoutConsts.SIZE.LARGE)	No	Android

tintColor

The color of the refresh indicator.

Type	Required	Platform
color	No	iOS

title

The title displayed under the refresh indicator.

Type	Required	Platform
string	No	iOS

titleColor

Title color.

Type	Required	Platform
color	No	iOS

SafeAreaView

The purpose of `SafeAreaView` is to render content within the safe area boundaries of a device. It is currently only applicable to iOS devices with iOS version 11 or later.

`SafeAreaView` renders nested content and automatically applies paddings to reflect the portion of the view that is not covered by navigation bars, tab bars, toolbars, and other ancestor views. Moreover, and most importantly, Safe Area's paddings reflect the physical limitation of the screen, such as rounded corners or camera notches (i.e. the sensor housing area on iPhone X).

Usage Example

Simply wrap your top level view with a `SafeAreaView` with a `flex: 1` style applied to it. You may also want to use a background color that matches your application's design.

```
<SafeAreaView style={{flex: 1, backgroundColor: '#fff'}}>
  <View style={{flex: 1}}>
    <Text>Hello World!</Text>
  </View>
</SafeAreaView>
```

Props

- [View props...](#)

ScrollView

Component that wraps platform ScrollView while providing integration with touch locking "responder" system.

Keep in mind that ScrollViews must have a bounded height in order to work, since they contain unbounded-height children into a bounded container (via a scroll interaction). In order to bound the height of a ScrollView, either set the height of the view directly (discouraged) or make sure all parent views have bounded height. Forgetting to transfer `{flex: 1}` down the view stack can lead to errors here, which the element inspector makes easy to debug.

Doesn't yet support other contained responders from blocking this scroll view from becoming the responder.

`<ScrollView>` vs `<FlatList>` - which one to use?

`ScrollView` simply renders all its react child components at once. That makes it very easy to understand and use.

On the other hand, this has a performance downside. Imagine you have a very long list of items you want to display, maybe several screens worth of content. Creating JS components and native views for everything all at once, much of which may not even be shown, will contribute to slow rendering and increased memory usage.

This is where `FlatList` comes into play. `FlatList` renders items lazily, just when they are about to appear, and removes items that scroll way off screen to save memory and processing time.

`FlatList` is also handy if you want to render separators between your items, multiple columns, infinite scroll loading, or any number of other features it supports out of the box.

Props

- [View props...](#)
- `alwaysBounceVertical`
- `contentContainerStyle`
- `keyboardDismissMode`
- `keyboardShouldPersistTaps`
- `onContentSizeChange`
- `onMomentumScrollBegin`
- `onMomentumScrollEnd`
- `onScroll`
- `onScrollBeginDrag`
- `onScrollEndDrag`
- `pagingEnabled`
- `refreshControl`
- `removeClippedSubviews`
- `scrollEnabled`
- `showsHorizontalScrollIndicator`
- `showsVerticalScrollIndicator`
- `stickyHeaderIndices`
- `endFillColor`
- `overScrollMode`
- `scrollPerfTag`

- `DEPRECATED_sendUpdatedChildFrames`
- `alwaysBounceHorizontal`
- `horizontal`
- `automaticallyAdjustContentInsets`
- `bounces`
- `bouncesZoom`
- `canCancelContentTouches`
- `centerContent`
- `contentInset`
- `contentInsetAdjustmentBehavior`
- `contentOffset`
- `decelerationRate`
- `directionalLockEnabled`
- `indicatorStyle`
- `maximumZoomScale`
- `minimumZoomScale`
- `pinchGestureEnabled`
- `scrollEventThrottle`
- `scrollIndicatorInsets`
- `scrollsToTop`
- `snapToAlignment`
- `snapToInterval`
- `snapToOffsets`
- `snapToStart`
- `snapToEnd`
- `zoomScale`
- `nestedScrollEnabled`

Methods

- `scrollTo`
 - `scrollToEnd`
 - `scrollWithoutAnimationTo`
 - `flashScrollIndicators`
-

Reference

Props

`alwaysBounceVertical`

When true, the scroll view bounces vertically when it reaches the end even if the content is smaller than the scroll view itself. The default value is false when `horizontal={true}` and true otherwise.

Type	Required	Platform
<code>bool</code>	No	iOS

contentContainerStyle

These styles will be applied to the scroll view content container which wraps all of the child views. Example:

```
return (
  <ScrollView contentContainerStyle={styles.contentContainer}>
    </ScrollView>
);
...
const styles = StyleSheet.create({
  contentContainer: {
    paddingVertical: 20
  }
});
```

Type	Required
StyleSheetPropType(View Style props)	No

keyboardDismissMode

Determines whether the keyboard gets dismissed in response to a drag.

Cross platform

- 'none' (the default), drags do not dismiss the keyboard.
- 'on-drag' , the keyboard is dismissed when a drag begins.

iOS Only

- 'interactive' , the keyboard is dismissed interactively with the drag and moves in synchrony with the touch; dragging upwards cancels the dismissal. On android this is not supported and it will have the same behavior as 'none'.

Type	Required
enum('none', 'on-drag', 'interactive')	No

keyboardShouldPersistTaps

Determines when the keyboard should stay visible after a tap.

- 'never' (the default), tapping outside of the focused text input when the keyboard is up dismisses the keyboard. When this happens, children won't receive the tap.
- 'always' , the keyboard will not dismiss automatically, and the scroll view will not catch taps, but children of the scroll view can catch taps.
- 'handled' , the keyboard will not dismiss automatically when the tap was handled by a children, (or captured by an ancestor).
- ~~false~~ , deprecated, use 'never' instead

- `true`, deprecated, use 'always' instead

Type	Required
<code>enum('always', 'never', 'handled', false, true)</code>	No

onContentSizeChange

Called when scrollable content view of the ScrollView changes.

Handler function is passed the content width and content height as parameters: `(contentWidth, contentHeight)`

It's implemented using onLayout handler attached to the content container which this ScrollView renders.

Type	Required
<code>function</code>	No

onMomentumScrollBegin

Called when the momentum scroll starts (scroll which occurs as the ScrollView glides to a stop).

Type	Required
<code>function</code>	No

onMomentumScrollEnd

Called when the momentum scroll ends (scroll which occurs as the ScrollView glides to a stop).

Type	Required
<code>function</code>	No

onScroll

Fires at most once per frame during scrolling. The frequency of the events can be controlled using the `scrollEventThrottle` prop.

Type	Required
<code>function</code>	No

onScrollBeginDrag

Called when the user begins to drag the scroll view.

Type	Required

function	No
----------	----

onScrollEndDrag

Called when the user stops dragging the scroll view and it either stops or begins to glide.

Type	Required
function	No

pagingEnabled

When true, the scroll view stops on multiples of the scroll view's size when scrolling. This can be used for horizontal pagination. The default value is false.

Note: Vertical pagination is not supported on Android.

Type	Required
bool	No

refreshControl

A RefreshControl component, used to provide pull-to-refresh functionality for the ScrollView. Only works for vertical ScrollViews (`horizontal` prop must be `false`).

See [RefreshControl](#).

Type	Required
element	No

removeClippedSubviews

Experimental: When true, offscreen child views (whose `overflow` value is `hidden`) are removed from their native backing superview when offscreen. This can improve scrolling performance on long lists. The default value is true.

Type	Required
bool	No

scrollEnabled

When false, the view cannot be scrolled via touch interaction. The default value is true.

Note that the view can always be scrolled by calling `scrollTo`.

Type	Required
bool	No

showsHorizontalScrollIndicator

When true, shows a horizontal scroll indicator. The default value is true.

Type	Required
bool	No

showsVerticalScrollIndicator

When true, shows a vertical scroll indicator. The default value is true.

Type	Required
bool	No

stickyHeaderIndices

An array of child indices determining which children get docked to the top of the screen when scrolling. For example, passing `stickyHeaderIndices={[0]}` will cause the first child to be fixed to the top of the scroll view. This property is not supported in conjunction with `horizontal={true}`.

Type	Required
array of number	No

endFillColor

Sometimes a scrollview takes up more space than its content fills. When this is the case, this prop will fill the rest of the scrollview with a color to avoid setting a background and creating unnecessary overdraw. This is an advanced optimization that is not needed in the general case.

Type	Required	Platform
color	No	Android

overScrollMode

Used to override default value of overScroll mode.

Possible values:

- `'auto'` - Default value, allow a user to over-scroll this view only if the content is large enough to meaningfully scroll.
- `'always'` - Always allow a user to over-scroll this view.
- `'never'` - Never allow a user to over-scroll this view.

Type	Required	Platform
<code>enum('auto', 'always', 'never')</code>	No	Android

scrollPerfTag

Tag used to log scroll performance on this scroll view. Will force momentum events to be turned on (see `sendMomentumEvents`). This doesn't do anything out of the box and you need to implement a custom native `FpsListener` for it to be useful.

Type	Required	Platform
<code>string</code>	No	Android

DEPRECATED_sendUpdatedChildFrames

When true, `ScrollView` will emit `updateChildFrames` data in scroll events, otherwise will not compute or emit child frame data. This only exists to support legacy issues, `onLayout` should be used instead to retrieve frame data. The default value is false.

Type	Required	Platform
<code>bool</code>	No	iOS

alwaysBounceHorizontal

When true, the scroll view bounces horizontally when it reaches the end even if the content is smaller than the scroll view itself. The default value is true when `horizontal={true}` and false otherwise.

Type	Required	Platform
<code>bool</code>	No	iOS

horizontal

When true, the scroll view's children are arranged horizontally in a row instead of vertically in a column. The default value is false.

Type	Required
<code>bool</code>	No

automaticallyAdjustContentInsets

Controls whether iOS should automatically adjust the content inset for scroll views that are placed behind a navigation bar or tab bar/ toolbar. The default value is true.

Type	Required	Platform
bool	No	iOS

bounces

When true, the scroll view bounces when it reaches the end of the content if the content is larger than the scroll view along the axis of the scroll direction. When false, it disables all bouncing even if the `alwaysBounce*` props are true. The default value is true.

Type	Required	Platform
bool	No	iOS

bouncesZoom

When true, gestures can drive zoom past min/max and the zoom will animate to the min/max value at gesture end, otherwise the zoom will not exceed the limits.

Type	Required	Platform
bool	No	iOS

canCancelContentTouches

When false, once tracking starts, won't try to drag if the touch moves. The default value is true.

Type	Required	Platform
bool	No	iOS

centerContent

When true, the scroll view automatically centers the content when the content is smaller than the scroll view bounds; when the content is larger than the scroll view, this property has no effect. The default value is false.

Type	Required	Platform
bool	No	iOS

contentInset

The amount by which the scroll view content is inset from the edges of the scroll view. Defaults to `{top: 0, left: 0, bottom: 0, right: 0}`.

Type	Required	Platform
<code>object: {top: number, left: number, bottom: number, right: number}</code>	No	iOS

contentInsetAdjustmentBehavior

This property specifies how the safe area insets are used to modify the content area of the scroll view. The default value of this property is "never". Available on iOS 11 and later.

Type	Required	Platform
<code>enum('automatic', 'scrollableAxes', 'never', 'always')</code>	No	iOS

contentOffset

Used to manually set the starting scroll offset. The default value is `{x: 0, y: 0}`.

Type	Required	Platform
<code>PointPropType</code>	No	iOS

decelerationRate

A floating-point number that determines how quickly the scroll view decelerates after the user lifts their finger. You may also use string shortcuts "normal" and "fast" which match the underlying iOS settings for `UIScrollViewDecelerationRateNormal` and `UIScrollViewDecelerationRateFast` respectively.

- 'normal' : 0.998 on iOS, 0.985 on Android (the default)
- 'fast' : 0.99 on iOS, 0.9 on Android

Type	Required
<code>enum('fast', 'normal'), ,number</code>	No

directionalLockEnabled

When true, the ScrollView will try to lock to only vertical or horizontal scrolling while dragging. The default value is false.

Type	Required	Platform
<code>bool</code>	No	iOS

indicatorStyle

The style of the scroll indicators.

- 'default' (the default), same as `black`.
- 'black' , scroll indicator is black. This style is good against a light background.
- 'white' , scroll indicator is white. This style is good against a dark background.

Type	Required	Platform
<code>enum('default', 'black', 'white')</code>	No	iOS

maximumZoomScale

The maximum allowed zoom scale. The default value is 1.0.

Type	Required	Platform
<code>number</code>	No	iOS

minimumZoomScale

The minimum allowed zoom scale. The default value is 1.0.

Type	Required	Platform
<code>number</code>	No	iOS

pinchGestureEnabled

When true, ScrollView allows use of pinch gestures to zoom in and out. The default value is true.

Type	Required	Platform
<code>bool</code>	No	iOS

scrollEventThrottle

This controls how often the scroll event will be fired while scrolling (as a time interval in ms). A lower number yields better accuracy for code that is tracking the scroll position, but can lead to scroll performance problems due to the volume of information being send over the bridge. You will not notice a difference between values set between 1-16 as the JS run loop is synced to the screen refresh rate. If you do not need precise scroll position tracking, set this value higher to limit the information being sent across the bridge. The default value is zero, which results in the scroll event being sent only once each time the view is scrolled.

Type	Required	Platform
<code>number</code>	No	iOS

scrollIndicatorInsets

The amount by which the scroll view indicators are inset from the edges of the scroll view. This should normally be set to the same value as the `contentInset`. Defaults to `{0, 0, 0, 0}`.

Type	Required	Platform
object: {top: number, left: number, bottom: number, right: number}	No	iOS

scrollsToTop

When true, the scroll view scrolls to top when the status bar is tapped. The default value is true.

Type	Required	Platform
bool	No	iOS

snapToAlignment

When `snapToInterval` is set, `snapToAlignment` will define the relationship of the snapping to the scroll view.

- `'start'` (the default) will align the snap at the left (horizontal) or top (vertical)
- `'center'` will align the snap in the center
- `'end'` will align the snap at the right (horizontal) or bottom (vertical)

Type	Required
enum('start', 'center', 'end')	No

snapToInterval

When set, causes the scroll view to stop at multiples of the value of `snapToInterval`. This can be used for paginating through children that have lengths smaller than the scroll view. Typically used in combination with `snapToAlignment` and `decelerationRate="fast"`. Overrides less configurable `pagingEnabled` prop.

Type	Required
number	No

snapToOffsets

When set, causes the scroll view to stop at the defined offsets. This can be used for paginating through variously sized children that have lengths smaller than the scroll view. Typically used in combination with `decelerationRate="fast"`. Overrides less configurable `pagingEnabled` and `snapToInterval` props.

Type	Required

array of number	No
-----------------	----

snapToStart

Use in conjunction with `snapToOffsets`. By default, the beginning of the list counts as a snap offset. Set `snapToStart` to false to disable this behavior and allow the list to scroll freely between its start and the first `snapToOffsets` offset. The default value is true.

Type	Required
boolean	No

snapToEnd

Use in conjunction with `snapToOffsets`. By default, the end of the list counts as a snap offset. Set `snapToEnd` to false to disable this behavior and allow the list to scroll freely between its end and the last `snapToOffsets` offset. The default value is true.

Type	Required
boolean	No

zoomScale

The current scale of the scroll view content. The default value is 1.0.

Type	Required	Platform
number	No	iOS

nestedScrollEnabled

Enables nested scrolling for Android API level 21+. Nested scrolling is supported by default on iOS.

Type	Required	Platform
bool	No	Android

Methods

scrollTo()

```
scrollTo(([y]: number), object, ([x]: number), ([animated]: boolean));
```

Scrolls to a given x, y offset, either immediately or with a smooth animation.

Example:

```
scrollTo({x: 0, y: 0, animated: true})
```

Note: The weird function signature is due to the fact that, for historical reasons, the function also accepts separate arguments as an alternative to the options object. This is deprecated due to ambiguity (y before x), and SHOULD NOT BE USED.

scrollToEnd()

```
scrollToEnd(([options]: object));
```

If this is a vertical ScrollView scrolls to the bottom. If this is a horizontal ScrollView scrolls to the right.

Use `scrollToEnd({animated: true})` for smooth animated scrolling, `scrollToEnd({animated: false})` for immediate scrolling. If no options are passed, `animated` defaults to true.

scrollWithoutAnimationTo()

```
scrollWithoutAnimationTo(y, x);
```

Deprecated, use `scrollTo` instead.

flashScrollIndicators()

```
flashScrollIndicators();
```

Displays the scroll indicators momentarily.

SectionList

A performant interface for rendering sectioned lists, supporting the most handy features:

- Fully cross-platform.
- Configurable viewability callbacks.
- List header support.
- List footer support.
- Item separator support.
- Section header support.
- Section separator support.
- Heterogeneous data and item rendering support.
- Pull to Refresh.
- Scroll loading.

If you don't need section support and want a simpler interface, use `<FlatList>`.

Simple Examples:

```
// Example 1 (Homogeneous Rendering)
<SectionList
  renderItem={({item, index, section}) => <Text key={index}>{item}</Text>}
  renderSectionHeader={({section: {title}}) => (
    <Text style={{fontWeight: 'bold'}}>{title}</Text>
  )}
  sections={[
    {title: 'Title1', data: ['item1', 'item2']},
    {title: 'Title2', data: ['item3', 'item4']},
    {title: 'Title3', data: ['item5', 'item6']},
  ]}
  keyExtractor={({item, index}) => item + index}
/>
```

```
// Example 2 (Heterogeneous Rendering / No Section Headers)
const overrideRenderItem = ({ item, index, section: { title, data } }) => <Text key={index}>Override{item}</Text>

<SectionList
  renderItem={({ item, index, section }) => <Text key={index}>{item}</Text>}
  sections={[
    { title: 'Title1', data: ['item1', 'item2'], renderItem: overrideRenderItem },
    { title: 'Title2', data: ['item3', 'item4'] },
    { title: 'Title3', data: ['item5', 'item6'] },
  ]}
/>
```

This is a convenience wrapper around `<VirtualizedList>`, and thus inherits its props (as well as those of `<ScrollView>` that aren't explicitly listed here, along with the following caveats:

- Internal state is not preserved when content scrolls out of the render window. Make sure all your data is captured in the item data or external stores like Flux, Redux, or Relay.
- This is a `PureComponent` which means that it will not re-render if `props` remain shallow- equal. Make sure that everything your `renderItem` function depends on is passed as a prop (e.g. `extraData`) that is not `==` after

updates, otherwise your UI may not update on changes. This includes the `data` prop and parent component state.

- In order to constrain memory and enable smooth scrolling, content is rendered asynchronously offscreen. This means it's possible to scroll faster than the fill rate and momentarily see blank content. This is a tradeoff that can be adjusted to suit the needs of each application, and we are working on improving it behind the scenes.
- By default, the list looks for a `key` prop on each item and uses that for the React key. Alternatively, you can provide a custom `keyExtractor` prop.

Props

- `ScrollView` props...

Required props:

- `sections`

Optional props:

- `initialNumToRender`
- `keyExtractor`
- `renderItem`
- `onEndReached`
- `extraData`
- `ItemSeparatorComponent`
- `inverted`
- `ListFooterComponent`
- `legacyImplementation`
- `ListEmptyComponent`
- `onEndReachedThreshold`
- `onRefresh`
- `onViewableItemsChanged`
- `refreshing`
- `removeClippedSubviews`
- `ListHeaderComponent`
- `renderSectionFooter`
- `renderSectionHeader`
- `SectionSeparatorComponent`
- `stickySectionHeadersEnabled`

Methods

- `scrollToLocation`
- `recordInteraction`
- `flashScrollIndicators`

Type Definitions

- `Section`

Reference

Props

sections

The actual data to render, akin to the `data` prop in `FlatList`.

Type	Required
array of Sections	Yes

initialNumToRender

How many items to render in the initial batch. This should be enough to fill the screen but not much more. Note these items will never be unmounted as part of the windowed rendering in order to improve perceived performance of scroll-to-top actions.

Type	Required
number	Yes

keyExtractor

Used to extract a unique key for a given item at the specified index. Key is used for caching and as the react key to track item re-ordering. The default extractor checks `item.key`, then falls back to using the index, like react does. Note that this sets keys for each item, but each overall section still needs its own key.

Type	Required
(item: Item, index: number) => string	Yes

renderItem

Default renderer for every item in every section. Can be over-ridden on a per-section basis. Should return a React element.

Type	Required
function	Yes

The render function will be passed an object with the following keys:

- 'item' (object) - the item object as specified in this section's `data` key
- 'index' (number) - Item's index within the section.
- 'section' (object) - The full section object as specified in `sections`.
- 'separators' (object) - An object with the following keys:
 - 'highlight' (function) - `() => void`

- 'unhighlight' (function) - `() => void`
 - 'updateProps' (function) - `(select, newProps) => void`
 - 'select' (enum) - possible values are 'leading', 'trailing'
 - 'newProps' (object)
-

onEndReached

Called once when the scroll position gets within `onEndReachedThreshold` of the rendered content.

Type	Required
<code>[(info: {distanceFromEnd: number}) => void]</code>	No

extraData

A marker property for telling the list to re-render (since it implements `PureComponent`). If any of your `renderItem`, Header, Footer, etc. functions depend on anything outside of the `data` prop, stick it here and treat it immutably.

Type	Required
any	No

ItemSeparatorComponent

Rendered in between each item, but not at the top or bottom. By default, `highlighted`, `section`, and `[leading/trailing][Item/Separator]` props are provided. `renderItem` provides `separators.highlight / unhighlight` which will update the `highlighted` prop, but you can also add custom props with `separators.updateProps`.

Type	Required
<code>[component, function, element]</code>	No

inverted

Reverses the direction of scroll. Uses scale transforms of -1.

Type	Required
<code>[boolean]</code>	No

ListFooterComponent

Rendered at the very end of the list. Can be a React Component Class, a render function, or a rendered element.

Type	Required
<code>[component, function, element]</code>	No

legacyImplementation

Type	Required
[boolean]	No

ListEmptyComponent

Rendered when the list is empty. Can be a React Component Class, a render function, or a rendered element.

Type	Required
[component, function, element]	No

onEndReachedThreshold

How far from the end (in units of visible length of the list) the bottom edge of the list must be from the end of the content to trigger the `onEndReached` callback. Thus a value of 0.5 will trigger `onEndReached` when the end of the content is within half the visible length of the list.

Type	Required
[number]	No

onRefresh

If provided, a standard RefreshControl will be added for "Pull to Refresh" functionality. Make sure to also set the `refreshing` prop correctly.

Type	Required
[() => void]	No

onViewableItemsChanged

Called when the viewability of rows changes, as defined by the `viewabilityConfig` prop.

Type	Required
function	No

The function will be passed an object with the following keys:

- 'viewableItems' (array of `ViewToken`s)
- 'changed' (array of `ViewToken`s)

The `ViewToken` type is exported by `ViewabilityHelper.js`:

Name	Type	Required
item	any	Yes
key	string	Yes
index	number	No
isViewable	boolean	Yes
section	any	No

refreshing

Set this true while waiting for new data from a refresh.

Type	Required
[boolean]	No

removeClippedSubviews

Note: may have bugs (missing content) in some circumstances - use at your own risk.

This may improve scroll performance for large lists.

Type	Required
boolean	No

ListHeaderComponent

Rendered at the very beginning of the list. Can be a React Component Class, a render function, or a rendered element.

Type	Required
component, function, element	No

renderSectionFooter

Rendered at the bottom of each section.

Type	Required
<code>[(info: {section: SectionT}) => ?React.Element]</code>	No

renderSectionHeader

Rendered at the top of each section. These stick to the top of the `ScrollView` by default on iOS. See `stickySectionHeadersEnabled`.

Type	Required
<code>[(info: {section: SectionT}) => ?React.Element]</code>	No

SectionSeparatorComponent

Rendered at the top and bottom of each section (note this is different from `ItemSeparatorComponent` which is only rendered between items). These are intended to separate sections from the headers above and below and typically have the same highlight response as `ItemSeparatorComponent`. Also receives `highlighted`, `[leading/trailing][Item/Separator]`, and any custom props from `separators.updateProps`.

Type	Required
<code>[ReactClass]</code>	No

stickySectionHeadersEnabled

Makes section headers stick to the top of the screen until the next one pushes it off. Only enabled by default on iOS because that is the platform standard there.

Type	Required
<code>boolean</code>	No

Methods

scrollToLocation()

```
scrollToLocation(params);
```

Scrolls to the item at the specified `sectionIndex` and `itemIndex` (within the section) positioned in the viewable area such that `viewPosition` 0 places it at the top (and may be covered by a sticky header), 1 at the bottom, and 0.5 centered in the middle.

Note: Cannot scroll to locations outside the render window without specifying the `getItemLayout` or `onScrollToIndexFailed` prop.

Parameters:

Name	Type	Required	Description
<code>params</code>	<code>object</code>	Yes	See below.

Valid `params` keys are:

- 'animated' (boolean) - Whether the list should do an animation while scrolling. Defaults to `true`.
- 'itemIndex' (number) - Index within section for the item to scroll to. Required.
- 'sectionIndex' (number) - Index for section that contains the item to scroll to. Required.
- 'viewOffset' (number) - A fixed number of pixels to offset the final target position, e.g. to compensate for sticky headers.
- 'viewPosition' (number) - A value of `0` places the item specified by index at the top, `1` at the bottom, and `0.5` centered in the middle.

recordInteraction()

```
recordInteraction();
```

Tells the list an interaction has occurred, which should trigger viewability calculations, e.g. if `waitForInteractions` is true and the user has not scrolled. This is typically called by taps on items or by navigation actions.

flashScrollIndicators()

```
flashScrollIndicators();
```

Displays the scroll indicators momentarily.

Type Definitions

Section

An object that identifies the data to be rendered for a given section.

Type
any

Properties:

Name	Type	Description
data	array	The data for rendering items in this section. Array of objects, much like FlatList's data prop .
[key]	string	Optional key to keep track of section re-ordering. If you don't plan on re-ordering sections, the array index will be used by default.
[renderItem]	function	Optionally define an arbitrary item renderer for this section, overriding the default <code>renderItem</code> for the list.
[ItemSeparatorComponent]	component, function,	Optionally define an arbitrary item separator for this section, overriding the default <code>ItemSeparatorComponent</code> for

	element	the list.
[keyExtractor]	function	Optionally define an arbitrary key extractor for this section, overriding the default <code>keyExtractor</code> .

SegmentedControlIOS

Use `SegmentedControlIOS` to render a UISegmentedControl iOS.

Programmatically changing selected index

The selected index can be changed on the fly by assigning the `selectedIndex` prop to a state variable, then changing that variable. Note that the state variable would need to be updated as the user selects a value and changes the index, as shown in the example below.

Example

```
<SegmentedControlIOS
  values={['One', 'Two']}
  selectedIndex={this.state.selectedIndex}
  onChange={(event) => {
    this.setState({selectedIndex: event.nativeEvent.selectedSegmentIndex});
  }}
/>
```



Props

- [View props...](#)
- `enabled`
- `momentary`
- `onChange`
- `onValueChange`
- `selectedIndex`
- `tintColor`
- `values`

Reference

Props

enabled

If false the user won't be able to interact with the control. Default value is true.

Type	Required
bool	No



momentary

If true, then selecting a segment won't persist visually. The `onValueChange` callback will still work as expected.

Type	Required
bool	No



onChange

Callback that is called when the user taps a segment; passes the event as an argument

Type	Required
function	No

onValueChange

Callback that is called when the user taps a segment; passes the segment's value as an argument

Type	Required
function	No

selectedIndex

The index in `props.values` of the segment to be (pre)selected.

Type	Required
number	No

tintColor

Accent color of the control.

Type	Required
string	No



values

The labels for the control's segment buttons, in order.

Type	Required
array of string	No

Slider

A component used to select a single value from a range of values.

Props

- [View props...](#)
 - `style`
 - `disabled`
 - `maximumValue`
 - `minimumTrackTintColor`
 - `minimumValue`
 - `onSlidingComplete`
 - `onValueChange`
 - `step`
 - `maximumTrackTintColor`
 - `testID`
 - `value`
 - `thumbTintColor`
 - `maximumTrackImage`
 - `minimumTrackImage`
 - `thumbImage`
 - `trackImage`
-

Reference

Props

`style`

Used to style and layout the `Slider`. See `StyleSheet.js` and `viewStylePropTypes.js` for more info.

Type	Required
<code>View.style</code>	No

`disabled`

If true the user won't be able to move the slider. Default value is false.

Type	Required
<code>bool</code>	No

maximumValue

Initial maximum value of the slider. Default value is 1.

Type	Required
number	No

minimumTrackTintColor

The color used for the track to the left of the button. Overrides the default blue gradient image on iOS.

Type	Required
color	No

minimumValue

Initial minimum value of the slider. Default value is 0.

Type	Required
number	No

onSlidingComplete

Callback that is called when the user releases the slider, regardless if the value has changed. The current value is passed as an argument to the callback handler.

Type	Required
function	No

onValueChange

Callback continuously called while the user is dragging the slider.

Type	Required
function	No

step

Step value of the slider. The value should be between 0 and (maximumValue - minimumValue). Default value is 0.

Type	Required
number	No

maximumTrackTintColor

The color used for the track to the right of the button. Overrides the default gray gradient image on iOS.

Type	Required
color	No

testID

Used to locate this view in UI automation tests.

Type	Required
string	No

value

Initial value of the slider. The value should be between minimumValue and maximumValue, which default to 0 and 1 respectively. Default value is 0.

This is not a controlled component, you don't need to update the value during dragging.

Type	Required
number	No

thumbTintColor

Color of the foreground switch grip.

Type	Required	Platform
color	No	Android

maximumTrackImage

Assigns a maximum track image. Only static images are supported. The leftmost pixel of the image will be stretched to fill the track.

Type	Required	Platform
Image.propTypes.source	No	iOS

minimumTrackImage

Assigns a minimum track image. Only static images are supported. The rightmost pixel of the image will be stretched to fill the track.

Type	Required	Platform
Image.propTypes.source	No	iOS

thumbImage

Sets an image for the thumb. Only static images are supported.

Type	Required	Platform
Image.propTypes.source	No	iOS

trackImage

Assigns a single image for the track. Only static images are supported. The center pixel of the image will be stretched to fill the track.

Type	Required	Platform
Image.propTypes.source	No	iOS

SnapshotViewIOS

Props

- [View props...](#)
 - `onSnapshotReady`
 - `testIdentifier`
-

Reference

Props

`onSnapshotReady`

Type	Required
function	No

`testIdentifier`

Type	Required
string	No

StatusBar

Component to control the app status bar.

Usage with Navigator

It is possible to have multiple `StatusBar` components mounted at the same time. The props will be merged in the order the `StatusBar` components were mounted.

```
<View>
  <StatusBar
    backgroundColor="blue"
    barStyle="light-content"
  />
  <View>
    <StatusBar hidden={route.statusBarHidden} />
    ...
  </View>
</View>
```

Imperative API

For cases where using a component is not ideal, there is also an imperative API exposed as static functions on the component. It is however not recommended to use the static API and the component for the same prop because any value set by the static API will get overriden by the one set by the component in the next render.

Constants

`currentHeight` (Android only) The height of the status bar.

Props

- `animated`
- `barStyle`
- `hidden`
- `backgroundColor`
- `translucent`
- `networkActivityIndicatorVisible`
- `showHideTransition`

Methods

- `setHidden`
- `setBarStyle`
- `setNetworkActivityIndicatorVisible`
- `setBackgroundColor`
- `setTranslucent`

Type Definitions

- [StatusBarStyle](#)
 - [StatusBarAnimation](#)
-

Reference

Props

animated

If the transition between status bar property changes should be animated. Supported for backgroundColor, barStyle and hidden.

Type	Required
bool	No

barStyle

Sets the color of the status bar text.

Type	Required
enum('default', 'light-content', 'dark-content')	No

hidden

If the status bar is hidden.

Type	Required
bool	No

backgroundColor

The background color of the status bar.

Type	Required	Platform
color	No	Android

translucent

If the status bar is translucent. When translucent is set to true, the app will draw under the status bar. This is useful when using a semi transparent status bar color.

Type	Required	Platform
bool	No	Android

networkActivityIndicatorVisible

If the network activity indicator should be visible.

Type	Required	Platform
bool	No	iOS

showHideTransition

The transition effect when showing and hiding the status bar using the `hidden` prop. Defaults to 'fade'.

Type	Required	Platform
enum('fade', 'slide')	No	iOS

Methods

setHidden()

```
static setHidden(hidden: boolean, [animation]: StatusBarAnimation)
```

Show or hide the status bar

Parameters:

Name	Type	Required	Description
hidden	boolean	Yes	Hide the status bar.
animation	StatusBarAnimation	No	Optional animation when changing the status bar hidden property.

setBarStyle()

```
static setBarStyle(style: StatusBarStyle, [animated]: boolean)
```

Set the status bar style

Parameters:

Name	Type	Required	Description
style	StatusBarStyle	Yes	Status bar style to set
animated	boolean	No	Animate the style change.

setNetworkActivityIndicatorVisible()

```
static setNetworkActivityIndicatorVisible(visible: boolean)
```

Control the visibility of the network activity indicator

Parameters:

Name	Type	Required	Description
visible	boolean	Yes	Show the indicator.

setBackgroundColor()

```
static setBackgroundColor(color: string, [animated]: boolean)
```

Set the background color for the status bar

Parameters:

Name	Type	Required	Description
color	string	Yes	Background color.
animated	boolean	No	Animate the style change.

setTranslucent()

```
static setTranslucent(translucent: boolean)
```

Control the translucency of the status bar

Parameters:

Name	Type	Required	Description
translucent	boolean	Yes	Set as translucent.

Type Definitions

StatusBarStyle

Status bar style

Type
\$Enum

Constants:

Value	Description
default	Default status bar style (dark for iOS, light for Android)
light-content	Dark background, white texts and icons
dark-content	Light background, dark texts and icons (requires API>=23 on Android)

StatusBarAnimation

Status bar animation

Type
\$Enum

Constants:

Value	Description
none	No animation
fade	Fade animation
slide	Slide animation

Switch

Renders a boolean input.

This is a controlled component that requires an `onValueChange` callback that updates the `value` prop in order for the component to reflect user actions. If the `value` prop is not updated, the component will continue to render the supplied `value` prop instead of the expected result of any user actions.

Props

- [View props...](#)
 - `disabled`
 - `trackColor`
 - `ios_backgroundColor`
 - `onValueChange`
 - `testID`
 - `thumbColor`
 - `tintColor`
 - `value`
-

Reference

Props

`disabled`

If true the user won't be able to toggle the switch. Default value is false.

Type	Required
bool	No

`trackColor`

Custom colors for the switch track.

iOS: When the switch value is false, the track shrinks into the border. If you want to change the color of the background exposed by the shrunken track, use `ios_backgroundColor`.

Type	Required
<code>object: {false: color, true: color}</code>	No

ios_backgroundColor

On iOS, custom color for the background. This background color can be seen either when the switch value is false or when the switch is disabled (and the switch is translucent).

Type	Required
color	No

onValueChange

Invoked with the new value when the value changes.

Type	Required
function	No

testID

Used to locate this view in end-to-end tests.

Type	Required
string	No

thumbColor

Color of the foreground switch grip. If this is set on iOS, the switch grip will lose its drop shadow.

Type	Required
color	No

tintColor

`tintColor` is deprecated, use `trackColor` instead.

Border color on iOS and background color on Android when the switch is turned off.

Type	Required
color	No

value

The value of the switch. If true the switch will be turned on. Default value is false.

Type	Required
bool	No

TabBarIOS.Item

Props

- [View props...](#)
 - `selected`
 - `badge`
 - `icon`
 - `onPress`
 - `renderAsOriginal`
 - `badgeColor`
 - `selectedIcon`
 - `style`
 - `systemIcon`
 - `title`
 - `isTVSelectable`
-

Reference

Props

`selected`

It specifies whether the children are visible or not. If you see a blank content, you probably forgot to add a selected one.

Type	Required
bool	No

`badge`

Little red bubble that sits at the top right of the icon.

Type	Required
string, ,number	No

`icon`

A custom icon for the tab. It is ignored when a system icon is defined.

Type	Required
Image.propTypes.source	No

onPress

Callback when this tab is being selected, you should change the state of your component to set selected={true}.

Type	Required
function	No

renderAsOriginal

If set to true it renders the image as original, it defaults to being displayed as a template

Type	Required
bool	No

badgeColor

Background color for the badge. Available since iOS 10.

Type	Required
color	No

selectedIcon

A custom icon when the tab is selected. It is ignored when a system icon is defined. If left empty, the icon will be tinted in blue.

Type	Required
Image.propTypes.source	No

style

React style object.

Type	Required
View.style	No

systemIcon

Items comes with a few predefined system icons. Note that if you are using them, the title and selectedIcon will be overridden with the system ones.

Type	Required
enum('bookmarks', 'contacts', 'downloads', 'favorites', 'featured', 'history', 'more', 'most-recent', 'most-viewed', 'recents', 'search', 'top-rated')	No

title

Text that appears under the icon. It is ignored when a system icon is defined.

Type	Required
string	No

isTVSelectable

(Apple TV only)* When set to true, this view will be focusable and navigable using the Apple TV remote.

Type	Required	Platform
bool	No	iOS

TabBarIOS

Props

- [View props...](#)
 - `barStyle`
 - `barTintColor`
 - `itemPositioning`
 - `style`
 - `tintColor`
 - `translucent`
 - `unselectedItemTintColor`
 - `unselectedTintColor`
-

Reference

Props

barStyle

The style of the tab bar. Supported values are 'default', 'black'. Use 'black' instead of setting `barTintColor` to black. This produces a tab bar with the native iOS style with higher translucency.

Type	Required
<code>enum('default', 'black')</code>	No

barTintColor

Background color of the tab bar

Type	Required
<code>color</code>	No

itemPositioning

Specifies tab bar item positioning. Available values are:

- fill - distributes items across the entire width of the tab bar
- center - centers item in the available tab bar space
- auto (default) - distributes items dynamically according to the user interface idiom. In a horizontally compact

environment (e.g. iPhone 5) this value defaults to `fill`, in a horizontally regular one (e.g. iPad) it defaults to center.

Type	Required
<code>enum('fill', 'center', 'auto')</code>	No

style

Type	Required
<code>View.style</code>	No

tintColor

Color of the currently selected tab icon

Type	Required
<code>color</code>	No

translucent

A Boolean value that indicates whether the tab bar is translucent

Type	Required
<code>bool</code>	No

unselectedItemTintColor

Color of unselected tab icons. Available since iOS 10.

Type	Required
<code>color</code>	No

unselectedTintColor

Color of text on unselected tabs

Type	Required
<code>color</code>	No

Text

A React component for displaying text.

`Text` supports nesting, styling, and touch handling.

In the following example, the nested title and body text will inherit the `fontFamily` from `styles.baseText`, but the title provides its own additional styles. The title and body will stack on top of each other on account of the literal newlines:

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

export default class TextInANest extends Component {
  constructor(props) {
    super(props);
    this.state = {
      titleText: "Bird's Nest",
      bodyText: 'This is not really a bird nest.'
    };
  }

  render() {
    return (
      <Text style={styles.baseText}>
        <Text style={styles.titleText} onPress={this.onPressTitle}>
          {this.state.titleText}{'\n'}{'\n'}
        </Text>
        <Text numberOfLines={5}>
          {this.state.bodyText}
        </Text>
      </Text>
    );
  }
}

const styles = StyleSheet.create({
  baseText: {
    fontFamily: 'Cochin',
  },
  titleText: {
    fontSize: 20,
    fontWeight: 'bold',
  },
});

// skip this line if using Create React Native App
AppRegistry.registerComponent('TextInANest', () => TextInANest);
```

Nested text

Both iOS and Android allow you to display formatted text by annotating ranges of a string with specific formatting like bold or colored text (`NSAttributedString` on iOS, `SpannableString` on Android). In practice, this is very tedious. For React Native, we decided to use web paradigm for this where you can nest text to achieve the same effect.

```

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

export default class BoldAndBeautiful extends Component {
  render() {
    return (
      <Text style={{fontWeight: 'bold'}}>
        I am bold
        <Text style={{color: 'red'}}>
          and red
        </Text>
      </Text>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => BoldAndBeautiful);

```

Behind the scenes, React Native converts this to a flat `NSAttributedString` or `SpannableString` that contains the following information:

```

"I am bold and red"
0-9: bold
9-17: bold, red

```

Nested views (iOS only)

On iOS, you can nest views within your Text component. Here's an example:

```

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

export default class BlueIsCool extends Component {
  render() {
    return (
      <Text>
        There is a blue square
        <View style={{width: 50, height: 50, backgroundColor: 'steelblue'}} />
        in between my text.
      </Text>
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => BlueIsCool);

```

In order to use this feature, you must give the view a `width` and a `height`.

Containers

The `<Text>` element is special relative to layout: everything inside is no longer using the flexbox layout but using text layout. This means that elements inside of a `<Text>` are no longer rectangles, but wrap when they see the end of the line.

```
<Text>
  <Text>First part and </Text>
  <Text>second part</Text>
</Text>
// Text container: the text will be inline if the space allowed it
// |First part and second part|


// otherwise, the text will flow as if it was one
// |First part |
// |and second |
// |part       |


<View>
  <Text>First part and </Text>
  <Text>second part</Text>
</View>
// View container: each text is its own block
// |First part and|
// |second part    |


// the text will flow in its own block
// |First part |
// |and         |
// |second part|
```

Limited Style Inheritance

On the web, the usual way to set a font family and size for the entire document is to take advantage of inherited CSS properties like so:

```
html {
  font-family: 'lucida grande', tahoma, verdana, arial, sans-serif;
  font-size: 11px;
  color: #141823;
}
```

All elements in the document will inherit this font unless they or one of their parents specifies a new rule.

In React Native, we are more strict about it: **you must wrap all the text nodes inside of a `<Text>` component**. You cannot have a text node directly under a `<View>`.

```
// BAD: will raise exception, can't have a text node as child of a <View>
<View>
  Some text
</View>

// GOOD
<View>
  <Text>
    Some text
  </Text>
</View>
```

You also lose the ability to set up a default font for an entire subtree. The recommended way to use consistent fonts and sizes across your application is to create a component `MyAppText` that includes them and use this component across your app. You can also use this component to make more specific components like `MyAppHeaderText` for other kinds of text.

```
<View>
  <MyAppText>
    Text styled with the default font for the entire application
  </MyAppText>
  <MyAppHeaderText>Text styled as a header</MyAppHeaderText>
</View>
```

Assuming that `MyAppText` is a component that simply renders out its children into a `Text` component with styling, then `MyAppHeaderText` can be defined as follows:

```
class MyAppHeaderText extends Component {
  render() {
    return (
      <MyAppText>
        <Text style={{fontSize: 20}}>{this.props.children}</Text>
      </MyAppText>
    );
  }
}
```

Composing `MyAppText` in this way ensures that we get the styles from a top-level component, but leaves us the ability to add / override them in specific use cases.

React Native still has the concept of style inheritance, but limited to text subtrees. In this case, the second part will be both bold and red.

```
<Text style={{fontWeight: 'bold'}}>
  I am bold
  <Text style={{color: 'red'}}>and red</Text>
</Text>
```

We believe that this more constrained way to style text will yield better apps:

- (Developer) React components are designed with strong isolation in mind: You should be able to drop a component anywhere in your application, trusting that as long as the props are the same, it will look and behave the same way. Text properties that could inherit from outside of the props would break this isolation.
- (Implementor) The implementation of React Native is also simplified. We do not need to have a `fontFamily` field on every single element, and we do not need to potentially traverse the tree up to the root every time we display a text node. The style inheritance is only encoded inside of the native Text component and doesn't leak to other components or the system itself.

Props

- `selectable`
- `accessibilityHint`

- `accessibilityLabel`
 - `accessible`
 - `ellipsizeMode`
 - `nativeID`
 - `numberOfLines`
 - `onLayout`
 - `onLongPress`
 - `onPress`
 - `pressRetentionOffset`
 - `allowFontScaling`
 - `style`
 - `testID`
 - `disabled`
 - `selectionColor`
 - `textBreakStrategy`
 - `adjustsFontSizeToFit`
 - `minimumFontSize`
 - `suppressHighlighting`
-

Reference

Props

selectable

Lets the user select text, to use the native copy and paste functionality.

Type	Required
bool	No

accessibilityHint

An accessibility hint helps users understand what will happen when they perform an action on the accessibility element when that result is not obvious from the accessibility label.

Type	Required
string	No

accessibilityLabel

Overrides the text that's read by the screen reader when the user interacts with the element. By default, the label is constructed by traversing all the children and accumulating all the `Text` nodes separated by space.

Type	Required
node	No

accessible

When set to `true`, indicates that the view is an accessibility element. The default value for a `Text` element is `true`.

See the [Accessibility guide](#) for more information.

Type	Required
bool	No

ellipsizeMode

When `numberOfLines` is set, this prop defines how text will be truncated. `numberOfLines` must be set in conjunction with this prop.

This can be one of the following values:

- `head` - The line is displayed so that the end fits in the container and the missing text at the beginning of the line is indicated by an ellipsis glyph. e.g., "...wxyz"
- `middle` - The line is displayed so that the beginning and end fit in the container and the missing text in the middle is indicated by an ellipsis glyph. "ab...yz"
- `tail` - The line is displayed so that the beginning fits in the container and the missing text at the end of the line is indicated by an ellipsis glyph. e.g., "abcd..."
- `clip` - Lines are not drawn past the edge of the text container.

The default is `tail`.

Type	Required
enum('head', 'middle', 'tail', 'clip')	No

nativeID

Used to locate this view from native code.

Type	Required
string	No

numberOfLines

Used to truncate the text with an ellipsis after computing the text layout, including line wrapping, such that the total number of lines does not exceed this number.

This prop is commonly used with `ellipsizeMode`.

Type	Required
number	No

onLayout

Invoked on mount and layout changes with

```
{nativeEvent: {layout: {x, y, width, height}}}
```

Type	Required
function	No

onLongPress

This function is called on long press.

e.g., `onLongPress={this.increaseSize}>`

Type	Required
function	No

onPress

This function is called on press.

e.g., `onPress={() => console.log('1st')}`

Type	Required
function	No

pressRetentionOffset

When the scroll view is disabled, this defines how far your touch may move off of the button, before deactivating the button. Once deactivated, try moving it back and you'll see that the button is once again reactivated! Move it back and forth several times while the scroll view is disabled. Ensure you pass in a constant to reduce memory allocations.

Type	Required
object: {top: number, left: number, bottom: number, right: number}	No

allowFontScaling

Specifies whether fonts should scale to respect Text Size accessibility settings. The default is `true`.

Type	Required
<code>bool</code>	No

style

Type	Required
<code>style</code>	No

- [View Style Props...](#)
- `textShadowOffset` : Object: {width: number,height: number}
- `color` : [color](#)
- `fontSize` : number
- `fontStyle` : enum('normal', 'italic')
- `fontWeight` : enum('normal', 'bold', '100', '200', '300', '400', '500', '600', '700', '800', '900')

Specifies font weight. The values 'normal' and 'bold' are supported for most fonts. Not all fonts have a variant for each of the numeric values, in that case the closest one is chosen.

- `lineHeight` : number
- `textAlign` : enum('auto', 'left', 'right', 'center', 'justify')

Specifies text alignment. The value 'justify' is only supported on iOS and fallbacks to `left` on Android.

- `textDecorationLine` : enum('none', 'underline', 'line-through', 'underline line-through')
- `textShadowColor` : [color](#)
- `fontFamily` : string
- `textShadowRadius` : number
- `includeFontPadding` : bool (*Android*)

Set to `false` to remove extra font padding intended to make space for certain ascenders / descenders. With some fonts, this padding can make text look slightly misaligned when centered vertically. For best results also set `textAlignVertical` to `center`. Default is true.

- `textAlignVertical` : enum('auto', 'top', 'bottom', 'center') (*Android*)
- `fontVariant` : array of enum('small-caps', 'oldstyle-nums', 'lining-nums', 'tabular-nums', 'proportional-nums') (*iOS*)
- `letterSpacing` : number

Increase or decrease the spacing between characters. The default is 0, for no extra letter spacing.

iOS: The additional space will be rendered after each glyph.

Android: Only supported since Android 5.0 - older versions will ignore this attribute. Please note that additional space will be added *around* the glyphs (half on each side), which differs from the iOS rendering. It is possible to emulate the iOS rendering by using layout attributes, e.g. negative margins, as appropriate for your situation.

- `textDecorationColor` : `color` (iOS)
 - `textDecorationStyle` : enum('solid', 'double', 'dotted', 'dashed') (iOS)
 - `textTransform` : enum('none', 'uppercase', 'lowercase', 'capitalize')
 - `writingDirection` : enum('auto', 'ltr', 'rtl') (iOS)
-

testID

Used to locate this view in end-to-end tests.

Type	Required
string	No

disabled

Specifies the disabled state of the text view for testing purposes

Type	Required	Platform
bool	No	Android

selectionColor

The highlight color of the text.

Type	Required	Platform
color	No	Android

textBreakStrategy

Set text break strategy on Android API Level 23+, possible values are `simple` , `highQuality` , `balanced` . The default value is `highQuality` .

Type	Required	Platform
enum('simple', 'highQuality', 'balanced')	No	Android

adjustsFontSizeToFit

Specifies whether font should be scaled down automatically to fit given style constraints.

Type	Required	Platform
bool	No	iOS

minimumFontSize

Specifies smallest possible scale a font can reach when `adjustsFontSizeToFit` is enabled. (values 0.01-1.0).

Type	Required	Platform
number	No	iOS

suppressHighlighting

When `true`, no visual change is made when text is pressed down. By default, a gray oval highlights the text on press down.

Type	Required	Platform
bool	No	iOS

TextInput

A foundational component for inputting text into the app via a keyboard. Props provide configurability for several features, such as auto-correction, auto-capitalization, placeholder text, and different keyboard types, such as a numeric keypad.

The simplest use case is to plop down a `TextInput` and subscribe to the `onChangeText` events to read the user input. There are also other events, such as `onSubmitEditing` and `onFocus` that can be subscribed to. A simple example:

```
import React, { Component } from 'react';
import { AppRegistry, TextInput } from 'react-native';

export default class UselessTextInput extends Component {
  constructor(props) {
    super(props);
    this.state = { text: 'Useless Placeholder' };
  }

  render() {
    return (
      <TextInput
        style={{height: 40, borderColor: 'gray', borderWidth: 1}}
        onChangeText={({text}) => this.setState({text})}
        value={this.state.text}
      />
    );
  }
}

// skip this line if using Create React Native App
AppRegistry.registerComponent('AwesomeProject', () => UselessTextInput);
```

Two methods exposed via the native element are `.focus()` and `.blur()` that will focus or blur the `TextInput` programmatically.

Note that some props are only available with `multiline={true/false}`. Additionally, border styles that apply to only one side of the element (e.g., `borderBottomColor`, `borderLeftWidth`, etc.) will not be applied if `multiline=false`. To achieve the same effect, you can wrap your `TextInput` in a `View`:

```
import React, { Component } from 'react';
import { AppRegistry, View, TextInput } from 'react-native';

class UselessTextInput extends Component {
  render() {
    return (
      <View style={{borderWidth: 1, padding: 5}}>
        <TextInput
          {...this.props} // Inherit any props passed to it; e.g., multiline, numberOfLines below
          editable = {true}
          maxLength = {40}
        />
      </View>
    );
  }
}

export default class UselessTextInputMultiline extends Component {
```

```

constructor(props) {
  super(props);
  this.state = {
    text: 'Useless Multiline Placeholder',
  };
}

// If you type something in the text box that is a color, the background will change to that
// color.
render() {
  return (
    <View style={{{
      backgroundColor: this.state.text,
      borderBottomColor: '#000000',
      borderBottomWidth: 1 }}}
    >
      <UselessTextInput
        multiline = {true}
        numberOfLines = {4}
        onChangeText={(text) => this.setState({text})}
        value={this.state.text}
      />
    </View>
  );
}

// skip these lines if using Create React Native App
AppRegistry.registerComponent(
  'AwesomeProject',
  () => UselessTextInputMultiline
);

```

`TextInput` has by default a border at the bottom of its view. This border has its padding set by the background image provided by the system, and it cannot be changed. Solutions to avoid this is to either not set height explicitly, case in which the system will take care of displaying the border in the correct position, or to not display the border by setting `underlineColorAndroid` to transparent.

Note that on Android performing text selection in input can change app's activity `windowSoftInputMode` param to `adjustResize`. This may cause issues with components that have position: 'absolute' while keyboard is active. To avoid this behavior either specify `windowSoftInputMode` in `AndroidManifest.xml` (<https://developer.android.com/guide/topics/manifest/activity-element.html>) or control this param programmatically with native code.

Props

- [View props...](#)
- `allowFontScaling`
- `autoCapitalize`
- `autoCorrect`
- `autoFocus`
- `blurOnSubmit`
- `caretHidden`
- `clearButtonMode`
- `clearTextOnFocus`
- `contextMenuHidden`

- `dataDetectorTypes`
- `defaultValue`
- `disableFullscreenUI`
- `editable`
- `enablesReturnKeyAutomatically`
- `inlineImageLeft`
- `inlineImagePadding`
- `keyboardAppearance`
- `keyboardType`
- `maxLength`
- `multiline`
- `numberOfLines`
- `onBlur`
- `onChange`
- `onChangeText`
- `onContentSizeChange`
- `onEndEditing`
- `onFocus`
- `onKeyPress`
- `onLayout`
- `onScroll`
- `onSelectionChange`
- `onSubmitEditing`
- `placeholder`
- `placeholderTextColor`
- `returnKeyLabel`
- `returnKeyType`
- `scrollEnabled`
- `secureTextEntry`
- `selection`
- `selectionColor`
- `selectionState`
- `selectTextOnFocus`
- `spellCheck`
- `textContentType`
- `style`
- `textBreakStrategy`
- `underlineColorAndroid`
- `value`

Methods

- `clear`
 - `isFocused`
-

Reference

Props

allowFontScaling

Specifies whether fonts should scale to respect Text Size accessibility settings. The default is `true`.

Type	Required
bool	No

autoCapitalize

Can tell `TextInput` to automatically capitalize certain characters. This property is not supported by some keyboard types such as `name-phone-pad`.

- `characters` : all characters.
- `words` : first letter of each word.
- `sentences` : first letter of each sentence (*default*).
- `none` : don't auto capitalize anything.

Type	Required
<code>enum('none', 'sentences', 'words', 'characters')</code>	No

autoCorrect

If `false`, disables auto-correct. The default value is `true`.

Type	Required
bool	No

autoFocus

If `true`, focuses the input on `componentDidMount`. The default value is `false`.

Type	Required
bool	No

blurOnSubmit

If `true`, the text field will blur when submitted. The default value is true for single-line fields and false for multiline fields. Note that for multiline fields, setting `blurOnSubmit` to `true` means that pressing return will blur the field and trigger the `onSubmitEditing` event instead of inserting a newline into the field.

Type	Required

bool	No
------	----

caretHidden

If `true`, caret is hidden. The default value is `false`.

Type	Required
bool	No

clearButtonMode

When the clear button should appear on the right side of the text view. This property is supported only for single-line TextInput component. The default value is `never`.

Type	Required	Platform
enum('never', 'while-editing', 'unless-editing', 'always')	No	iOS

clearTextOnFocus

If `true`, clears the text field automatically when editing begins.

Type	Required	Platform
bool	No	iOS

contextMenuHidden

If `true`, context menu is hidden. The default value is `false`.

Type	Required
bool	No

dataDetectorTypes

Determines the types of data converted to clickable URLs in the text input. Only valid if `multiline={true}` and `editable={false}`. By default no data types are detected.

You can provide one type or an array of many types.

Possible values for `dataDetectorTypes` are:

- `'phoneNumber'`
- `'link'`
- `'address'`

- 'calendarEvent'
- 'none'
- 'all'

Type	Required	Platform
enum('phoneNumber', 'link', 'address', 'calendarEvent', 'none', 'all'), ,array of enum('phoneNumber', 'link', 'address', 'calendarEvent', 'none', 'all')	No	iOS

defaultValue

Provides an initial value that will change when the user starts typing. Useful for simple use-cases where you do not want to deal with listening to events and updating the value prop to keep the controlled state in sync.

Type	Required
string	No

disableFullscreenUI

When `false`, if there is a small amount of space available around a text input (e.g. landscape orientation on a phone), the OS may choose to have the user edit the text inside of a full screen text input mode. When `true`, this feature is disabled and users will always edit the text directly inside of the text input. Defaults to `false`.

Type	Required	Platform
bool	No	Android

editable

If `false`, text is not editable. The default value is `true`.

Type	Required
bool	No

enablesReturnKeyAutomatically

If `true`, the keyboard disables the return key when there is no text and automatically enables it when there is text. The default value is `false`.

Type	Required	Platform
bool	No	iOS

inlineImageLeft

If defined, the provided image resource will be rendered on the left. The image resource must be inside `/android/app/src/main/res/drawable` and referenced like

```
<TextInput  
    inlineImageLeft='search_icon'  
/>
```

Type	Required	Platform
string	No	Android

inlineImagePadding

Padding between the inline image, if any, and the text input itself.

Type	Required	Platform
number	No	Android

keyboardAppearance

Determines the color of the keyboard.

Type	Required	Platform
enum('default', 'light', 'dark')	No	iOS

keyboardType

Determines which keyboard to open, e.g. `numeric`.

The following values work across platforms:

- `default`
- `number-pad`
- `decimal-pad`
- `numeric`
- `email-address`
- `phone-pad`

iOS Only

The following values work on iOS only:

- `ascii-capable`
- `numbers-and-punctuation`
- `url`
- `name-phone-pad`
- `twitter`

- `web-search`

Android Only

The following values work on Android only:

- `visible-password`

Type	Required
<code>enum('default', 'email-address', 'numeric', 'phone-pad', 'ascii-capable', 'numbers-and-punctuation', 'url', 'number-pad', 'name-phone-pad', 'decimal-pad', 'twitter', 'web-search', 'visible-password')</code>	No

maxLength

Limits the maximum number of characters that can be entered. Use this instead of implementing the logic in JS to avoid flicker.

Type	Required
<code>number</code>	No

multiline

If `true`, the text input can be multiple lines. The default value is `false`.

Type	Required
<code>bool</code>	No

numberOfLines

Sets the number of lines for a `TextInput`. Use it with multiline set to `true` to be able to fill the lines.

Type	Required	Platform
<code>number</code>	No	Android

onBlur

Callback that is called when the text input is blurred.

Type	Required
<code>function</code>	No

onChange

Callback that is called when the text input's text changes.

Type	Required
function	No

onChangeText

Callback that is called when the text input's text changes. Changed text is passed as an argument to the callback handler.

Type	Required
function	No

onContentSizeChange

Callback that is called when the text input's content size changes. This will be called with `{ nativeEvent: { contentSize: { width, height } } }`.

Only called for multiline text inputs.

Type	Required
function	No

onEndEditing

Callback that is called when text input ends.

Type	Required
function	No

onFocus

Callback that is called when the text input is focused.

Type	Required
function	No

onKeyPress

Callback that is called when a key is pressed. This will be called with `{ nativeEvent: { key: keyValue } }` where `keyValue` is `'Enter'` OR `'Backspace'` for respective keys and the typed-in character otherwise including `' '` for space. Fires before `onChange` callbacks. Note: on Android only the inputs from soft keyboard are handled, not the

hardware keyboard inputs.

Type	Required
function	No

onLayout

Invoked on mount and layout changes with `{x, y, width, height}`.

Type	Required
function	No

onScroll

Invoked on content scroll with `{ nativeEvent: { contentOffset: { x, y } } }`. May also contain other properties from ScrollEvent but on Android contentSize is not provided for performance reasons.

Type	Required
function	No

onSelectionChange

Callback that is called when the text input selection is changed. This will be called with `{ nativeEvent: { selection: { start, end } } }`.

Type	Required
function	No

onSubmitEditing

Callback that is called when the text input's submit button is pressed. Invalid if `multiline={true}` is specified.

Type	Required
function	No

placeholder

The string that will be rendered before text input has been entered.

Type	Required
string	No

placeholderTextColor

The text color of the placeholder string.

Type	Required
color	No

returnKeyLabel

Sets the return key to the label. Use it instead of `returnKeyType`.

Type	Required	Platform
string	No	Android

returnKeyType

Determines how the return key should look. On Android you can also use `returnKeyLabel`.

Cross platform

The following values work across platforms:

- `done`
- `go`
- `next`
- `search`
- `send`

Android Only

The following values work on Android only:

- `none`
- `previous`

iOS Only

The following values work on iOS only:

- `default`
- `emergency-call`
- `google`
- `join`
- `route`
- `yahoo`

Type	Required
<code>enum('done', 'go', 'next', 'search', 'send', 'none', 'previous', 'default', 'emergency-call', 'google', 'join', 'route', 'yahoo')</code>	No

scrollEnabled

If `false`, scrolling of the text view will be disabled. The default value is `true`. Only works with `multiline={true}`.

Type	Required	Platform
bool	No	iOS

secureTextEntry

If `true`, the text input obscures the text entered so that sensitive text like passwords stay secure. The default value is `false`. Does not work with `multiline={true}`.

Type	Required
bool	No

selection

The start and end of the text input's selection. Set start and end to the same value to position the cursor.

Type	Required
object: {start: number,end: number}	No

selectionColor

The highlight and cursor color of the text input.

Type	Required
color	No

selectionState

An instance of `DocumentSelectionState`, this is some state that is responsible for maintaining selection information for a document.

Some functionality that can be performed with this instance is:

- `blur()`
- `focus()`
- `update()`

You can reference `DocumentSelectionState` in [vendor/document/selection/DocumentSelectionState.js](#)

Type	Required	Platform

DocumentSelectionState	No	iOS
------------------------	----	-----

selectTextOnFocus

If `true`, all text will automatically be selected on focus.

Type	Required
bool	No

spellCheck

If `false`, disables spell-check style (i.e. red underlines). The default value is inherited from `autoCorrect`.

Type	Required	Platform
bool	No	iOS

textContentType

Give the keyboard and the system information about the expected semantic meaning for the content that users enter.

For iOS 11+ you can set `textContentType` to `username` or `password` to enable autofill of login details from the device keychain.

For iOS 12+ `newPassword` can be used to indicate a new password input the user may want to save in the keychain, and `oneTimeCode` can be used to indicate that a field can be autofilled by a code arriving in an SMS.

To disable autofill, set `textContentType` to `none`.

Possible values for `textContentType` are:

- `none`
- `URL`
- `addressCity`
- `addressCityAndState`
- `addressState`
- `countryName`
- `creditCardNumber`
- `emailAddress`
- `familyName`
- `fullStreetAddress`
- `givenName`
- `jobTitle`
- `location`
- `middleName`
- `name`
- `namePrefix`

- `nameSuffix`
- `nickname`
- `organizationName`
- `postalCode`
- `streetAddressLine1`
- `streetAddressLine2`
- `sublocality`
- `telephoneNumber`
- `username`
- `password`
- `newPassword`
- `oneTimeCode`

Type	Required	Platform
<code>enum('none', 'URL', 'addressCity', 'addressCityAndState', 'addressState', 'countryName', 'creditCardNumber', 'emailAddress', 'familyName', 'fullStreetAddress', 'givenName', 'jobTitle', 'location', 'middleName', 'name', 'namePrefix', 'nameSuffix', 'nickname', 'organizationName', 'postalCode', 'streetAddressLine1', 'streetAddressLine2', 'sublocality', 'telephoneNumber', 'username', 'password')</code>	No	iOS

style

Note that not all Text styles are supported, an incomplete list of what is not supported includes:

- `borderLeftWidth`
- `borderTopWidth`
- `borderRightWidth`
- `borderBottomWidth`
- `borderTopLeftRadius`
- `borderTopRightRadius`
- `borderBottomRightRadius`
- `borderBottomLeftRadius`

see [Issue#7070](#) for more detail.

Styles

Type	Required
<code>Text</code>	No

textBreakStrategy

Set text break strategy on Android API Level 23+, possible values are `simple`, `highQuality`, `balanced`. The default value is `simple`.

Type	Required	Platform
<code>enum('simple', 'highQuality', 'balanced')</code>	No	Android

underlineColorAndroid

The color of the `TextInput` underline.

Type	Required	Platform
<code>color</code>	No	Android

value

The value to show for the text input. `TextInput` is a controlled component, which means the native value will be forced to match this value prop if provided. For most uses, this works great, but in some cases this may cause flickering - one common cause is preventing edits by keeping value the same. In addition to simply setting the same value, either set `editable={false}`, or set/update `maxLength` to prevent unwanted edits without flicker.

Type	Required
<code>string</code>	No

Methods

clear()

```
clear();
```

Removes all text from the `TextInput`.

isFocused()

```
isFocused();
```

Returns `true` if the input is currently focused; `false` otherwise.

ToolbarAndroid

React component that wraps the Android-only `Toolbar` widget. A Toolbar can display a logo, navigation icon (e.g. hamburger menu), a title & subtitle and a list of actions. The title and subtitle are expanded so the logo and navigation icons are displayed on the left, title and subtitle in the middle and the actions on the right.

If the toolbar has an only child, it will be displayed between the title and actions.

Although the Toolbar supports remote images for the logo, navigation and action icons, this should only be used in DEV mode where `require('./some_icon.png')` translates into a packager URL. In release mode you should always use a drawable resource for these icons. Using `require('./some_icon.png')` will do this automatically for you, so as long as you don't explicitly use e.g. `{uri: 'http://...')}`, you will be good.

Example:

```
render: function() {
  return (
    <ToolbarAndroid
      logo={require('./app_logo.png')}
      title="AwesomeApp"
      actions={[{title: 'Settings', icon: require('./icon_settings.png'), show: 'always'}]}
      onActionSelected={this.onActionSelected} />
  )
},
onActionSelected: function(position) {
  if (position === 0) { // index of 'Settings'
    showSettings();
  }
}
```

Props

- [View props...](#)
- `overflowIcon`
- `actions`
- `contentInsetStart`
- `logo`
- `navIcon`
- `onActionSelected`
- `onIconClicked`
- `contentInsetEnd`
- `rtl`
- `subtitle`
- `subtitleColor`
- `testID`
- `title`
- `titleColor`

Reference

Props

overflowIcon

Sets the overflow icon.

Type	Required
optionalImageSource	No

actions

Sets possible actions on the toolbar as part of the action menu. These are displayed as icons or text on the right side of the widget. If they don't fit they are placed in an 'overflow' menu.

This property takes an array of objects, where each object has the following keys:

- `title` : **required**, the title of this action
- `icon` : the icon for this action, e.g. `require('./some_icon.png')`
- `show` : when to show this action as an icon or hide it in the overflow menu: `always` , `ifRoom` OR `never`
- `showWithText` : boolean, whether to show text alongside the icon or not

Type	Required
array of object: {title: string,icon: optionalImageSource,show: enum('always', 'ifRoom', 'never'),showWithText: bool}	No

contentInsetStart

Sets the content inset for the toolbar starting edge.

The content inset affects the valid area for Toolbar content other than the navigation button and menu. Insets define the minimum margin for these components and can be used to effectively align Toolbar content along well-known gridlines.

Type	Required
number	No

logo

Sets the toolbar logo.

Type	Required
optionalImageSource	No

navIcon

Sets the navigation icon.

Type	Required
optionalImageSource	No

onActionSelected

Callback that is called when an action is selected. The only argument that is passed to the callback is the position of the action in the actions array.

Type	Required
function	No

onIconClicked

Callback called when the icon is selected.

Type	Required
function	No

contentInsetEnd

Sets the content inset for the toolbar ending edge.

The content inset affects the valid area for Toolbar content other than the navigation button and menu. Insets define the minimum margin for these components and can be used to effectively align Toolbar content along well-known gridlines.

Type	Required
number	No

rtl

Used to set the toolbar direction to RTL. In addition to this property you need to add

`android:supportsRtl="true"`

to your application `AndroidManifest.xml` and then call `setLayoutDirection(LayoutDirection.RTL)` in your `MainActivity` `onCreate` method.

Type	Required

bool	No
------	----

subtitle

Sets the toolbar subtitle.

Type	Required
string	No

subtitleColor

Sets the toolbar subtitle color.

Type	Required
color	No

testID

Used to locate this view in end-to-end tests.

Type	Required
string	No

title

Sets the toolbar title.

Type	Required
string	No

titleColor

Sets the toolbar title color.

Type	Required
color	No

TouchableHighlight

A wrapper for making views respond properly to touches. On press down, the opacity of the wrapped view is decreased, which allows the underlay color to show through, darkening or tinting the view.

The underlay comes from wrapping the child in a new View, which can affect layout, and sometimes cause unwanted visual artifacts if not used correctly, for example if the backgroundColor of the wrapped view isn't explicitly set to an opaque color.

TouchableHighlight must have one child (not zero or more than one). If you wish to have several child components, wrap them in a View.

Example:

```
renderButton: function() {
  return (
    <TouchableHighlight onPress={this._onPressButton}>
      <Image
        style={styles.button}
        source={require('./myButton.png')}
      />
    </TouchableHighlight>
  );
},
```

Example

```
import React, { Component } from 'react'
import {
  AppRegistry,
  StyleSheet,
  TouchableHighlight,
  Text,
  View,
} from 'react-native'

class App extends Component {
  constructor(props) {
    super(props)
    this.state = { count: 0 }
  }

  onPress = () => {
    this.setState({
      count: this.state.count+1
    })
  }

  render() {
    return (
      <View style={styles.container}>
        <TouchableHighlight
          style={styles.button}
          onPress={this.onPress}
        >
          <Text> Touch Here </Text>
        </TouchableHighlight>
      </View>
    )
  }
}

AppRegistry.registerComponent('App', () => App)
```

```

        </TouchableHighlight>
        <View style={[styles.countContainer]}>
          <Text style={[styles.countText]}>
            { this.state.count !== 0 ? this.state.count: null}
          </Text>
        </View>
      )
    }
  }

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    paddingHorizontal: 10
  },
  button: {
    alignItems: 'center',
    backgroundColor: '#DDDDDD',
    padding: 10
  },
  countContainer: {
    alignItems: 'center',
    padding: 10
  },
  countText: {
    color: '#FF00FF'
  }
})

AppRegistry.registerComponent('App', () => App)

```

Props

- TouchableWithoutFeedback props...

- `activeOpacity`
 - `onHideUnderlay`
 - `onShowUnderlay`
 - `style`
 - `underlayColor`
 - `hasTVPreferredFocus`
 - `tvParallaxProperties`
-

Reference

Props

activeOpacity

Determines what the opacity of the wrapped view should be when touch is active. The value should be between 0 and 1. Defaults to 0.85.

Type	Required
number	No

onHideUnderlay

Called immediately after the underlay is hidden.

Type	Required
function	No

onShowUnderlay

Called immediately after the underlay is shown.

Type	Required
function	No

style

Type	Required
View.style	No

underlayColor

The color of the underlay that will show through when the touch is active.

Type	Required
color	No

hasTVPREFERREDFocus

(Apple TV only) TV preferred focus (see documentation for the View component).

Type	Required	Platform
bool	No	iOS

tvParallaxProperties

(Apple TV only) Object with properties to control Apple TV parallax effects.

enabled: If true, parallax effects are enabled. Defaults to true. shiftDistanceX: Defaults to 2.0. shiftDistanceY: Defaults to 2.0. tiltAngle: Defaults to 0.05. magnification: Defaults to 1.0. pressMagnification: Defaults to 1.0. pressDuration: Defaults to 0.3. pressDelay: Defaults to 0.0.

Type	Required	Platform
object	No	iOS

TouchableNativeFeedback

A wrapper for making views respond properly to touches (Android only). On Android this component uses native state drawable to display touch feedback.

At the moment it only supports having a single View instance as a child node, as it's implemented by replacing that View with another instance of RCTView node with some additional properties set.

Background drawable of native feedback touchable can be customized with `background` property.

Example:

```
renderButton: function() {
  return (
    <TouchableNativeFeedback
      onPress={this._onPressButton}
      background={TouchableNativeFeedback.SelectableBackground()}>
      <View style={{width: 150, height: 100, backgroundColor: 'red'}}>
        <Text style={{margin: 30}}>Button</Text>
      </View>
    </TouchableNativeFeedback>
  );
},
```

Props

- `TouchableWithoutFeedback` props...
- `background`
- `useForeground`

Methods

- `SelectableBackground`
- `SelectableBackgroundBorderless`
- `Ripple`
- `canUseNativeForeground`

Reference

Props

background

Determines the type of background drawable that's going to be used to display feedback. It takes an object with `type` property and extra data depending on the `type`. It's recommended to use one of the static methods to generate that dictionary.

Type	Required
backgroundPropType	No

useForeground

Set to true to add the ripple effect to the foreground of the view, instead of the background. This is useful if one of your child views has a background of its own, or you're e.g. displaying images, and you don't want the ripple to be covered by them.

Check `TouchableNativeFeedback.canUseNativeForeground()` first, as this is only available on Android 6.0 and above. If you try to use this on older versions you will get a warning and fallback to background.

Type	Required
bool	No

Methods

SelectableBackground()

```
static SelectableBackground()
```

Creates an object that represents android theme's default background for selectable elements (`? android:attr/selectableItemBackground`).

SelectableBackgroundBorderless()

```
static SelectableBackgroundBorderless()
```

Creates an object that represent android theme's default background for borderless selectable elements (`? android:attr/selectableItemBackgroundBorderless`). Available on android API level 21+.

Ripple()

```
static Ripple(color: string, borderless: boolean)
```

Creates an object that represents ripple drawable with specified color (as a string). If property `borderless` evaluates to true the ripple will render outside of the view bounds (see native actionbar buttons as an example of that behavior). This background type is available on Android API level 21+.

Parameters:

Name	Type	Required	Description
color	string	Yes	The ripple color
borderless	boolean	Yes	If the ripple can render outside it's bounds

canUseNativeForeground()

```
static canUseNativeForeground()
```

TouchableOpacity

A wrapper for making views respond properly to touches. On press down, the opacity of the wrapped view is decreased, dimming it.

Opacity is controlled by wrapping the children in an Animated.View, which is added to the view hierarchy. Be aware that this can affect layout.

Example:

```
renderButton: function() {
  return (
    <TouchableOpacity onPress={this._onPressButton}>
      <Image
        style={styles.button}
        source={require('../myButton.png')}>
    />
    </TouchableOpacity>
  );
},
```

Example

```
import React, { Component } from 'react'
import {
  AppRegistry,
  StyleSheet,
  TouchableOpacity,
  Text,
  View,
} from 'react-native'

class App extends Component {
  constructor(props) {
    super(props)
    this.state = { count: 0 }
  }

  onPress = () => {
    this.setState({
      count: this.state.count+1
    })
  }

  render() {
    return (
      <View style={styles.container}>
        <TouchableOpacity
          style={styles.button}
          onPress={this.onPress}>
        </TouchableOpacity>
        <View style={[styles.countContainer]}>
          <Text style={[styles.countText]}>
            { this.state.count !== 0 ? this.state.count: null}
          </Text>
        </View>
      </View>
    )
  }
}

AppRegistry.registerComponent('App', () => App)
```

```

        </View>
      </View>
    )
}
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    paddingHorizontal: 10
  },
  button: {
    alignItems: 'center',
    backgroundColor: '#DDDDDD',
    padding: 10
  },
  countContainer: {
    alignItems: 'center',
    padding: 10
  },
  countText: {
    color: '#FF00FF'
  }
})

AppRegistry.registerComponent('App', () => App)

```

Props

- `TouchableWithoutFeedback` props...
- `activeOpacity`
- `tvParallaxProperties`
- `hasTVPreferredFocus`

Methods

- `setOpacityTo`

Reference

Props

activeOpacity

Determines what the opacity of the wrapped view should be when touch is active. Defaults to 0.2.

Type	Required
number	No

tvParallaxProperties

(*Apple TV only*) Object with properties to control Apple TV parallax effects.

enabled: If true, parallax effects are enabled. Defaults to true. shiftDistanceX: Defaults to 2.0. shiftDistanceY: Defaults to 2.0. tiltAngle: Defaults to 0.05. magnification: Defaults to 1.0. pressMagnification: Defaults to 1.0. pressDuration: Defaults to 0.3. pressDelay: Defaults to 0.0.

Type	Required	Platform
object	No	iOS

hasTVPreferredFocus

(*Apple TV only*) TV preferred focus (see documentation for the View component).

Type	Required	Platform
bool	No	iOS

Methods

setOpacityTo()

```
setOpacityTo((value: number), (duration: number));
```

Animate the touchable to a new opacity.

TouchableWithoutFeedback

Do not use unless you have a very good reason. All elements that respond to press should have a visual feedback when touched.

TouchableWithoutFeedback supports only one child. If you wish to have several child components, wrap them in a View.

Props

- `accessibilityComponentType`
- `accessibilityHint`
- `accessibilityLabel`
- `accessibilityRole`
- `accessibilityStates`
- `accessibilityTraits`
- `accessible`
- `delayLongPress`
- `delayPressIn`
- `delayPressOut`
- `disabled`
- `hitSlop`
- `onBlur`
- `onFocus`
- `onLayout`
- `onLongPress`
- `onPress`
- `onPressIn`
- `onPressOut`
- `pressRetentionOffset`

Type Definitions

- `Event`
-

Reference

Props

accessibilityComponentType

> Note: `accessibilityComponentType` will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead.

Type	Required
------	----------

AccessibilityComponentTypes	No
-----------------------------	----

accessibilityHint

An accessibility hint helps users understand what will happen when they perform an action on the accessibility element when that result is not obvious from the accessibility label.

Type	Required
string	No

accessibilityLabel

Overrides the text that's read by the screen reader when the user interacts with the element. By default, the label is constructed by traversing all the children and accumulating all the `Text` nodes separated by space.

Type	Required
node	No

accessibilityRole

Type	Required
AccessibilityRoles	No

accessibilityStates

Type	Required
array of AccessibilityStates	No

accessibilityTraits

> Note: `accessibilityTraits` will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead.

Type	Required
AccessibilityTraits, ,array of AccessibilityTraits	No

accessible

Type	Required

bool	No
------	----

delayLongPress

Delay in ms, from onPressIn, before onLongPress is called.

Type	Required
number	No

delayPressIn

Delay in ms, from the start of the touch, before onPressIn is called.

Type	Required
number	No

delayPressOut

Delay in ms, from the release of the touch, before onPressOut is called.

Type	Required
number	No

disabled

If true, disable all interactions for this component.

Type	Required
bool	No

hitSlop

This defines how far your touch can start away from the button. This is added to `pressRetentionOffset` when moving off of the button. **NOTE** The touch area never extends past the parent view bounds and the Z-index of sibling views always takes precedence if a touch hits two overlapping views.

Type	Required
object: {top: number, left: number, bottom: number, right: number}	No

onBlur

Invoked when the item loses focus.

Type	Required
function	No

onFocus

Invoked when the item receives focus.

Type	Required
function	No

onLayout

Invoked on mount and layout changes with

```
{nativeEvent: {layout: {x, y, width, height}}}
```

Type	Required
function	No

onLongPress

Type	Required
function	No

onPress

Called when the touch is released, but not if cancelled (e.g. by a scroll that steals the responder lock).

Type	Required
function	No

onPressIn

Called as soon as the touchable element is pressed and invoked even before onPress. This can be useful when making network requests.

Type	Required
function	No

onPressOut

Called as soon as the touch is released even before onPress.

Type	Required
function	No

pressRetentionOffset

When the scroll view is disabled, this defines how far your touch may move off of the button, before deactivating the button. Once deactivated, try moving it back and you'll see that the button is once again reactivated! Move it back and forth several times while the scroll view is disabled. Ensure you pass in a constant to reduce memory allocations.

Type	Required
object: {top: number, left: number, bottom: number, right: number}	No

Type Definitions

Event

Type
Object

View

The most fundamental component for building a UI, `View` is a container that supports layout with [flexbox](#), [style](#), [some touch handling](#), and [accessibility](#) controls. `View` maps directly to the native view equivalent on whatever platform React Native is running on, whether that is a `UIView`, `<div>`, `android.view`, etc.

`View` is designed to be nested inside other views and can have 0 to many children of any type.

This example creates a `View` that wraps two colored boxes and a text component in a row with padding.

```
class ViewColoredBoxesWithText extends Component {
  render() {
    return (
      <View
        style={{
          flexDirection: 'row',
          height: 100,
          padding: 20,
        }}>
        <View style={{backgroundColor: 'blue', flex: 0.3}} />
        <View style={{backgroundColor: 'red', flex: 0.5}} />
        <Text>Hello World!</Text>
      </View>
    );
  }
}
```

`View`s are designed to be used with [StyleSheet](#) for clarity and performance, although inline styles are also supported.

Synthetic Touch Events

For `View` responder props (e.g., `onResponderMove`), the synthetic touch event passed to them are of the following form:

- `nativeEvent`
 - `changedTouches` - Array of all touch events that have changed since the last event.
 - `identifier` - The ID of the touch.
 - `locationX` - The X position of the touch, relative to the element.
 - `locationY` - The Y position of the touch, relative to the element.
 - `pageX` - The X position of the touch, relative to the root element.
 - `pageY` - The Y position of the touch, relative to the root element.
 - `target` - The node id of the element receiving the touch event.
 - `timestamp` - A time identifier for the touch, useful for velocity calculation.
 - `touches` - Array of all current touches on the screen.

Props

- `onStartShouldSetResponder`
- `accessibilityLabel`
- `accessibilityHint`

- `hitSlop`
 - `nativeID`
 - `onAccessibilityTap`
 - `onLayout`
 - `onMagicTap`
 - `onMoveShouldSetResponder`
 - `onMoveShouldSetResponderCapture`
 - `onResponderGrant`
 - `onResponderMove`
 - `onResponderReject`
 - `onResponderRelease`
 - `onResponderTerminate`
 - `onResponderTerminationRequest`
 - `accessible`
 - `onStartShouldSetResponderCapture`
 - `pointerEvents`
 - `removeClippedSubviews`
 - `style`
 - `testID`
 - `accessibilityComponentType`
 - `accessibilityLiveRegion`
 - `collapsible`
 - `importantForAccessibility`
 - `needsOffscreenAlphaCompositing`
 - `renderToHardwareTextureAndroid`
 - `accessibilityRole`
 - `accessibilityStates`
 - `accessibilityTraits`
 - `accessibilityViewIsModal`
 - `accessibilityElementsHidden`
 - `accessibilityIgnoresInvertColors`
 - `shouldRasterizeIOS`
-

Reference

Props

`onStartShouldSetResponder`

Does this view want to become responder on the start of a touch?

`View.props.onStartShouldSetResponder: (event) => [true | false]`, where `event` is a synthetic touch event as described above.

Type	Required
<code>function</code>	No

accessibilityHint

An accessibility hint helps users understand what will happen when they perform an action on the accessibility element when that result is not obvious from the accessibility label.

Type	Required
string	No

accessibilityLabel

Overrides the text that's read by the screen reader when the user interacts with the element. By default, the label is constructed by traversing all the children and accumulating all the `Text` nodes separated by space.

Type	Required
node	No

hitSlop

This defines how far a touch event can start away from the view. Typical interface guidelines recommend touch targets that are at least 30 - 40 points/density-independent pixels.

For example, if a touchable view has a height of 20 the touchable height can be extended to 40 with `hitSlop={{top: 10, bottom: 10, left: 0, right: 0}}`

The touch area never extends past the parent view bounds and the Z-index of sibling views always takes precedence if a touch hits two overlapping views.

Type	Required
<code>object: {top: number, left: number, bottom: number, right: number}</code>	No

nativeID

Used to locate this view from native classes.

This disables the 'layout-only view removal' optimization for this view!

Type	Required
string	No

onAccessibilityTap

When `accessible` is true, the system will try to invoke this function when the user performs accessibility tap gesture.

Type	Required
function	No

onLayout

Invoked on mount and layout changes with:

```
{nativeEvent: { layout: {x, y, width, height}}}
```

This event is fired immediately once the layout has been calculated, but the new layout may not yet be reflected on the screen at the time the event is received, especially if a layout animation is in progress.

Type	Required
function	No

onMagicTap

When `accessible` is `true`, the system will invoke this function when the user performs the magic tap gesture.

Type	Required
function	No

onMoveShouldSetResponder

Does this view want to "claim" touch responsiveness? This is called for every touch move on the `View` when it is not the responder.

```
View.props.onMoveShouldSetResponder: (event) => [true | false], where event is a synthetic touch event as described above.
```

Type	Required
function	No

onMoveShouldSetResponderCapture

If a parent `View` wants to prevent a child `View` from becoming responder on a move, it should have this handler which returns `true`.

```
View.props.onMoveShouldSetResponderCapture: (event) => [true | false], where event is a synthetic touch event as described above.
```

Type	Required
function	No

onResponderGrant

The View is now responding for touch events. This is the time to highlight and show the user what is happening.

`View.props.onResponderGrant: (event) => {}`, where `event` is a synthetic touch event as described above.

Type	Required
function	No

onResponderMove

The user is moving their finger.

`View.props.onResponderMove: (event) => {}`, where `event` is a synthetic touch event as described above.

Type	Required
function	No

onResponderReject

Another responder is already active and will not release it to that `view` asking to be the responder.

`View.props.onResponderReject: (event) => {}`, where `event` is a synthetic touch event as described above.

Type	Required
function	No

onResponderRelease

Fired at the end of the touch.

`View.props.onResponderRelease: (event) => {}`, where `event` is a synthetic touch event as described above.

Type	Required
function	No

onResponderTerminate

The responder has been taken from the `view`. Might be taken by other views after a call to `onResponderTerminationRequest`, or might be taken by the OS without asking (e.g., happens with control center/notification center on iOS)

`View.props.onResponderTerminate: (event) => {}`, where `event` is a synthetic touch event as described above.

Type	Required

function	No
----------	----

onResponderTerminationRequest

Some other `view` wants to become responder and is asking this `view` to release its responder. Returning `true` allows its release.

`View.props.onResponderTerminationRequest: (event) => {}`, where `event` is a synthetic touch event as described above.

Type	Required
function	No

accessible

When `true`, indicates that the view is an accessibility element. By default, all the touchable elements are accessible.

Type	Required
bool	No

onStartShouldSetResponderCapture

If a parent `view` wants to prevent a child `view` from becoming responder on a touch start, it should have this handler which returns `true`.

`View.props.onStartShouldSetResponderCapture: (event) => [true | false]`, where `event` is a synthetic touch event as described above.

Type	Required
function	No

pointerEvents

Controls whether the `view` can be the target of touch events.

- `'auto'` : The View can be the target of touch events.
- `'none'` : The View is never the target of touch events.
- `'box-none'` : The View is never the target of touch events but its subviews can be. It behaves like if the view had the following classes in CSS:

```
.box-none {  
    pointer-events: none;  
}  
.box-none * {
```

```
    pointer-events: all;
}
```

- 'box-only' : The view can be the target of touch events but its subviews cannot be. It behaves like if the view had the following classes in CSS:

```
.box-only {
  pointer-events: all;
}
.box-only * {
  pointer-events: none;
}
```

Since `pointerEvents` does not affect layout/appearance, and we are already deviating from the spec by adding additional modes, we opt to not include `pointerEvents` on `style`. On some platforms, we would need to implement it as a `className` anyways. Using `style` or not is an implementation detail of the platform.

Type	Required
enum('box-none', 'none', 'box-only', 'auto')	No

removeClippedSubviews

This is a special performance property exposed by `RCTView` and is useful for scrolling content when there are many subviews, most of which are offscreen. For this property to be effective, it must be applied to a view that contains many subviews that extend outside its bound. The subviews must also have `overflow: hidden`, as should the containing view (or one of its superviews).

Type	Required
bool	No

style

Type	Required
view styles	No

testID

Used to locate this view in end-to-end tests.

This disables the 'layout-only view removal' optimization for this view!

Type	Required
string	No

accessibilityComponentType

> Note: `accessibilityComponentType` will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead.

Indicates to accessibility services to treat UI component like a native one. Works for Android only.

Possible values are one of:

- `'none'`
- `'button'`
- `'radiobutton_checked'`
- `'radiobutton_unchecked'`

Type	Required	Platform
AccessibilityComponentTypes	No	Android

accessibilityLiveRegion

Indicates to accessibility services whether the user should be notified when this view changes. Works for Android API >= 19 only. Possible values:

- `'none'` - Accessibility services should not announce changes to this view.
- `'polite'` - Accessibility services should announce changes to this view.
- `'assertive'` - Accessibility services should interrupt ongoing speech to immediately announce changes to this view.

See the [Android View docs](#) for reference.

Type	Required	Platform
enum('none', 'polite', 'assertive')	No	Android

collapsible

Views that are only used to layout their children or otherwise don't draw anything may be automatically removed from the native hierarchy as an optimization. Set this property to `false` to disable this optimization and ensure that this `View` exists in the native view hierarchy.

Type	Required	Platform
bool	No	Android

importantForAccessibility

Controls how view is important for accessibility which is if it fires accessibility events and if it is reported to accessibility services that query the screen. Works for Android only.

Possible values:

- `'auto'` - The system determines whether the view is important for accessibility - default (recommended).
- `'yes'` - The view is important for accessibility.
- `'no'` - The view is not important for accessibility.
- `'no-hide-descendants'` - The view is not important for accessibility, nor are any of its descendant views.

See the [Android `importantForAccessibility` docs](#) for reference.

Type	Required	Platform
<code>enum('auto', 'yes', 'no', 'no-hide-descendants')</code>	No	Android

needsOffscreenAlphaCompositing

Whether this `view` needs to be rendered offscreen and composited with an alpha in order to preserve 100% correct colors and blending behavior. The default (`false`) falls back to drawing the component and its children with an alpha applied to the paint used to draw each element instead of rendering the full component offscreen and compositing it back with an alpha value. This default may be noticeable and undesired in the case where the `view` you are setting an opacity on has multiple overlapping elements (e.g. multiple overlapping `view`s, or text and a background).

Rendering offscreen to preserve correct alpha behavior is extremely expensive and hard to debug for non-native developers, which is why it is not turned on by default. If you do need to enable this property for an animation, consider combining it with `renderToHardwareTextureAndroid` if the view **contents** are static (i.e. it doesn't need to be redrawn each frame). If that property is enabled, this View will be rendered off-screen once, saved in a hardware texture, and then composited onto the screen with an alpha each frame without having to switch rendering targets on the GPU.

Type	Required	Platform
<code>bool</code>	No	Android

renderToHardwareTextureAndroid

Whether this `view` should render itself (and all of its children) into a single hardware texture on the GPU.

On Android, this is useful for animations and interactions that only modify opacity, rotation, translation, and/or scale: in those cases, the view doesn't have to be redrawn and display lists don't need to be re-executed. The texture can just be re-used and re-composited with different parameters. The downside is that this can use up limited video memory, so this prop should be set back to false at the end of the interaction/animation.

Type	Required	Platform
<code>bool</code>	No	Android

accessibilityRole

> Note: `AccessibilityRole` and `AccessibilityStates` are meant to be a cross-platform solution to replace `accessibilityTraits` and `accessibilityComponentType`, which will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead of `accessibilityTraits` and `accessibilityComponentType`.

Tells the screen reader to treat the currently focused on element as having a specific role.

Possible values for `AccessibilityRole` is one of:

- `'none'` - The element has no role.
- `'button'` - The element should be treated as a button.
- `'link'` - The element should be treated as a link.
- `'header'` - The element is a header that divides content into sections.
- `'search'` - The element should be treated as a search field.
- `'image'` - The element should be treated as an image.
- `'key'` - The element should be treated like a keyboard key.
- `'text'` - The element should be treated as text.
- `'summary'` - The element provides app summary information.
- `'imagebutton'` - The element has the role of both an image and also a button.
- `'adjustable'` - The element allows adjustment over a range of values.

On iOS, these roles map to corresponding Accessibility Traits. Image button has the same functionality as if the trait was set to both 'image' and 'button'. See the [Accessibility guide](#) for more information.

On Android, these roles have similar functionality on TalkBack as adding Accessibility Traits does on Voiceover in iOS

Type	Required
<code>AccessibilityRole</code>	No

accessibilityStates

> Note: `AccessibilityRole` and `AccessibilityStates` are meant to be a cross-platform solution to replace `accessibilityTraits` and `accessibilityComponentType`, which will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead of `accessibilityTraits` and `accessibilityComponentType`.

Tells the screen reader to treat the currently focused on element as being in a specific state.

You can provide one state, no state, or both states. The states must be passed in through an array. Ex: `['selected']` or `['selected', 'disabled']`

Possible values for `AccessibilityStates` are:

- `'selected'` - The element is in a selected state.
- `'disabled'` - The element is in a disabled state.

Type	Required
array of <code>AccessibilitStates</code>	No

accessibilityTraits

> Note: `accessibilityTraits` will soon be deprecated. When possible, use `accessibilityRole` and `accessibilityStates` instead.

Provides additional traits to screen reader. By default no traits are provided unless specified otherwise in element.

You can provide one trait or an array of many traits.

Possible values for `AccessibilityTraits` are:

- `'none'` - The element has no traits.
- `'button'` - The element should be treated as a button.
- `'link'` - The element should be treated as a link.
- `'header'` - The element is a header that divides content into sections.
- `'search'` - The element should be treated as a search field.
- `'image'` - The element should be treated as an image.
- `'selected'` - The element is selected.
- `'plays'` - The element plays sound.
- `'key'` - The element should be treated like a keyboard key.
- `'text'` - The element should be treated as text.
- `'summary'` - The element provides app summary information.
- `'disabled'` - The element is disabled.
- `'frequentUpdates'` - The element frequently changes its value.
- `'startsMedia'` - The element starts a media session.
- `'adjustable'` - The element allows adjustment over a range of values.
- `'allowsDirectInteraction'` - The element allows direct touch interaction for VoiceOver users.
- `'pageTurn'` - Informs VoiceOver that it should scroll to the next page when it finishes reading the contents of the element.

See the [Accessibility guide](#) for more information.

Type	Required	Platform
<code>AccessibilityTraits</code> , ,array of <code>AccessibilityTraits</code>	No	iOS

accessibilityViewIsModal

A value indicating whether VoiceOver should ignore the elements within views that are siblings of the receiver.

Default is `false`.

See the [Accessibility guide](#) for more information.

Type	Required	Platform
<code>bool</code>	No	iOS

accessibilityElementsHidden

A value indicating whether the accessibility elements contained within this accessibility element are hidden.

Default is `false`.

See the [Accessibility guide](#) for more information.

Type	Required	Platform
bool	No	iOS

accessibilityIgnoresInvertColors

A value indicating this view should or should not be inverted when color inversion is turned on. A value of `true` will tell the view to not be inverted even if color inversion is turned on.

See the [Accessibility guide](#) for more information.

Type	Required	Platform
bool	No	iOS

shouldRasterizeIOS

Whether this `view` should be rendered as a bitmap before compositing.

On iOS, this is useful for animations and interactions that do not modify this component's dimensions nor its children; for example, when translating the position of a static view, rasterization allows the renderer to reuse a cached bitmap of a static view and quickly composite it during each frame.

Rasterization incurs an off-screen drawing pass and the bitmap consumes memory. Test and measure when using this property.

Type	Required	Platform
bool	No	iOS

ViewPagerAndroid

Container that allows to flip left and right between child views. Each child view of the `ViewPagerAndroid` will be treated as a separate page and will be stretched to fill the `ViewPagerAndroid`.

It is important all children are `<View>`s and not composite components. You can set style properties like `padding` or `backgroundColor` for each child. It is also important that each child have a `key` prop.

Example:

```
render: function() {
  return (
    <ViewPagerAndroid
      style={styles.viewPager}
      initialPage={0}>
      <View style={styles.pageStyle} key="1">
        <Text>First page</Text>
      </View>
      <View style={styles.pageStyle} key="2">
        <Text>Second page</Text>
      </View>
    </ViewPagerAndroid>
  );
}

...

var styles = {
  ...
  viewPager: {
    flex: 1
  },
  pageStyle: {
    alignItems: 'center',
    padding: 20,
  }
}
```

Props

- [View props...](#)
- `initialPage`
- `keyboardDismissMode`
- `onPageScroll`
- `onPageScrollStateChanged`
- `onPageSelected`
- `pageMargin`
- `peekEnabled`
- `scrollEnabled`
- `setPage`
- `setPageWithoutAnimation`

Type Definitions

- [ViewPageScrollState](#)
-

Reference

Props

initialPage

Index of initial page that should be selected. Use `setPage` method to update the page, and `onPageSelected` to monitor page changes

Type	Required
number	No

keyboardDismissMode

Determines whether the keyboard gets dismissed in response to a drag.

- 'none' (the default), drags do not dismiss the keyboard.
- 'on-drag', the keyboard is dismissed when a drag begins.

Type	Required
enum('none', 'on-drag')	No

onPageScroll

Executed when transitioning between pages (either because of animation for the requested page change or when user is swiping/dragging between pages) The `event.nativeEvent` object for this callback will carry following data:

- position - index of first page from the left that is currently visible
- offset - value from range [0, 1] describing stage between page transitions. Value x means that $(1 - x)$ fraction of the page at "position" index is visible, and x fraction of the next page is visible.

Type	Required
function	No

onPageScrollStateChanged

Function called when the page scrolling state has changed. The page scrolling state can be in 3 states:

- idle, meaning there is no interaction with the page scroller happening at the time

- dragging, meaning there is currently an interaction with the page scroller
- settling, meaning that there was an interaction with the page scroller, and the page scroller is now finishing its closing or opening animation

Type	Required
function	No

onPageSelected

This callback will be called once ViewPager finish navigating to selected page (when user swipes between pages). The `event.nativeEvent` object passed to this callback will have following fields:

- position - index of page that has been selected

Type	Required
function	No

pageMargin

Blank space to show between pages. This is only visible while scrolling, pages are still edge-to-edge.

Type	Required
number	No

peekEnabled

Whether enable showing peekFraction or not. If this is true, the preview of last and next page will show in current screen. Defaults to false.

Type	Required
bool	No

scrollEnabled

When false, the content does not scroll. The default value is true.

Type	Required
bool	No

setPage

A helper function to scroll to a specific page in the ViewPager. The transition between pages will be animated.

- position - index of page that will be selected

Type	Required
Number	Yes

setPageWithoutAnimation

A helper function to scroll to a specific page in the ViewPager. The transition between pages will *not* be animated.

- position - index of page that will be selected

Type	Required
Number	Yes

Type Definitions

ViewPagerScrollState

Type
\$Enum

Constants:

Value	Description
idle	
dragging	
settling	

VirtualizedList

Base implementation for the more convenient `<FlatList>` and `<SectionList>` components, which are also better documented. In general, this should only really be used if you need more flexibility than `FlatList` provides, e.g. for use with immutable data instead of plain arrays.

Virtualization massively improves memory consumption and performance of large lists by maintaining a finite render window of active items and replacing all items outside of the render window with appropriately sized blank space. The window adapts to scrolling behavior, and items are rendered incrementally with low-pri (after any running interactions) if they are far from the visible area, or with hi-pri otherwise to minimize the potential of seeing blank space.

Some caveats:

- Internal state is not preserved when content scrolls out of the render window. Make sure all your data is captured in the item data or external stores like Flux, Redux, or Relay.
- This is a `PureComponent` which means that it will not re-render if `props` remain shallow- equal. Make sure that everything your `renderItem` function depends on is passed as a prop (e.g. `extraData`) that is not `==` after updates, otherwise your UI may not update on changes. This includes the `data` prop and parent component state.
- In order to constrain memory and enable smooth scrolling, content is rendered asynchronously offscreen. This means it's possible to scroll faster than the fill rate and momentarily see blank content. This is a tradeoff that can be adjusted to suit the needs of each application, and we are working on improving it behind the scenes.
- By default, the list looks for a `key` prop on each item and uses that for the React key. Alternatively, you can provide a custom `keyExtractor` prop.

Props

- `ScrollView` props...
- `renderItem`
- `data`
- `getItem`
- `getItemCount`
- `debug`
- `extraData`
- `getItemLayout`
- `initialScrollIndex`
- `inverted`
- `CellRendererComponent`
- `ListEmptyComponent`
- `ListFooterComponent`
- `ListHeaderComponent`
- `onEndReached`
- `onLayout`
- `onRefresh`
- `onScrollToIndexFailed`
- `onViewableItemsChanged`

- `refreshing`
- `removeClippedSubviews`
- `renderScrollIndicator`
- `viewabilityConfig`
- `viewabilityConfigCallbackPairs`
- `horizontal`
- `initialNumToRender`
- `keyExtractor`
- `maxToRenderPerBatch`
- `onEndReachedThreshold`
- `updateCellsBatchingPeriod`
- `windowSize`
- `disableVirtualization`
- `progressViewOffset`

Methods

- `scrollToEnd`
 - `scrollToIndex`
 - `scrollToItem`
 - `scrollToOffset`
 - `recordInteraction`
 - `flashScrollIndicators`
-

Reference

Props

`renderItem`

```
(info: any) => ?React.Element<any>
```

Takes an item from `data` and renders it into the list

Type	Required
function	Yes

`data`

The default accessor functions assume this is an array of objects with shape `{key: string}` but you can override `getItem`, `getCount`, and `keyExtractor` to handle any type of index-based data.

Type	Required

any	Yes
-----	-----

getItem

```
(data: any, index: number) => object;
```

A generic accessor for extracting an item from any sort of data blob.

Type	Required
function	Yes

getItemCount

```
(data: any) => number;
```

Determines how many items are in the data blob.

Type	Required
function	Yes

debug

`debug` will turn on extra logging and visual overlays to aid with debugging both usage and implementation, but with a significant perf hit.

Type	Required
boolean	No

extraData

A marker property for telling the list to re-render (since it implements `PureComponent`). If any of your `renderItem`, Header, Footer, etc. functions depend on anything outside of the `data` prop, stick it here and treat it immutably.

Type	Required
any	No

getItemLayout

```
(  
  data: any,  
  index: number,  
) => {length: number, offset: number, index: number}
```

Type	Required
function	No

initialScrollIndex

Instead of starting at the top with the first item, start at `initialScrollIndex`. This disables the "scroll to top" optimization that keeps the first `initialNumToRender` items always rendered and immediately renders the items starting at this initial index. Requires `getItemLayout` to be implemented.

Type	Required
number	No

inverted

Reverses the direction of scroll. Uses scale transforms of -1.

Type	Required
boolean	No

CellRendererComponent

Each cell is rendered using this element. Can be a React Component Class, or a render function. Defaults to using `View`.

Type	Required
component, function	No

ListEmptyComponent

Rendered when the list is empty. Can be a React Component Class, a render function, or a rendered element.

Type	Required
component, function, element	No

ListFooterComponent

Rendered at the bottom of all the items. Can be a React Component Class, a render function, or a rendered element.

Type	Required
component, function, element	No

ListHeaderComponent

Rendered at the top of all the items. Can be a React Component Class, a render function, or a rendered element.

Type	Required
component, function, element	No

onLayout

Type	Required
function	No

onRefresh

```
() => void
```

If provided, a standard `RefreshControl` will be added for "Pull to Refresh" functionality. Make sure to also set the `refreshing` prop correctly.

Type	Required
function	No

onScrollToIndexFailed

```
(info: {  
  index: number,  
  highestMeasuredFrameIndex: number,  
  averageItemLength: number,  
}) => void
```

Used to handle failures when scrolling to an index that has not been measured yet. Recommended action is to either compute your own offset and `scrollTo` it, or scroll as far as possible and then try again after more items have been rendered.

Type	Required

function	No
----------	----

onViewableItemsChanged

```
(info: {  
  viewableItems: array,  
  changed: array,  
}) => void
```

Called when the viewability of rows changes, as defined by the `viewabilityConfig` prop.

Type	Required
function	No

refreshing

Set this true while waiting for new data from a refresh.

Type	Required
boolean	No

removeClippedSubviews

This may improve scroll performance for large lists.

Note: May have bugs (missing content) in some circumstances - use at your own risk.

Type	Required
boolean	No

renderScrollIndicator

```
(props: object) => element;
```

Render a custom scroll component, e.g. with a differently styled `RefreshControl`.

Type	Required
function	No

viewabilityConfig

See `ViewabilityHelper.js` for flow type and further documentation.

Type	Required
<code>ViewabilityConfig</code>	No

viewabilityConfigCallbackPairs

List of `ViewabilityConfig / onViewableItemsChanged` pairs. A specific `onViewableItemsChanged` will be called when its corresponding `ViewabilityConfig`'s conditions are met. See `ViewabilityHelper.js` for flow type and further documentation.

Type	Required
<code>array of ViewabilityConfigCallbackPair</code>	No

horizontal

Type	Required
<code>boolean</code>	No

initialNumToRender

How many items to render in the initial batch. This should be enough to fill the screen but not much more. Note these items will never be unmounted as part of the windowed rendering in order to improve perceived performance of scroll-to-top actions.

Type	Required
<code>number</code>	No

keyExtractor

```
(item: object, index: number) => string;
```

Used to extract a unique key for a given item at the specified index. Key is used for caching and as the react key to track item re-ordering. The default extractor checks `item.key`, then falls back to using the index, like React does.

Type	Required
<code>function</code>	No

maxToRenderPerBatch

The maximum number of items to render in each incremental render batch. The more rendered at once, the better the fill rate, but responsiveness may suffer because rendering content may interfere with responding to button taps or other interactions.

Type	Required
number	No

onEndReached

```
(info: {distanceFromEnd: number}) => void
```

Called once when the scroll position gets within `onEndReachedThreshold` of the rendered content.

Type	Required
function	No

onEndReachedThreshold

How far from the end (in units of visible length of the list) the bottom edge of the list must be from the end of the content to trigger the `onEndReached` callback. Thus a value of 0.5 will trigger `onEndReached` when the end of the content is within half the visible length of the list.

Type	Required
number	No

updateCellsBatchingPeriod

Amount of time between low-pri item render batches, e.g. for rendering items quite a ways off screen. Similar fill rate/responsiveness tradeoff as `maxToRenderPerBatch`.

Type	Required
number	No

windowSize

Determines the maximum number of items rendered outside of the visible area, in units of visible lengths. So if your list fills the screen, then `windowSize={21}` (the default) will render the visible screen area plus up to 10 screens above and 10 below the viewport. Reducing this number will reduce memory consumption and may

improve performance, but will increase the chance that fast scrolling may reveal momentary blank areas of unrendered content.

Type	Required
number	No

disableVirtualization

DEPRECATED. Virtualization provides significant performance and memory optimizations, but fully unmounts react instances that are outside of the render window. You should only need to disable this for debugging purposes.

Type	Required
	No

progressViewOffset

Set this when offset is needed for the loading indicator to show correctly.

Type	Required	Platform
number	No	Android

Methods

scrollToEnd()

```
scrollToEnd(([params]: object));
```

scrollToIndex()

```
scrollToIndex((params: object));
```

scrollToItem()

```
scrollToItem((params: object));
```

scrollToOffset()

```
scrollToOffset((params: object));
```

Scroll to a specific content pixel offset in the list.

Param `offset` expects the offset to scroll to. In case of `horizontal` is true, the offset is the x-value, in any other case the offset is the y-value.

Param `animated` (`true` by default) defines whether the list should do an animation while scrolling.

recordInteraction()

```
recordInteraction();
```

flashScrollIndicators()

```
flashScrollIndicators();
```

WebView

`WebView` renders web content in a native view.

```
import React, { Component } from 'react';
import { WebView } from 'react-native';

class MyWeb extends Component {
  render() {
    return (
      <WebView
        source={{uri: 'https://github.com/facebook/react-native'}}
        style={{marginTop: 20}}
      />
    );
  }
}
```

Minimal example with inline HTML:

```
import React, { Component } from 'react';
import { WebView } from 'react-native';

class MyInlineWeb extends Component {
  render() {
    return (
      <WebView
        originWhitelist={['*']}
        source={{ html: '<h1>Hello world</h1>' }}
      />
    );
  }
}
```

You can use this component to navigate back and forth in the web view's history and configure various properties for the web content.

On iOS, the `useWebKit` prop can be used to opt into a WKWebView-backed implementation.

Security Warning: Currently, `onMessage` and `postMessage` do not allow specifying an origin. This can lead to cross-site scripting attacks if an unexpected document is loaded within a `WebView` instance. Please refer to the MDN documentation for `Window.postMessage()` for more details on the security implications of this.

Props

- [View props...](#)
- `source`
- `automaticallyAdjustContentInsets`
- `injectJavaScript`
- `injectedJavaScript`
- `mediaPlaybackRequiresUserAction`

- `nativeConfig`
- `onError`
- `onLoad`
- `onLoadEnd`
- `onLoadStart`
- `onMessage`
- `onNavigationStateChange`
- `originWhitelist`
- `renderError`
- `renderLoading`
- `scalesPageToFit`
- `onShouldStartLoadWithRequest`
- `startInLoadingState`
- `style`
- `decelerationRate`
- `domStorageEnabled`
- `javaScriptEnabled`
- `mixedContentMode`
- `thirdPartyCookiesEnabled`
- `userAgent`
- `allowsInlineMediaPlayback`
- `allowFileAccess`
- `bounces`
- `contentInset`
- `dataDetectorTypes`
- `scrollEnabled`
- `geolocationEnabled`
- `allowUniversalAccessFromFileURLs`
- `useWebKit`
- `url`
- `html`

Methods

- `extraNativeComponentConfig`
 - `goForward`
 - `goBack`
 - `reload`
 - `stopLoading`
-

Reference

Props

`source`

Loads static HTML or a URI (with optional headers) in the WebView. Note that static HTML will require setting `originWhitelist` to `["*"]`.

The object passed to `source` can have either of the following shapes:

Load uri

- `uri` (string) - The URI to load in the `WebView`. Can be a local or remote file.
- `method` (string) - The HTTP Method to use. Defaults to GET if not specified. On Android, the only supported methods are GET and POST.
- `headers` (object) - Additional HTTP headers to send with the request. On Android, this can only be used with GET requests.
- `body` (string) - The HTTP body to send with the request. This must be a valid UTF-8 string, and will be sent exactly as specified, with no additional encoding (e.g. URL-escaping or base64) applied. On Android, this can only be used with POST requests.

Static HTML

- `html` (string) - A static HTML page to display in the WebView.
- `baseUrl` (string) - The base URL to be used for any relative links in the HTML.

Type	Required
object	No

automaticallyAdjustContentInsets

Controls whether to adjust the content inset for web views that are placed behind a navigation bar, tab bar, or toolbar. The default value is `true`.

Type	Required
bool	No

injectJavaScript

Function that accepts a string that will be passed to the WebView and executed immediately as JavaScript.

Type	Required
function	No

injectedJavaScript

Set this to provide JavaScript that will be injected into the web page when the view loads.

Type	Required
string	No

mediaPlaybackRequiresUserAction

Boolean that determines whether HTML5 audio and video requires the user to tap them before they start playing. The default value is `true`.

Type	Required
bool	No

nativeConfig

Override the native component used to render the WebView. Enables a custom native WebView which uses the same JavaScript as the original WebView.

The `nativeConfig` prop expects an object with the following keys:

- `component` (any)
- `props` (object)
- `viewManager` (object)

Type	Required
object	No

onError

Function that is invoked when the `WebView` load fails.

Type	Required
function	No

onLoad

Function that is invoked when the `WebView` has finished loading.

Type	Required
function	No

onLoadEnd

Function that is invoked when the `WebView` load succeeds or fails.

Type	Required
function	No

onLoadStart

Function that is invoked when the `WebView` starts loading.

Type	Required
function	No

onMessage

A function that is invoked when the webview calls `window.postMessage`. Setting this property will inject a `postMessage` global into your webview, but will still call pre-existing values of `postMessage`.

`window.postMessage` accepts one argument, `data`, which will be available on the event object, `event.nativeEvent.data`. `data` must be a string.

Type	Required
function	No

onNavigationStateChange

Function that is invoked when the `WebView` loading starts or ends.

Type	Required
function	No

originWhitelist

List of origin strings to allow being navigated to. The strings allow wildcards and get matched against *just* the origin (not the full URL). If the user taps to navigate to a new page but the new page is not in this whitelist, the URL will be handled by the OS. The default whitelisted origins are "http://" and "https://".

Type	Required
array of strings	No

renderError

Function that returns a view to show if there's an error.

Type	Required
function	No

renderLoading

Function that returns a loading indicator. The `startInLoadingState` prop must be set to true in order to use this prop.

Type	Required
function	No

scalesPageToFit

Boolean that controls whether the web content is scaled to fit the view and enables the user to change the scale. The default value is `true`.

On iOS, when `useWebKit=true`, this prop will not work.

Type	Required
bool	No

onShouldStartLoadWithRequest

Function that allows custom handling of any web view requests. Return `true` from the function to continue loading the request and `false` to stop loading.

Type	Required	Platform
function	No	iOS

startInLoadingState

Boolean value that forces the `WebView` to show the loading view on the first load. This prop must be set to `true` in order for the `renderLoading` prop to work.

Type	Required
bool	No

decelerationRate

A floating-point number that determines how quickly the scroll view decelerates after the user lifts their finger. You may also use the string shortcuts `"normal"` and `"fast"` which match the underlying iOS settings for `UIScrollViewDecelerationRateNormal` and `UIScrollViewDecelerationRateFast` respectively:

- normal: 0.998
- fast: 0.99 (the default for iOS web view)

Type	Required	Platform

number	No	iOS
--------	----	-----

domStorageEnabled

Boolean value to control whether DOM Storage is enabled. Used only in Android.

Type	Required	Platform
bool	No	Android

javaScriptEnabled

Boolean value to enable JavaScript in the `WebView`. Used on Android only as JavaScript is enabled by default on iOS. The default value is `true`.

Type	Required	Platform
bool	No	Android

mixedContentMode

Specifies the mixed content mode. i.e WebView will allow a secure origin to load content from any other origin.

Possible values for `mixedContentMode` are:

- `never` (default) - WebView will not allow a secure origin to load content from an insecure origin.
- `always` - WebView will allow a secure origin to load content from any other origin, even if that origin is insecure.
- `compatibility` - WebView will attempt to be compatible with the approach of a modern web browser with regard to mixed content.

Type	Required	Platform
string	No	Android

thirdPartyCookiesEnabled

Boolean value to enable third party cookies in the `WebView`. Used on Android Lollipop and above only as third party cookies are enabled by default on Android Kitkat and below and on iOS. The default value is `true`.

Type	Required	Platform
bool	No	Android

userAgent

Sets the user-agent for the `WebView`.

Type	Required	Platform
string	No	Android

allowsInlineMediaPlayback

Boolean that determines whether HTML5 videos play inline or use the native full-screen controller. The default value is `false`.

NOTE

In order for video to play inline, not only does this property need to be set to `true`, but the video element in the HTML document must also include the `webkit-playsinline` attribute.

Type	Required	Platform
bool	No	iOS

bounces

Boolean value that determines whether the web view bounces when it reaches the edge of the content. The default value is `true`.

Type	Required	Platform
bool	No	iOS

contentInset

The amount by which the web view content is inset from the edges of the scroll view. Defaults to `{top: 0, left: 0, bottom: 0, right: 0}`.

Type	Required	Platform
<code>object: {top: number, left: number, bottom: number, right: number}</code>	No	iOS

dataDetectorTypes

Determines the types of data converted to clickable URLs in the web view's content. By default only phone numbers are detected.

You can provide one type or an array of many types.

Possible values for `dataDetectorTypes` are:

- `phoneNumber`
- `link`

- address
- calendarEvent
- none
- all

With the [new WebKit](#) implementation, we have three new values:

- trackingNumber
- flightNumber
- lookupSuggestion

Type	Required	Platform
string, or array	No	iOS

scrollEnabled

Boolean value that determines whether scrolling is enabled in the `WebView`. The default value is `true`.

Type	Required	Platform
bool	No	iOS

geolocationEnabled

Set whether Geolocation is enabled in the `WebView`. The default value is `false`. Used only in Android.

Type	Required	Platform
bool	No	Android

allowUniversalAccessFromFileURLs

Boolean that sets whether JavaScript running in the context of a file scheme URL should be allowed to access content from any origin. Including accessing content from other file scheme URLs. The default value is `false`.

Type	Required	Platform
bool	No	Android

allowFileAccess

Boolean that sets whether the `WebView` has access to the file system. The default value is `false`.

Type	Required	Platform
bool	No	Android

useWebKit

If true, use WKWebView instead of UIWebView.

Type	Required	Platform
boolean	No	iOS

url

Deprecated. Use the `source` prop instead.

Type	Required
string	No

html

Deprecated. Use the `source` prop instead.

Type	Required
string	No

Methods

extraNativeComponentConfig()

```
static extraNativeComponentConfig()
```

goForward()

```
goForward();
```

Go forward one page in the web view's history.

goBack()

```
goBack();
```

Go back one page in the web view's history.

reload()

```
reload();
```

Reloads the current page.

stopLoading()

```
stopLoading();
```

Stop loading the current page.

AccessibilityInfo

Sometimes it's useful to know whether or not the device has a screen reader that is currently active. The `AccessibilityInfo` API is designed for this purpose. You can use it to query the current state of the screen reader as well as to register to be notified when the state of the screen reader changes.

Here's a small example illustrating how to use `AccessibilityInfo`:

```
class ScreenReaderStatusExample extends React.Component {
  state = {
    screenReaderEnabled: false,
  };

  componentDidMount() {
    AccessibilityInfo.addEventListener(
      'change',
      this._handleScreenReaderToggled,
    );
    AccessibilityInfo.fetch().then((isEnabled) => {
      this.setState({
        screenReaderEnabled: isEnabled,
      });
    });
  }

  componentWillUnmount() {
    AccessibilityInfo.removeEventListener(
      'change',
      this._handleScreenReaderToggled,
    );
  }

  _handleScreenReaderToggled = (isEnabled) => {
    this.setState({
      screenReaderEnabled: isEnabled,
    });
  };

  render() {
    return (
      <View>
        <Text>
          The screen reader is{' '}
          {this.state.screenReaderEnabled ? 'enabled' : 'disabled'}.
        </Text>
      </View>
    );
  }
}
```

Methods

- `fetch`
- `addEventListener`
- `setAccessibilityFocus`
- `announceForAccessibility`
- `removeEventListener`

Reference

Methods

fetch()

```
static fetch()
```

Query whether a screen reader is currently enabled. Returns a promise which resolves to a boolean. The result is `true` when a screen reader is enabled and `false` otherwise.

addEventListener()

```
static addEventListener(eventName, handler)
```

Add an event handler. Supported events:

- `change` : Fires when the state of the screen reader changes. The argument to the event handler is a boolean. The boolean is `true` when a screen reader is enabled and `false` otherwise.
- `announcementFinished` : iOS-only event. Fires when the screen reader has finished making an announcement. The argument to the event handler is a dictionary with these keys:
 - `announcement` : The string announced by the screen reader.
 - `success` : A boolean indicating whether the announcement was successfully made.

setAccessibilityFocus()

```
static setAccessibilityFocus(reactTag)
```

Set accessibility focus to a React component. On Android, this is equivalent to

```
UIManager.sendAccessibilityEvent(reactTag, UIManager.AccessibilityEventTypes.typeViewFocused); .
```

announceForAccessibility()

```
static announceForAccessibility(announcement)
```

iOS-Only. Post a string to be announced by the screen reader.

removeEventListener()

```
static removeEventListener(eventName, handler)
```

Remove an event handler.

ActionSheetIOS

Methods

- `showActionSheetWithOptions`
 - `showShareActionSheetWithOptions`
-

Reference

Methods

`showActionSheetWithOptions()`

```
static showActionSheetWithOptions(options, callback)
```

Display an iOS action sheet. The `options` object must contain one or more of:

- `options` (array of strings) - a list of button titles (required)
- `cancelButtonIndex` (int) - index of cancel button in `options`
- `destructiveButtonIndex` (int) - index of destructive button in `options`
- `title` (string) - a title to show above the action sheet
- `message` (string) - a message to show below the title
- `tintColor` (string) - the `color` used for non-destructive button titles

The 'callback' function takes one parameter, the zero-based index of the selected item.

Minimal example:

```
ActionSheetIOS.showActionSheetWithOptions({  
  options: ['Cancel', 'Remove'],  
  destructiveButtonIndex: 1,  
  cancelButtonIndex: 0,  
},  
(buttonIndex) => {  
  if (buttonIndex === 1) { /* destructive action */ }  
});
```

`showShareActionSheetWithOptions()`

```
static showShareActionSheetWithOptions(options, failureCallback, successCallback)
```

Display the iOS share sheet. The `options` object should contain one or both of `message` and `url` and can additionally have a `subject` or `excludedActivityTypes`:

- `url` (string) - a URL to share
- `message` (string) - a message to share
- `subject` (string) - a subject for the message
- `excludedActivityTypes` (array) - the activities to exclude from the ActionSheet

NOTE: if `url` points to a local file, or is a base64-encoded uri, the file it points to will be loaded and shared directly. In this way, you can share images, videos, PDF files, etc.

The 'failureCallback' function takes one parameter, an error object. The only property defined on this object is an optional `stack` property of type `string`.

The 'successCallback' function takes two parameters:

- a boolean value signifying success or failure
- a string that, in the case of success, indicates the method of sharing

Alert

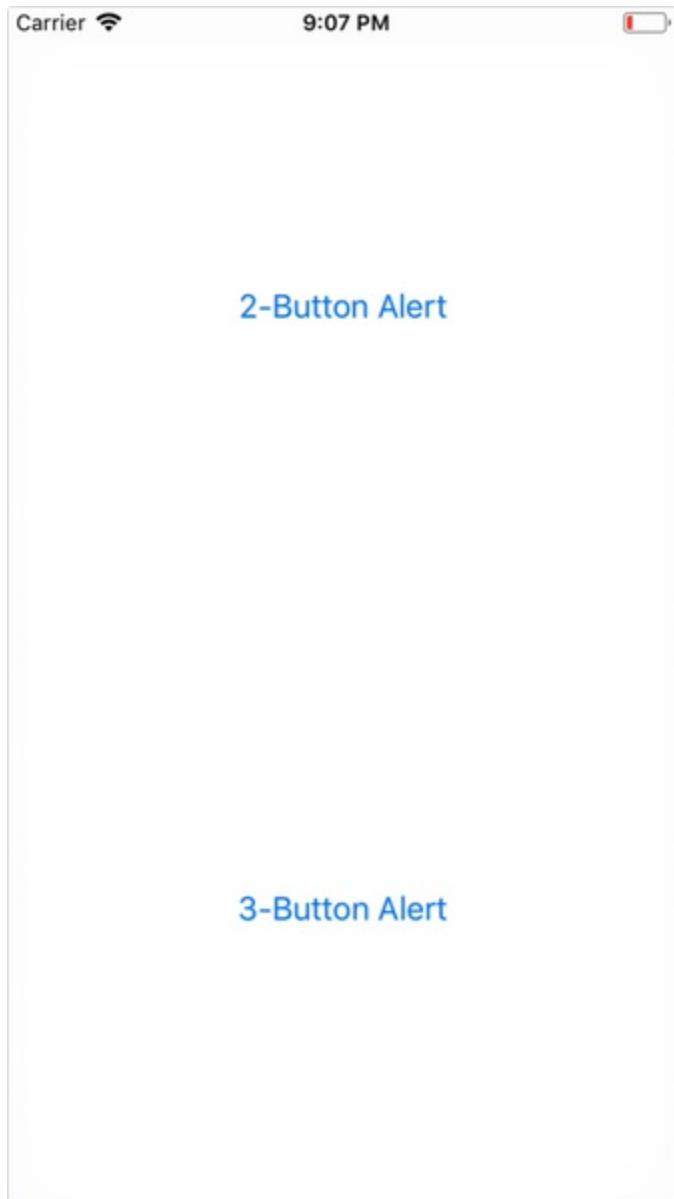
Launches an alert dialog with the specified title and message.

Optionally provide a list of buttons. Tapping any button will fire the respective `onPress` callback and dismiss the alert. By default, the only button will be an 'OK' button.

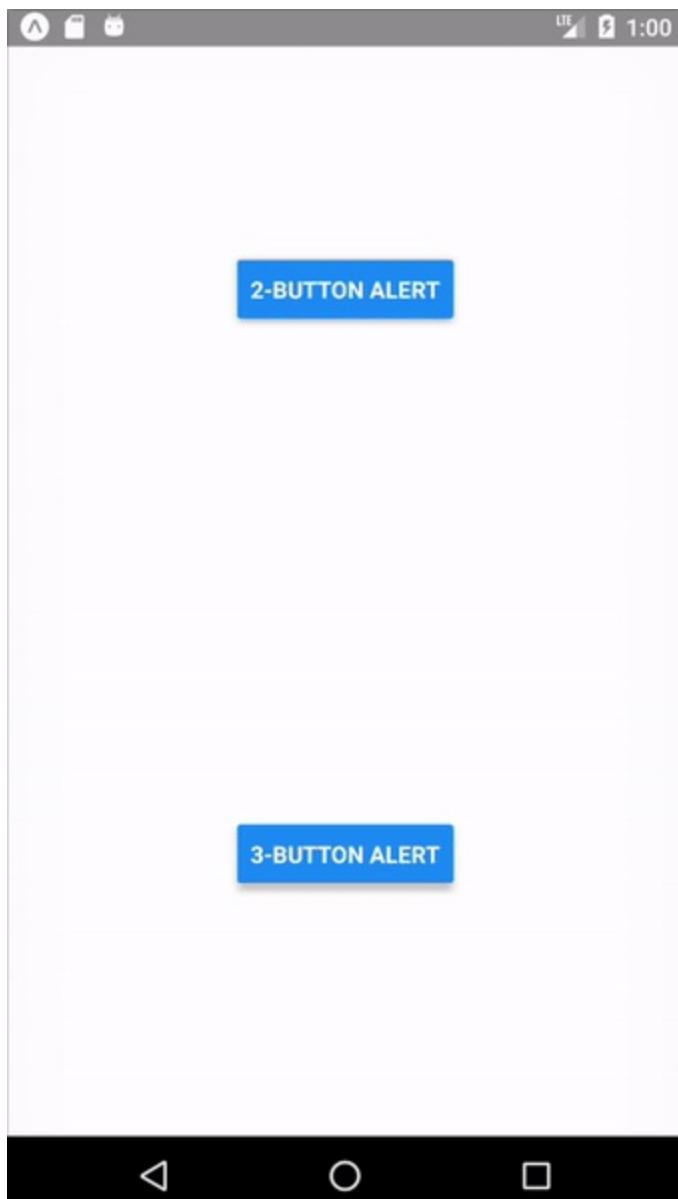
This is an API that works both on iOS and Android and can show static alerts. To show an alert that prompts the user to enter some information, see `AlertIOS`; entering text in an alert is common on iOS only.

Example

iOS Alert Example



Android Alert Example



iOS

On iOS you can specify any number of buttons. Each button can optionally specify a style, which is one of 'default', 'cancel' or 'destructive'.

Android

On Android at most three buttons can be specified. Android has a concept of a neutral, negative and a positive button:

- If you specify one button, it will be the 'positive' one (such as 'OK')
- Two buttons mean 'negative', 'positive' (such as 'Cancel', 'OK')
- Three buttons mean 'neutral', 'negative', 'positive' (such as 'Later', 'Cancel', 'OK')

By default alerts on Android can be dismissed by tapping outside of the alert box. This event can be handled by providing an optional `options` parameter, with an `onDismiss` callback property `{ onDismiss: () => {} }`.

Alternatively, the dismissing behavior can be disabled altogether by providing an optional `options` parameter with the `cancelable` property set to `false` i.e. `{ cancelable: false }`

Example usage:

```
// Works on both iOS and Android
Alert.alert(
  'Alert Title',
  'My Alert Msg',
  [
    {text: 'Ask me later', onPress: () => console.log('Ask me later pressed')},
    {text: 'Cancel', onPress: () => console.log('Cancel Pressed'), style: 'cancel'},
    {text: 'OK', onPress: () => console.log('OK Pressed')},
  ],
  {cancelable: false}
)
```

Methods

- `alert`

Reference

Methods

`alert()`

```
static alert(title, message?, buttons?, options?, type?)
```

AlertIOS

`AlertIOS` provides functionality to create an iOS alert dialog with a message or create a prompt for user input.

Creating an iOS alert:

```
AlertIOS.alert(  
  'Sync Complete',  
  'All your data are belong to us.'  
);
```

Creating an iOS prompt:

```
AlertIOS.prompt(  
  'Enter a value',  
  null,  
  text => console.log("You entered "+text)  
);
```

We recommend using the `Alert.alert` method for cross-platform support if you don't need to create iOS-only prompts.

Methods

- `alert`
- `prompt`

Type Definitions

- `AlertType`
- `AlertButtonStyle`
- `ButtonsArray`

Reference

Methods

`alert()`

```
static alert(title: string, [message]: string, [callbackOrButtons]: ?(() => void), ButtonsArray, [type]: AlertType):  
[object Object]
```

Create and display a popup alert.

Parameters:

Name	Type	Required	Description
title	string	Yes	The dialog's title. Passing null or "" will hide the title.
message	string	No	An optional message that appears below the dialog's title.
callbackOrButtons	?(() => void), ButtonsArray	No	This optional argument should be either a single-argument function or an array of buttons. If passed a function, it will be called when the user taps 'OK'. If passed an array of button configurations, each button should include a <code>text</code> key, as well as optional <code>onPress</code> and <code>style</code> keys. <code>style</code> should be one of 'default', 'cancel' or 'destructive'.
type	AlertType	No	Deprecated, do not use.

Example with custom buttons:

```
AlertIOS.alert(
  'Update available',
  'Keep your app up to date to enjoy the latest features',
  [
    {
      text: 'Cancel',
      onPress: () => console.log('Cancel Pressed'),
      style: 'cancel',
    },
    {
      text: 'Install',
      onPress: () => console.log('Install Pressed'),
    },
  ],
);
```

prompt()

```
static prompt(title: string, [message]: string, [callbackOrButtons]: ?(({text: string} => void), ButtonsArray, [type]: AlertType, [defaultValue]: string, [keyboardType]: string): [object Object]
```

Create and display a prompt to enter some text.

Parameters:

Name	Type	Required	Description
title	string	Yes	The dialog's title.
message	string	No	An optional message that appears above the text input.
			This optional argument should be either a single-argument function or an array of buttons. If passed a function, it will be called

callbackOrButtons	?((text: string) => void), ButtonsArray	No	with the prompt's value when the user taps 'OK'. If passed an array of button configurations, each button should include a <code>text</code> key, as well as optional <code>onPress</code> and <code>style</code> keys (see example). <code>style</code> should be one of 'default', 'cancel' or 'destructive'.
type	AlertType	No	This configures the text input. One of 'plain-text', 'secure-text' or 'login-password'.
defaultValue	string	No	The default text in text input.
keyboardType	string	No	The keyboard type of first text field(if exists). One of 'default', 'email-address', 'numeric', 'phone-pad', 'ascii-capable', 'numbers-and-punctuation', 'url', 'number-pad', 'name-phone-pad', 'decimal-pad', 'twitter' or 'web-search'.

Example with custom buttons:

```
AlertIOS.prompt(
  'Enter password',
  'Enter your password to claim your $1.5B in lottery winnings',
  [
    {
      text: 'Cancel',
      onPress: () => console.log('Cancel Pressed'),
      style: 'cancel',
    },
    {
      text: 'OK',
      onPress: (password) => console.log('OK Pressed, password: ' + password),
    },
  ],
  'secure-text',
);
```

Example with the default button and a custom callback:

```
AlertIOS.prompt(
  'Update username',
  null,
  (text) => console.log('Your username is ' + text),
  null,
  'default',
);
```

Type Definitions

AlertType

An Alert button type

Type

\$Enum

Constants:

Value	Description
default	Default alert with no inputs
plain-text	Plain text input alert
secure-text	Secure text input alert
login-password	Login and password alert

AlertButtonStyle

An Alert button style

Type
\$Enum

Constants:

Value	Description
default	Default button style
cancel	Cancel button style
destructive	Destructive button style

ButtonsArray

Array or buttons

Type
Array

Properties:

Name	Type	Description
[text]	string	Button label
[onPress]	function	Callback function when button pressed
[style]	AlertButtonStyle	Button style

Constants:

Value	Description
text	Button label
onPress	Callback function when button pressed

style

Button style

Animated

The `Animated` library is designed to make animations fluid, powerful, and easy to build and maintain. `Animated` focuses on declarative relationships between inputs and outputs, with configurable transforms in between, and simple `start / stop` methods to control time-based animation execution.

The simplest workflow for creating an animation is to create an `Animated.Value`, hook it up to one or more style attributes of an animated component, and then drive updates via animations using `Animated.timing()`:

```
Animated.timing(
  // Animate value over time
  this.state.fadeAnim, // The value to drive
  {
    toValue: 1, // Animate to final value of 1
  },
).start(); // Start the animation
```

Refer to the [Animations](#) guide to see additional examples of `Animated` in action.

Overview

There are two value types you can use with `Animated`:

- `Animated.Value()` for single values
- `Animated.ValueXY()` for vectors

`Animated.Value` can bind to style properties or other props, and can be interpolated as well. A single `Animated.Value` can drive any number of properties.

Configuring animations

`Animated` provides three types of animation types. Each animation type provides a particular animation curve that controls how your values animate from their initial value to the final value:

- `Animated.decay()` starts with an initial velocity and gradually slows to a stop.
- `Animated.spring()` provides a simple spring physics model.
- `Animated.timing()` animates a value over time using [easing functions](#).

In most cases, you will be using `timing()`. By default, it uses a symmetric `easeInOut` curve that conveys the gradual acceleration of an object to full speed and concludes by gradually decelerating to a stop.

Working with animations

Animations are started by calling `start()` on your animation. `start()` takes a completion callback that will be called when the animation is done. If the animation finished running normally, the completion callback will be invoked with `{finished: true}`. If the animation is done because `stop()` was called on it before it could finish (e.g. because it was interrupted by a gesture or another animation), then it will receive `{finished: false}`.

Using the native driver

By using the native driver, we send everything about the animation to native before starting the animation, allowing native code to perform the animation on the UI thread without having to go through the bridge on every frame. Once the animation has started, the JS thread can be blocked without affecting the animation.

You can use the native driver by specifying `useNativeDriver: true` in your animation configuration. See the [Animations](#) guide to learn more.

Animatable components

Only animatable components can be animated. These special components do the magic of binding the animated values to the properties, and do targeted native updates to avoid the cost of the react render and reconciliation process on every frame. They also handle cleanup on unmount so they are safe by default.

- `createAnimatedComponent()` can be used to make a component animatable.

`Animated` exports the following animatable components using the above wrapper:

- `Animated.Image`
- `AnimatedScrollView`
- `Animated.Text`
- `Animated.View`

Composing animations

Animations can also be combined in complex ways using composition functions:

- `Animated.delay()` starts an animation after a given delay.
- `Animated.parallel()` starts a number of animations at the same time.
- `Animated.sequence()` starts the animations in order, waiting for each to complete before starting the next.
- `Animated.stagger()` starts animations in order and in parallel, but with successive delays.

Animations can also be chained together simply by setting the `toValue` of one animation to be another `Animated.Value`. See [Tracking dynamic values](#) in the Animations guide.

By default, if one animation is stopped or interrupted, then all other animations in the group are also stopped.

Combining animated values

You can combine two animated values via addition, subtraction, multiplication, division, or modulo to make a new animated value:

- `Animated.add()`
- `Animated.subtract()`
- `Animated.divide()`
- `Animated.modulo()`
- `Animated.multiply()`

Interpolation

The `interpolate()` function allows input ranges to map to different output ranges. By default, it will extrapolate the curve beyond the ranges given, but you can also have it clamp the output value. It uses lineal interpolation by default but also supports easing functions.

- `interpolate()`

Read more about interpolation in the [Animation](#) guide.

Handling gestures and other events

Gestures, like panning or scrolling, and other events can map directly to animated values using `Animated.event()`. This is done with a structured map syntax so that values can be extracted from complex event objects. The first level is an array to allow mapping across multiple args, and that array contains nested objects.

- `Animated.event()`

For example, when working with horizontal scrolling gestures, you would do the following in order to map

```
event.nativeEvent.contentOffset.x to scrollX (an Animated.Value):
```

```
onScroll={Animated.event(
  // scrollX = e.nativeEvent.contentOffset.x
  [{ nativeEvent: {
    contentOffset: {
      x: scrollX
    }
  }]
)}
```

Methods

- `decay`
- `timing`
- `spring`
- `add`
- `subtract`
- `divide`
- `multiply`
- `modulo`
- `diffClamp`
- `delay`
- `sequence`
- `parallel`
- `stagger`
- `loop`
- `event`
- `forkEvent`
- `unforkEvent`

Properties

- `Value`
- `ValueXY`
- `Interpolation`
- `Node`
- `createAnimatedComponent`
- `attachNativeEvent`

Reference

Methods

When the given value is a ValueXY instead of a Value, each config option may be a vector of the form `{x: ..., y: ...}` instead of a scalar.

decay()

```
static decay(value, config)
```

Animates a value from an initial velocity to zero based on a decay coefficient.

Config is an object that may have the following options:

- `velocity` : Initial velocity. Required.
- `deceleration` : Rate of decay. Default 0.997.
- `isInteraction` : Whether or not this animation creates an "interaction handle" on the `InteractionManager`. Default true.
- `useNativeDriver` : Uses the native driver when true. Default false.

timing()

```
static timing(value, config)
```

Animates a value along a timed easing curve. The `Easing` module has tons of predefined curves, or you can use your own function.

Config is an object that may have the following options:

- `duration` : Length of animation (milliseconds). Default 500.
- `easing` : Easing function to define curve. Default is `Easing.inOut(Easing.ease)`.
- `delay` : Start the animation after delay (milliseconds). Default 0.
- `isInteraction` : Whether or not this animation creates an "interaction handle" on the `InteractionManager`. Default true.
- `useNativeDriver` : Uses the native driver when true. Default false.

spring()

```
static spring(value, config)
```

Animates a value according to an analytical spring model based on [damped harmonic oscillation](#). Tracks velocity state to create fluid motions as the `toValue` updates, and can be chained together.

Config is an object that may have the following options.

Note that you can only define one of bounciness/speed, tension/friction, or stiffness/damping/mass, but not more than one:

The friction/tension or bounciness/speed options match the spring model in [Facebook Pop](#), [Rebound](#), and [Origami](#).

- `friction` : Controls "bounciness"/overshoot. Default 7.
- `tension` : Controls speed. Default 40.
- `speed` : Controls speed of the animation. Default 12.
- `bounciness` : Controls bounciness. Default 8.

Specifying stiffness/damping/mass as parameters makes `Animated.spring` use an analytical spring model based on the motion equations of a [damped harmonic oscillator](#). This behavior is slightly more precise and faithful to the physics behind spring dynamics, and closely mimics the implementation in iOS's CASpringAnimation primitive.

- `stiffness` : The spring stiffness coefficient. Default 100.
- `damping` : Defines how the spring's motion should be damped due to the forces of friction. Default 10.
- `mass` : The mass of the object attached to the end of the spring. Default 1.

Other configuration options are as follows:

- `velocity` : The initial velocity of the object attached to the spring. Default 0 (object is at rest).
- `overshootClamping` : Boolean indicating whether the spring should be clamped and not bounce. Default false.
- `restDisplacementThreshold` : The threshold of displacement from rest below which the spring should be considered at rest. Default 0.001.
- `restSpeedThreshold` : The speed at which the spring should be considered at rest in pixels per second. Default 0.001.
- `delay` : Start the animation after delay (milliseconds). Default 0.
- `isInteraction` : Whether or not this animation creates an "interaction handle" on the `InteractionManager`. Default true.
- `useNativeDriver` : Uses the native driver when true. Default false.

add()

```
static add(a, b)
```

Creates a new Animated value composed from two Animated values added together.

subtract()

```
static subtract(a, b)
```

Creates a new Animated value composed by subtracting the second Animated value from the first Animated value.

divide()

```
static divide(a, b)
```

Creates a new Animated value composed by dividing the first Animated value by the second Animated value.

multiply()

```
static multiply(a, b)
```

Creates a new Animated value composed from two Animated values multiplied together.

modulo()

```
static modulo(a, modulus)
```

Creates a new Animated value that is the (non-negative) modulo of the provided Animated value

diffClamp()

```
static diffClamp(a, min, max)
```

Create a new Animated value that is limited between 2 values. It uses the difference between the last value so even if the value is far from the bounds it will start changing when the value starts getting closer again. (`value = clamp(value + diff, min, max)`).

This is useful with scroll events, for example, to show the navbar when scrolling up and to hide it when scrolling down.

delay()

```
static delay(time)
```

Starts an animation after the given delay.

sequence()

```
static sequence(animations)
```

Starts an array of animations in order, waiting for each to complete before starting the next. If the current running animation is stopped, no following animations will be started.

parallel()

```
static parallel(animations, config?)
```

Starts an array of animations all at the same time. By default, if one of the animations is stopped, they will all be stopped. You can override this with the `stopTogether` flag.

stagger()

```
static stagger(time, animations)
```

Array of animations may run in parallel (overlap), but are started in sequence with successive delays. Nice for doing trailing effects.

loop()

```
static loop(animation)
```

Loops a given animation continuously, so that each time it reaches the end, it resets and begins again from the start. Can specify number of times to loop using the key `iterations` in the config. Will loop without blocking the UI thread if the child animation is set to `useNativeDriver: true`. In addition, loops can prevent `VirtualizedList`-based components from rendering more rows while the animation is running. You can pass `isInteraction: false` in the child animation config to fix this.

event()

```
static event(argMapping, config?)
```

Takes an array of mappings and extracts values from each arg accordingly, then calls `setValue` on the mapped outputs. e.g.

```
onScroll={Animated.event(
  [{nativeEvent: {contentOffset: {x: this._scrollX}}}],
  {listener: (event) => console.log(event)}, // Optional async listener
)}
...
onPanResponderMove: Animated.event([
  null, // raw event arg ignored
  {dx: this._panX}], // gestureState arg
{listener: (event, gestureState) => console.log(event, gestureState)}, // Optional async listener
),
```

Config is an object that may have the following options:

- `listener` : Optional async listener.
 - `useNativeDriver` : Uses the native driver when true. Default false.
-

forkEvent()

```
static forkEvent(event, listener)
```

Advanced imperative API for snooping on animated events that are passed in through props. It permits to add a new javascript listener to an existing `AnimatedEvent`. If `animatedEvent` is a simple javascript listener, it will merge the 2 listeners into a single one, and if `animatedEvent` is null/undefined, it will assign the javascript listener directly. Use values directly where possible.

unforkEvent()

```
static unforkEvent(event, listener)
```

Properties

Value

Standard value class for driving animations. Typically initialized with `new Animated.Value(0);`

ValueXY

2D value class for driving 2D animations, such as pan gestures.

Interpolation

Exported to use the Interpolation type in flow.

Node

Exported for ease of type checking. All animated values derive from this class.

createAnimatedComponent

Make any React component Animatable. Used to create `Animated.View`, etc.

attachNativeEvent

Imperative API to attach an animated value to an event on a view. Prefer using `Animated.event` with `useNativeDriver: true` if possible.

AppState

`AppState` can tell you if the app is in the foreground or background, and notify you when the state changes.

`AppState` is frequently used to determine the intent and proper behavior when handling push notifications.

App States

- `active` - The app is running in the foreground
- `background` - The app is running in the background. The user is either:
 - in another app
 - on the home screen
 - [Android] on another `Activity` (even if it was launched by your app)
- `inactive` - This is a state that occurs when transitioning between foreground & background, and during periods of inactivity such as entering the Multitasking view or in the event of an incoming call

For more information, see [Apple's documentation](#)

Basic Usage

To see the current state, you can check `AppState.currentState`, which will be kept up-to-date. However, `currentState` will be null at launch while `AppState` retrieves it over the bridge.

```
import React, {Component} from 'react'
import {AppState, Text} from 'react-native'

class AppStateExample extends Component {

  state = {
    appState: AppState.currentState
  }

  componentDidMount() {
    AppState.addEventListener('change', this._handleAppStateChange);
  }

  componentWillUnmount() {
    AppState.removeEventListener('change', this._handleAppStateChange);
  }

  _handleAppStateChange = (nextAppState) => {
    if (this.state.appState.match(/inactive|background/) && nextAppState === 'active') {
      console.log('App has come to the foreground!')
    }
    this.setState({appState: nextAppState});
  }

  render() {
    return (
      <Text>Current state is: {this.state.appState}</Text>
    );
  }
}
```

This example will only ever appear to say "Current state is: active" because the app is only visible to the user when in the `active` state, and the null state will happen only momentarily.

Methods

- `addEventListener`
- `removeEventListener`

Properties

- `currentState`
-

Reference

Methods

`addEventListener()`

```
addEventListener(type, handler);
```

Add a handler to AppState changes by listening to the `change` event type and providing the handler

TODO: now that AppState is a subclass of NativeEventEmitter, we could deprecate `addEventListener` and `removeEventListener` and just use `addListener` and `listener.remove()` directly. That will be a breaking change though, as both the method and event names are different (addListener events are currently required to be globally unique).

`removeEventListener()`

```
removeEventListener(type, handler);
```

Remove a handler by passing the `change` event type and the handler

Properties

`currentState`

```
AppState.currentState;
```


AsyncStorage

`AsyncStorage` is a simple, unencrypted, asynchronous, persistent, key-value storage system that is global to the app. It should be used instead of `LocalStorage`.

It is recommended that you use an abstraction on top of `AsyncStorage` instead of `AsyncStorage` directly for anything more than light usage since it operates globally.

On iOS, `AsyncStorage` is backed by native code that stores small values in a serialized dictionary and larger values in separate files. On Android, `AsyncStorage` will use either `RocksDB` or SQLite based on what is available.

The `AsyncStorage` JavaScript code is a simple facade that provides a clear JavaScript API, real `Error` objects, and simple non-multi functions. Each method in the API returns a `Promise` object.

Importing the `AsyncStorage` library:

```
import { AsyncStorage } from "react-native"
```

Persisting data:

```
_storeData = async () => {
  try {
    await AsyncStorage.setItem('@MySuperStore:key', 'I like to save it.');
  } catch (error) {
    // Error saving data
  }
}
```

Fetching data:

```
_retrieveData = async () => {
  try {
    const value = await AsyncStorage.getItem('TASKS');
    if (value !== null) {
      // We have data!!
      console.log(value);
    }
  } catch (error) {
    // Error retrieving data
  }
}
```

Methods

- `getItem`
- `setItem`
- `removeItem`
- `mergeItem`
- `clear`
- `getAllKeys`

- `flushGetRequests`
 - `multiGet`
 - `multiSet`
 - `multiRemove`
 - `multiMerge`
-

Reference

Methods

`getItem()`

```
static getItem(key: string, [callback]: ?(error: ?Error, result: ?string) => void)
```

Fetches an item for a `key` and invokes a callback upon completion. Returns a `Promise` object.

Parameters:

Name	Type	Required	Description
key	string	Yes	Key of the item to fetch.
callback	?(error: ?Error, result: ?string) => void	No	Function that will be called with a result if found or any error.

`setItem()`

```
static setItem(key: string, value: string, [callback]: ?(error: ?Error) => void)
```

Sets the value for a `key` and invokes a callback upon completion. Returns a `Promise` object.

Parameters:

Name	Type	Required	Description
key	string	Yes	Key of the item to set.
value	string	Yes	Value to set for the <code>key</code> .
callback	?(error: ?Error) => void	No	Function that will be called with any error.

`removeItem()`

```
static removeItem(key: string, [callback]: ?(error: ?Error) => void)
```

Removes an item for a `key` and invokes a callback upon completion. Returns a `Promise` object.

Parameters:

Name	Type	Required	Description
key	string	Yes	Key of the item to remove.
callback	?(error: ?Error) => void	No	Function that will be called with any error.

mergeItem()

```
static mergeItem(key: string, value: string, [callback]: ?(error: ?Error) => void)
```

Merges an existing `key` value with an input value, assuming both values are stringified JSON. Returns a `Promise` object.

NOTE: This is not supported by all native implementations.

Parameters:

Name	Type	Required	Description
key	string	Yes	Key of the item to modify.
value	string	Yes	New value to merge for the <code>key</code> .
callback	?(error: ?Error) => void	No	Function that will be called with any error.

Example:

```
let UID123_object = {
  name: 'Chris',
  age: 30,
  traits: {hair: 'brown', eyes: 'brown'},
};

// You only need to define what will be added or updated
let UID123_delta = {
  age: 31,
  traits: {eyes: 'blue', shoe_size: 10},
};

AsyncStorage.setItem('UID123', JSON.stringify(UID123_object), () => {
  AsyncStorage.mergeItem('UID123', JSON.stringify(UID123_delta), () => {
    AsyncStorage.getItem('UID123', (err, result) => {
      console.log(result);
    });
  });
});

// Console log result:
// => {'name':'Chris','age':31,'traits':
//     {'shoe_size':10,'hair':'brown','eyes':'blue'}}
```

clear()

```
static clear([callback]: ?(error: ?Error) => void)
```

Erases *all* `AsyncStorage` for all clients, libraries, etc. You probably don't want to call this; use `removeItem` or `multiRemove` to clear only your app's keys. Returns a `Promise` object.

Parameters:

Name	Type	Required	Description
callback	?(error: ?Error) => void	No	Function that will be called with any error.

getAllKeys()

```
static getAllKeys([callback]: ?(error: ?Error, keys: ?Array<string>) => void)
```

Gets *all* keys known to your app; for all callers, libraries, etc. Returns a `Promise` object.

Parameters:

Name	Type	Required	Description
callback	?(error: ?Error, keys: ?Array<string>) => void	No	Function that will be called the keys found and any error.

flushGetRequests()

```
static flushGetRequests(): [object Object]
```

Flushes any pending requests using a single batch call to get the data.

multiGet()

```
static multiGet(keys: Array<string>, [callback]: ?(errors: ?Array<Error>, result: ?Array<Array<string>>) => void)
```

This allows you to batch the fetching of items given an array of `key` inputs. Your callback will be invoked with an array of corresponding key-value pairs found:

```
multiGet(['k1', 'k2'], cb) -> cb([[['k1', 'val1'], ['k2', 'val2']]])
```

The method returns a `Promise` object.

Parameters:

Name	Type	Required	Description
keys	Array<string>	Yes	Array of key for the items to get.
callback	?(errors: ?Array<Error>, result: ?Array<any>) => void	No	Function that will be called with a key-value array of the results, plus an array of any key-specific errors found.

Example:

```
AsyncStorage.getAllKeys((err, keys) => {
  AsyncStorage.multiGet(keys, (err, stores) => {
    stores.map((result, i, store) => {
      // get at each store's key/value so you can work with it
      let key = store[i][0];
      let value = store[i][1];
    });
  });
});
```

multiSet()

```
static multiSet(keyValuePairs: Array<Array<string>>, [callback]: ?(errors: ?Array<Error>) => void)
```

Use this as a batch operation for storing multiple key-value pairs. When the operation completes you'll get a single callback with any errors:

```
multiSet([[ 'k1', 'val1' ], [ 'k2', 'val2' ]], cb);
```

The method returns a `Promise` object.

Parameters:

Name	Type	Required	Description
keyValuePairs	Array<any>	Yes	Array of key-value array for the items to set.
callback	?(errors: ?Array<Error>) => void	No	Function that will be called with an array of any key-specific errors found.

multiRemove()

```
static multiRemove(keys: Array<string>, [callback]: ?(errors: ?Array<Error>) => void)
```

Call this to batch the deletion of all keys in the `keys` array. Returns a `Promise` object.

Parameters:

Name	Type	Required	Description
keys	Array\`	Yes	Array of key for the items to delete.
callback	?(errors: ?Array\` => void)	No	Function that will be called an array of any key-specific errors found.

Example:

```
let keys = ['k1', 'k2'];
AsyncStorage.multiRemove(keys, (err) => {
  // keys k1 & k2 removed, if they existed
  // do most stuff after removal (if you want)
});
```

multiMerge()

```
static multiMerge(keyValuePairs: Array<Array<string>>, [callback]: ?(errors: ?Array<Error>) => void)
```

Batch operation to merge in existing and new values for a given set of keys. This assumes that the values are stringified JSON. Returns a `Promise` object.

NOTE: This is not supported by all native implementations.

Parameters:

Name	Type	Required	Description
keyValuePairs	Array\>	Yes	Array of key-value array for the items to merge.
callback	?(errors: ?Array\` => void)	No	Function that will be called with an array of any key-specific errors found.

Example:

```
// first user, initial values
let UID234_object = {
  name: 'Chris',
  age: 30,
  traits: {hair: 'brown', eyes: 'brown'},
};

// first user, delta values
let UID234_delta = {
  age: 31,
  traits: {eyes: 'blue', shoe_size: 10},
};

// second user, initial values
let UID345_object = {
  name: 'Marge',
  age: 25,
  traits: {hair: 'blonde', eyes: 'blue'},
};
```

```

// second user, delta values
let UID345_delta = {
  age: 26,
  traits: {eyes: 'green', shoe_size: 6},
};

let multi_set_pairs = [
  ['UID234', JSON.stringify(UID234_object)],
  ['UID345', JSON.stringify(UID345_object)],
];
let multi_merge_pairs = [
  ['UID234', JSON.stringify(UID234_delta)],
  ['UID345', JSON.stringify(UID345_delta)],
];
;

AsyncStorage.multiSet(multi_set_pairs, (err) => {
  AsyncStorage.multiMerge(multi_merge_pairs, (err) => {
    AsyncStorage.multiGet(['UID234', 'UID345'], (err, stores) => {
      stores.map((result, i, store) => {
        let key = store[i][0];
        let val = store[i][1];
        console.log(key, val);
      });
    });
  });
});

// Console log results:
// => UID234 {"name": "Chris", "age": 31, "traits": {"shoe_size": 10, "hair": "brown", "eyes": "blue"}}
// => UID345 {"name": "Marge", "age": 26, "traits": {"shoe_size": 6, "hair": "blonde", "eyes": "green"}}


```

BackAndroid

Deprecated. Use BackHandler instead.

Methods

- [exitApp](#)
 - [addEventListener](#)
 - [removeEventListener](#)
-

Reference

Methods

exitApp()

```
static exitApp()
```

addEventListener()

```
static addEventListener(eventName, handler)
```

removeEventListener()

```
static removeEventListener(eventName, handler)
```

BackHandler

Detect hardware button presses for back navigation.

Android: Detect hardware back button presses, and programmatically invoke the default back button functionality to exit the app if there are no listeners or if none of the listeners return true.

tvOS: Detect presses of the menu button on the TV remote. (Still to be implemented: programmatically disable menu button handling functionality to exit the app if there are no listeners or if none of the listeners return true.)

iOS: Not applicable.

The event subscriptions are called in reverse order (i.e. last registered subscription first), and if one subscription returns true then subscriptions registered earlier will not be called.

Example:

```
BackHandler.addEventListener('hardwareBackPress', function() {
  // this.onMainScreen and this.goBack are just examples, you need to use your own implementation here
  // Typically you would use the navigator here to go to the last state.

  if (!this.onMainScreen()) {
    this.goBack();
    return true;
  }
  return false;
});
```

Lifecycle example:

```
componentDidMount() {
  BackHandler.addEventListener('hardwareBackPress', this.handleBackPressed);
}

componentWillUnmount() {
  BackHandler.removeEventListener('hardwareBackPress', this.handleBackPressed);
}

handleBackPressed = () => {
  this.goBack(); // works best when the goBack is async
  return true;
}
```

Lifecycle alternative:

```
componentDidMount() {
  this.backHandler = BackHandler.addEventListener('hardwareBackPress', () => {
    this.goBack(); // works best when the goBack is async
    return true;
  });
}

componentWillUnmount() {
  this.backHandler.remove();
```

```
}
```

Methods

- [exitApp](#)
 - [addEventListener](#)
 - [removeEventListener](#)
-

Reference

Methods

exitApp()

```
static exitApp()
```

addEventListener()

```
static addEventListener(eventName, handler)
```

removeEventListener()

```
static removeEventListener(eventName, handler)
```

Clipboard

`Clipboard` gives you an interface for setting and getting content from Clipboard on both iOS and Android

Methods

- `getString`
 - `setString`
-

Reference

Methods

`getString()`

```
static getString()
```

Get content of string type, this method returns a `Promise`, so you can use following code to get clipboard content

```
async _getContent() {
  var content = await Clipboard.getString();
}
```

`setString()`

```
static setString(content)
```

Set content of string type. You can use following code to set clipboard content

```
_setContent() {
  Clipboard.setString('hello world');
}
```

@param the content to be stored in the clipboard.

DatePickerAndroid

Opens the standard Android date picker dialog.

Example

```
try {
  const {action, year, month, day} = await DatePickerAndroid.open({
    // Use `new Date()` for current date.
    // May 25 2020. Month 0 is January.
    date: new Date(2020, 4, 25)
  });
  if (action !== DatePickerAndroid.dismissedAction) {
    // Selected year, month (0-11), day
  }
} catch ({code, message}) {
  console.warn('Cannot open date picker', message);
}
```

Methods

- `open`
 - `dateSetAction`
 - `dismissedAction`
-

Reference

Methods

`open()`

```
static open(options)
```

Opens the standard Android date picker dialog.

The available keys for the `options` object are:

- `date` (`Date` object or timestamp in milliseconds) - date to show by default
- `minDate` (`Date` or timestamp in milliseconds) - minimum date that can be selected
- `maxDate` (`Date` object or timestamp in milliseconds) - maximum date that can be selected
- `mode` (`enum('calendar', 'spinner', 'default')`) - To set the date-picker mode to calendar/spinner/default
 - 'calendar': Show a date picker in calendar mode.
 - 'spinner': Show a date picker in spinner mode.
 - 'default': Show a default native date picker(spinner/calendar) based on android versions.

Returns a Promise which will be invoked an object containing `action`, `year`, `month` (0-11), `day` if the user picked a date. If the user dismissed the dialog, the Promise will still be resolved with `action` being `DatePickerAndroid.dismissedAction` and all the other keys being undefined. **Always** check whether the `action` is equal to `DatePickerAndroid.dateSetAction` before reading the values.

Note the native date picker dialog has some UI glitches on Android 4 and lower when using the `minDate` and `maxDate` options.

dateSetAction()

```
static dateSetAction()
```

A date has been selected.

dismissedAction()

```
static dismissedAction()
```

The dialog has been dismissed.

Dimensions

Methods

- `set`
 - `get`
 - `addEventListener`
 - `removeEventListener`
-

Reference

Methods

`set()`

```
static set(dims)
```

This should only be called from native code by sending the `didUpdateDimensions` event.

`@param {object} dims` Simple string-keyed object of dimensions to set

`get()`

```
static get(dim)
```

Initial dimensions are set before `runApplication` is called so they should be available before any other require's are run, but may be updated later.

Note: Although dimensions are available immediately, they may change (e.g due to device rotation) so any rendering logic or styles that depend on these constants should try to call this function on every render, rather than caching the value (for example, using inline styles rather than setting a value in a `StyleSheet`).

Example: `var {height, width} = Dimensions.get('window');`

`@param {string} dim` Name of dimension as defined when calling `set`. `@returns {Object?}` Value for the dimension.

`addEventListener()`

```
static addEventListener(type, handler)
```

Add an event handler. Supported events:

- `change` : Fires when a property within the `Dimensions` object changes. The argument to the event handler is an object with `window` and `screen` properties whose values are the same as the return values of `Dimensions.get('window')` and `Dimensions.get('screen')`, respectively.
-

removeEventListener()

```
static removeEventListener(type, handler)
```

Remove an event handler.

Easing

The `Easing` module implements common easing functions. This module is used by `Animated.timing()` to convey physically believable motion in animations.

You can find a visualization of some common easing functions at <http://easings.net/>

Predefined animations

The `Easing` module provides several predefined animations through the following methods:

- `back` provides a simple animation where the object goes slightly back before moving forward
- `bounce` provides a bouncing animation
- `ease` provides a simple inertial animation
- `elastic` provides a simple spring interaction

Standard functions

Three standard easing functions are provided:

- `linear`
- `quad`
- `cubic`

The `poly` function can be used to implement quartic, quintic, and other higher power functions.

Additional functions

Additional mathematical functions are provided by the following methods:

- `bezier` provides a cubic bezier curve
- `circle` provides a circular function
- `sin` provides a sinusoidal function
- `exp` provides an exponential function

The following helpers are used to modify other easing functions.

- `in` runs an easing function forwards
- `inOut` makes any easing function symmetrical
- `out` runs an easing function backwards

Methods

- `step0`
- `step1`
- `linear`
- `ease`
- `quad`
- `cubic`
- `poly`
- `sin`

- `circle`
 - `exp`
 - `elastic`
 - `back`
 - `bounce`
 - `bezier`
 - `in`
 - `out`
 - `inOut`
-

Reference

Methods

`step0()`

```
static step0(n)
```

A stepping function, returns 1 for any positive value of `n`.

`step1()`

```
static step1(n)
```

A stepping function, returns 1 if `n` is greater than or equal to 1.

`linear()`

```
static linear(t)
```

A linear function, $f(t) = t$. Position correlates to elapsed time one to one.

<http://cubic-bezier.com/#0,0,1,1>

`ease()`

```
static ease(t)
```

A simple inertial interaction, similar to an object slowly accelerating to speed.

<http://cubic-bezier.com/#.42,0,1,1>

quad()

```
static quad(t)
```

A quadratic function, $f(t) = t * t$. Position equals the square of elapsed time.

<http://easings.net/#easeInQuad>

cubic()

```
static cubic(t)
```

A cubic function, $f(t) = t * t * t$. Position equals the cube of elapsed time.

<http://easings.net/#easeInCubic>

poly()

```
static poly(n)
```

A power function. Position is equal to the Nth power of elapsed time.

n = 4: <http://easings.net/#easeInQuart> n = 5: <http://easings.net/#easeInQuint>

sin()

```
static sin(t)
```

A sinusoidal function.

<http://easings.net/#easeInSine>

circle()

```
static circle(t)
```

A circular function.

<http://easings.net/#easeInCirc>

exp()

```
static exp(t)
```

An exponential function.

<http://easings.net/#easeInExpo>

elastic()

```
static elastic(bounciness)
```

A simple elastic interaction, similar to a spring oscillating back and forth.

Default bounciness is 1, which overshoots a little bit once. 0 bounciness doesn't overshoot at all, and bounciness of N > 1 will overshoot about N times.

<http://easings.net/#easeInElastic>

back()

```
static back(s)
```

Use with `Animated.parallel()` to create a simple effect where the object animates back slightly as the animation starts.

bounce()

```
static bounce(t)
```

Provides a simple bouncing effect.

<http://easings.net/#easeInBounce>

bezier()

```
static bezier(x1, y1, x2, y2)
```

Provides a cubic bezier curve, equivalent to CSS Transitions' `transition-timing-function`.

A useful tool to visualize cubic bezier curves can be found at <http://cubic-bezier.com/>

in()

```
static in easing;
```

Runs an easing function forwards.

out()

```
static out(easing)
```

Runs an easing function backwards.

inOut()

```
static inOut(easing)
```

Makes any easing function symmetrical. The easing function will run forwards for half of the duration, then backwards for the rest of the duration.

Image Style Props

Props

- `borderTopRightRadius`
 - `backfaceVisibility`
 - `borderBottomLeftRadius`
 - `borderBottomRightRadius`
 - `borderColor`
 - `borderRadius`
 - `borderTopLeftRadius`
 - `backgroundColor`
 - `borderWidth`
 - `opacity`
 - `overflow`
 - `resizeMode`
 - `tintColor`
 - `overlayColor`
-

Reference

Props

`borderTopRightRadius`

Type	Required
number	No

`backfaceVisibility`

Type	Required
enum('visible', 'hidden')	No

`borderBottomLeftRadius`

Type	Required
number	No

borderBottomRightRadius

Type	Required
number	No

borderColor

Type	Required
color	No

borderRadius

Type	Required
number	No

borderTopLeftRadius

Type	Required
number	No

backgroundColor

Type	Required
color	No

borderWidth

Type	Required
number	No

opacity

Type	Required
number	No

overflow

Type	Required
enum('visible', 'hidden')	No

resizeMode

Type	Required
Object.keys(ImageResizeMode)	No

tintColor

Changes the color of all the non-transparent pixels to the tintColor.

Type	Required
color	No

overlayColor

When the image has rounded corners, specifying an overlayColor will cause the remaining space in the corners to be filled with a solid color. This is useful in cases which are not supported by the Android implementation of rounded corners:

- Certain resize modes, such as 'contain'
- Animated GIFs

A typical way to use this prop is with images displayed on a solid background and setting the `overlayColor` to the same color as the background.

For details of how this works under the hood, see <http://frescolib.org/rounded-corners-and-circles.md>

Type	Required	Platform
string	No	Android

ImageStore

Methods

- `hasImageForTag`
 - `removeImageForTag`
 - `addImageFromBase64`
 - `getBase64ForTag`
-

Reference

Methods

`hasImageForTag()`

```
static hasImageForTag(uri, callback)
```

Check if the ImageStore contains image data for the specified URI. @platform ios

`removeImageForTag()`

```
static removeImageForTag(uri)
```

Delete an image from the ImageStore. Images are stored in memory and must be manually removed when you are finished with them, otherwise they will continue to use up RAM until the app is terminated. It is safe to call `removeImageForTag()` without first calling `hasImageForTag()`, it will simply fail silently. @platform ios

`addImageFromBase64()`

```
static addImageFromBase64(base64ImageData, success, failure)
```

Stores a base64-encoded image in the ImageStore, and returns a URI that can be used to access or display the image later. Images are stored in memory only, and must be manually deleted when you are finished with them by calling `removeImageForTag()`.

Note that it is very inefficient to transfer large quantities of binary data between JS and native code, so you should avoid calling this more than necessary. @platform ios

getBase64ForTag()

```
static getBase64ForTag(uri, success, failure)
```

Retrieves the base64-encoded data for an image in the ImageStore. If the specified URI does not match an image in the store, the failure callback will be called.

Note that it is very inefficient to transfer large quantities of binary data between JS and native code, so you should avoid calling this more than necessary. To display an image in the ImageStore, you can just pass the URI to an `<Image/>` component; there is no need to retrieve the base64 data.

InteractionManager

InteractionManager allows long-running work to be scheduled after any interactions/animations have completed. In particular, this allows JavaScript animations to run smoothly.

Applications can schedule tasks to run after interactions with the following:

```
InteractionManager.runAfterInteractions(() => {
  // ...long-running synchronous task...
});
```

Compare this to other scheduling alternatives:

- `requestAnimationFrame()`: for code that animates a view over time.
- `setImmediate/setTimeout()`: run code later, note this may delay animations.
- `runAfterInteractions()`: run code later, without delaying active animations.

The touch handling system considers one or more active touches to be an 'interaction' and will delay `runAfterInteractions()` callbacks until all touches have ended or been cancelled.

InteractionManager also allows applications to register animations by creating an interaction 'handle' on animation start, and clearing it upon completion:

```
var handle = InteractionManager.createInteractionHandle();
// run animation... (`runAfterInteractions` tasks are queued)
// later, on animation completion:
InteractionManager.clearInteractionHandle(handle);
// queued tasks run if all handles were cleared
```

`runAfterInteractions` takes either a plain callback function, or a `PromiseTask` object with a `gen` method that returns a `Promise`. If a `PromiseTask` is supplied, then it is fully resolved (including asynchronous dependencies that also schedule more tasks via `runAfterInteractions`) before starting on the next task that might have been queued up synchronously earlier.

By default, queued tasks are executed together in a loop in one `setImmediate` batch. If `setDeadline` is called with a positive number, then tasks will only be executed until the deadline (in terms of js event loop run time) approaches, at which point execution will yield via `setTimeout`, allowing events such as touches to start interactions and block queued tasks from executing, making apps more responsive.

Methods

- `runAfterInteractions`
- `createInteractionHandle`
- `clearInteractionHandle`
- `setDeadline`

Properties

- `Events`

- `addListener`
-

Reference

Methods

`runAfterInteractions()`

```
static runAfterInteractions(task)
```

Schedule a function to run after all interactions have completed. Returns a cancellable "promise".

`createInteractionHandle()`

```
static createInteractionHandle()
```

Notify manager that an interaction has started.

`clearInteractionHandle()`

```
static clearInteractionHandle(handle)
```

Notify manager that an interaction has completed.

`setDeadline()`

```
static setDeadline(deadline)
```

A positive number will use setTimeout to schedule any tasks after the eventLoopRunningTime hits the deadline value, otherwise all tasks will be executed in one setImmediate batch (default).

Properties

Keyboard

Keyboard module to control keyboard events.

Usage

The Keyboard module allows you to listen for native events and react to them, as well as make changes to the keyboard, like dismissing it.

```
import React, { Component } from 'react';
import { Keyboard, TextInput } from 'react-native';

class Example extends Component {
  componentDidMount () {
    this.keyboardDidShowListener = Keyboard.addListener('keyboardDidShow', this._keyboardDidShow);
    this.keyboardDidHideListener = Keyboard.addListener('keyboardDidHide', this._keyboardDidHide);
  }

  componentWillUnmount () {
    this.keyboardDidShowListener.remove();
    this.keyboardDidHideListener.remove();
  }

  _keyboardDidShow () {
    alert('Keyboard Shown');
  }

  _keyboardDidHide () {
    alert('Keyboard Hidden');
  }

  render() {
    return (
      <TextInput
        onSubmitEditing={Keyboard.dismiss}
      />
    );
  }
}
```

Methods

- `addListener`
- `removeListener`
- `removeAllListeners`
- `dismiss`

Reference

Methods

addListener()

```
static addListener(eventName, callback)
```

The `addListener` function connects a JavaScript function to an identified native keyboard notification event.

This function then returns the reference to the listener.

@param {string} eventName The `nativeEvent` is the string that identifies the event you're listening for. This can be any of the following:

- `keyboardWillShow`
- `keyboardDidShow`
- `keyboardWillHide`
- `keyboardDidHide`
- `keyboardWillChangeFrame`
- `keyboardDidChangeFrame`

Note that if you set `android:windowSoftInputMode` to `adjustResize` OR `adjustNothing`, only `keyboardDidShow` and `keyboardDidHide` events will be available on Android. `keyboardWillShow` as well as `keyboardWillHide` are generally not available on Android since there is no native corresponding event.

@param {function} callback function to be called when the event fires.

removeListener()

```
static removeListener(eventName, callback)
```

Removes a specific listener.

@param {string} eventName The `nativeEvent` is the string that identifies the event you're listening for. **@param {function} callback** function to be called when the event fires.

removeAllListeners()

```
static removeAllListeners(eventName)
```

Removes all listeners for a specific event type.

@param {string} eventType The native event string listeners are watching which will be removed.

dismiss()

```
static dismiss()
```

Dismisses the active keyboard and removes focus.

Layout Props

Props

- `alignContent`
- `alignItems`
- `alignSelf`
- `aspectRatio`
- `borderBottomWidth`
- `borderEndWidth`
- `borderLeftWidth`
- `borderRightWidth`
- `borderStartWidth`
- `borderTopWidth`
- `borderWidth`
- `bottom`
- `direction`
- `display`
- `end`
- `flex`
- `flexBasis`
- `flexDirection`
- `flexGrow`
- `flexShrink`
- `flexWrap`
- `height`
- `justifyContent`
- `left`
- `margin`
- `marginBottom`
- `marginEnd`
- `marginHorizontal`
- `marginLeft`
- `marginRight`
- `marginStart`
- `marginTop`
- `marginVertical`
- `maxHeight`
- `maxWidth`
- `minHeight`
- `minWidth`
- `overflow`
- `padding`
- `paddingBottom`
- `paddingEnd`
- `paddingHorizontal`
- `paddingLeft`

- `paddingRight`
 - `paddingStart`
 - `paddingTop`
 - `paddingVertical`
 - `position`
 - `right`
 - `start`
 - `top`
 - `width`
 - `zIndex`
-

Reference

Props

`alignContent`

`alignContent` controls how rows align in the cross direction, overriding the `alignContent` of the parent. See <https://developer.mozilla.org/en-US/docs/Web/CSS/align-content> for more details.

Type	Required
<code>enum('flex-start', 'flex-end', 'center', 'stretch', 'space-between', 'space-around')</code>	No

`alignItems`

`alignItems` aligns children in the cross direction. For example, if children are flowing vertically, `alignItems` controls how they align horizontally. It works like `align-items` in CSS (default: stretch). See <https://developer.mozilla.org/en-US/docs/Web/CSS/align-items> for more details.

Type	Required
<code>enum('flex-start', 'flex-end', 'center', 'stretch', 'baseline')</code>	No

`alignSelf`

`alignSelf` controls how a child aligns in the cross direction, overriding the `alignItems` of the parent. It works like `align-self` in CSS (default: auto). See <https://developer.mozilla.org/en-US/docs/Web/CSS/align-self> for more details.

Type	Required
<code>enum('auto', 'flex-start', 'flex-end', 'center', 'stretch', 'baseline')</code>	No

aspectRatio

Aspect ratio controls the size of the undefined dimension of a node. Aspect ratio is a non-standard property only available in React Native and not CSS.

- On a node with a set width/height aspect ratio controls the size of the unset dimension
- On a node with a set flex basis aspect ratio controls the size of the node in the cross axis if unset
- On a node with a measure function aspect ratio works as though the measure function measures the flex basis
- On a node with flex grow/shrink aspect ratio controls the size of the node in the cross axis if unset
- Aspect ratio takes min/max dimensions into account

Type	Required
number	No

borderBottomWidth

`borderBottomWidth` works like `border-bottom-width` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/border-bottom-width> for more details.

Type	Required
number	No

borderEndWidth

When direction is `ltr`, `borderEndWidth` is equivalent to `borderRightWidth`. When direction is `rtl`, `borderEndWidth` is equivalent to `borderLeftWidth`.

Type	Required
number	No

borderLeftWidth

`borderLeftWidth` works like `border-left-width` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/border-left-width> for more details.

Type	Required
number	No

borderRightWidth

`borderRightWidth` works like `border-right-width` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/border-right-width> for more details.

Type	Required
number	No

borderStartWidth

When direction is `ltr`, `borderStartWidth` is equivalent to `borderLeftWidth`. When direction is `rtl`, `borderStartWidth` is equivalent to `borderRightWidth`.

Type	Required
number	No

borderTopWidth

`borderTopWidth` works like `border-top-width` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/border-top-width> for more details.

Type	Required
number	No

borderWidth

`borderWidth` works like `border-width` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/border-width> for more details.

Type	Required
number	No

bottom

`bottom` is the number of logical pixels to offset the bottom edge of this component.

It works similarly to `bottom` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported.

See <https://developer.mozilla.org/en-US/docs/Web/CSS/bottom> for more details of how `bottom` affects layout.

Type	Required
number, string	No

direction

`direction` specifies the directional flow of the user interface. The default is `inherit`, except for root node which will have value based on the current locale. See <https://facebook.github.io/yoga/docs/rtl/> for more details.

Type	Required	Platform
<code>enum('inherit', 'ltr', 'rtl')</code>	No	iOS

display

`display` sets the display type of this component.

It works similarly to `display` in CSS, but only support 'flex' and 'none'. 'flex' is the default.

Type	Required
<code>enum('none', 'flex')</code>	No

end

When the direction is `ltr`, `end` is equivalent to `right`. When the direction is `rtl`, `end` is equivalent to `left`.

This style takes precedence over the `left` and `right` styles.

Type	Required
<code>number, string</code>	No

flex

In React Native `flex` does not work the same way that it does in CSS. `flex` is a number rather than a string, and it works according to the `yoga` library at <https://github.com/facebook/yoga>

When `flex` is a positive number, it makes the component flexible and it will be sized proportional to its flex value. So a component with `flex` set to 2 will take twice the space as a component with `flex` set to 1.

When `flex` is 0, the component is sized according to `width` and `height` and it is inflexible.

When `flex` is -1, the component is normally sized according `width` and `height`. However, if there's not enough space, the component will shrink to its `minWidth` and `minHeight`.

`flexGrow`, `flexShrink`, and `flexBasis` work the same as in CSS.

Type	Required
<code>number</code>	No

flexBasis

Type	Required

number, string	No
----------------	----

flexDirection

`flexDirection` controls which directions children of a container go. `row` goes left to right, `column` goes top to bottom, and you may be able to guess what the other two do. It works like `flex-direction` in CSS, except the default is `column`. See <https://developer.mozilla.org/en-US/docs/Web/CSS/flex-direction> for more details.

Type	Required
enum('row', 'row-reverse', 'column', 'column-reverse')	No

flexGrow

Type	Required
number	No

flexShrink

Type	Required
number	No

flexWrap

`flexWrap` controls whether children can wrap around after they hit the end of a flex container. It works like `flex-wrap` in CSS (default: nowrap). See <https://developer.mozilla.org/en-US/docs/Web/CSS/flex-wrap> for more details. Note it does not work anymore with `alignItems: stretch` (the default), so you may want to use `alignItems: flex-start` for example (breaking change details: <https://github.com/facebook/react-native/releases/tag/v0.28.0>).

Type	Required
enum('wrap', 'nowrap')	No

height

`height` sets the height of this component.

It works similarly to `height` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported. See <https://developer.mozilla.org/en-US/docs/Web/CSS/height> for more details.

Type	Required
number, string	No

justifyContent

`justifyContent` aligns children in the main direction. For example, if children are flowing vertically, `justifyContent` controls how they align vertically. It works like `justify-content` in CSS (default: flex-start). See <https://developer.mozilla.org/en-US/docs/Web/CSS/justify-content> for more details.

Type	Required
enum('flex-start', 'flex-end', 'center', 'space-between', 'space-around', 'space-evenly')	No

left

`left` is the number of logical pixels to offset the left edge of this component.

It works similarly to `left` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported.

See <https://developer.mozilla.org/en-US/docs/Web/CSS/left> for more details of how `left` affects layout.

Type	Required
number, string	No

margin

Setting `margin` has the same effect as setting each of `marginTop`, `marginLeft`, `marginBottom`, and `marginRight`. See <https://developer.mozilla.org/en-US/docs/Web/CSS/margin> for more details.

Type	Required
number, string	No

marginBottom

`marginBottom` works like `margin-bottom` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/margin-bottom> for more details.

Type	Required
number, string	No

marginEnd

When direction is `ltr`, `marginEnd` is equivalent to `marginRight`. When direction is `rtl`, `marginEnd` is equivalent to `marginLeft`.

Type	Required

number, string	No
----------------	----

marginHorizontal

Setting `marginHorizontal` has the same effect as setting both `marginLeft` and `marginRight`.

Type	Required
number, string	No

marginLeft

`marginLeft` works like `margin-left` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/margin-left> for more details.

Type	Required
number, string	No

marginRight

`marginRight` works like `margin-right` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/margin-right> for more details.

Type	Required
number, string	No

marginStart

When direction is `ltr`, `marginStart` is equivalent to `marginLeft`. When direction is `rtl`, `marginStart` is equivalent to `marginRight`.

Type	Required
number, string	No

marginTop

`marginTop` works like `margin-top` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/margin-top> for more details.

Type	Required
number, string	No

marginVertical

Setting `marginVertical` has the same effect as setting both `marginTop` and `marginBottom`.

Type	Required
number, string	No

maxHeight

`maxHeight` is the maximum height for this component, in logical pixels.

It works similarly to `max-height` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported.

See <https://developer.mozilla.org/en-US/docs/Web/CSS/max-height> for more details.

Type	Required
number, string	No

maxWidth

`maxWidth` is the maximum width for this component, in logical pixels.

It works similarly to `max-width` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported.

See <https://developer.mozilla.org/en-US/docs/Web/CSS/max-width> for more details.

Type	Required
number, string	No

minHeight

`minHeight` is the minimum height for this component, in logical pixels.

It works similarly to `min-height` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported.

See <https://developer.mozilla.org/en-US/docs/Web/CSS/min-height> for more details.

Type	Required
number, string	No

minWidth

`minWidth` is the minimum width for this component, in logical pixels.

It works similarly to `min-width` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported.

See <https://developer.mozilla.org/en-US/docs/Web/CSS/min-width> for more details.

Type	Required
number, string	No

overflow

`overflow` controls how children are measured and displayed. `overflow: hidden` causes views to be clipped while `overflow: scroll` causes views to be measured independently of their parents main axis. It works like `overflow` in CSS (default: visible). See <https://developer.mozilla.org/en/docs/Web/CSS/overflow> for more details. `overflow: visible` only works on iOS. On Android, all views will clip their children.

Type	Required
enum('visible', 'hidden', 'scroll')	No

padding

Setting `padding` has the same effect as setting each of `paddingTop`, `paddingBottom`, `paddingLeft`, and `paddingRight`. See <https://developer.mozilla.org/en-US/docs/Web/CSS/padding> for more details.

Type	Required
number, string	No

paddingBottom

`paddingBottom` works like `padding-bottom` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/padding-bottom> for more details.

Type	Required
number, string	No

paddingEnd

When direction is `ltr`, `paddingEnd` is equivalent to `paddingRight`. When direction is `rtl`, `paddingEnd` is equivalent to `paddingLeft`.

Type	Required
number, string	No

paddingHorizontal

Setting `paddingHorizontal` is like setting both of `paddingLeft` and `paddingRight`.

Type	Required
number, string	No

paddingLeft

`paddingLeft` works like `padding-left` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/padding-left> for more details.

Type	Required
number, string	No

paddingRight

`paddingRight` works like `padding-right` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/padding-right> for more details.

Type	Required
number, string	No

paddingStart

When direction is `ltr`, `paddingStart` is equivalent to `paddingLeft`. When direction is `rtl`, `paddingStart` is equivalent to `paddingRight`.

Type	Required
number, string	No

paddingTop

`paddingTop` works like `padding-top` in CSS. See <https://developer.mozilla.org/en-US/docs/Web/CSS/padding-top> for more details.

Type	Required
number, string	No

paddingVertical

Setting `paddingVertical` is like setting both of `paddingTop` and `paddingBottom`.

Type	Required
number, string	No

position

`position` in React Native is similar to regular CSS, but everything is set to `relative` by default, so `absolute` positioning is always just relative to the parent.

If you want to position a child using specific numbers of logical pixels relative to its parent, set the child to have `absolute` position.

If you want to position a child relative to something that is not its parent, just don't use styles for that. Use the component tree.

See <https://github.com/facebook/yoga> for more details on how `position` differs between React Native and CSS.

Type	Required
enum('absolute', 'relative')	No

right

`right` is the number of logical pixels to offset the right edge of this component.

It works similarly to `right` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported.

See <https://developer.mozilla.org/en-US/docs/Web/CSS/right> for more details of how `right` affects layout.

Type	Required
number, string	No

start

When the direction is `ltr`, `start` is equivalent to `left`. When the direction is `rtl`, `start` is equivalent to `right`.

This style takes precedence over the `left`, `right`, and `end` styles.

Type	Required
number, string	No

top

`top` is the number of logical pixels to offset the top edge of this component.

It works similarly to `top` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported.

See <https://developer.mozilla.org/en-US/docs/Web/CSS/top> for more details of how `top` affects layout.

Type	Required
number, string	No

width

`width` sets the width of this component.

It works similarly to `width` in CSS, but in React Native you must use points or percentages. Ems and other units are not supported. See <https://developer.mozilla.org/en-US/docs/Web/CSS/width> for more details.

Type	Required
number, string	No

zIndex

`zIndex` controls which components display on top of others. Normally, you don't use `zIndex`. Components render according to their order in the document tree, so later components draw over earlier ones. `zIndex` may be useful if you have animations or custom modal interfaces where you don't want this behavior.

It works like the CSS `z-index` property - components with a larger `zIndex` will render on top. Think of the z-direction like it's pointing from the phone into your eyeball. See <https://developer.mozilla.org/en-US/docs/Web/CSS/z-index> for more details.

On iOS, `zIndex` may require `view`s to be siblings of each other for it to work as expected.

Type	Required
number	No

LayoutAnimation

Automatically animates views to their new positions when the next layout happens.

A common way to use this API is to call it before calling `setState`.

Note that in order to get this to work on **Android** you need to set the following flags via `UIManager`:

```
UIManager.setLayoutAnimationEnabledExperimental && UIManager.setLayoutAnimationEnabledExperimental(true);
```

Methods

- `configureNext`
- `create`
- `checkConfig`

Properties

- `Types`
- `Properties`
- `Presets`
- `easeInEaseOut`
- `linear`
- `spring`

Reference

Methods

`configureNext()`

```
static configureNext(config, onAnimationDidEnd?)
```

Schedules an animation to happen on the next layout.

Parameters:

Name	Type	Required	Description
config	object	Yes	See config parameters below.
onAnimationDidEnd	function	No	Called when the animation finished. Only supported on iOS.

`config`

- `duration` in milliseconds
 - `create`, config for animating in new views (see `Anim` type)
 - `update`, config for animating views that have been updated (see `Anim` type)
-

create()

```
static create(duration, type, creationProp)
```

Helper for creating a config for `configureNext`.

checkConfig()

```
static checkConfig(config, location, name)
```

Properties

Types

An enumerate of animation types to be used in `create` method.

Types
spring
linear
easeInEaseOut
easeIn
easeOut
keyboard

Properties

An enumerate of object property to be animated, used in `create` method.

Properties
opacity
scaleX
scaleY
scaleXY

Presets

A set of predefined animation config.

Presets	Value
easeInEaseOut	<code>create(300, 'easeInEaseOut', 'opacity')</code>
linear	<code>create(500, 'linear', 'opacity')</code>
spring	<code>{ duration: 700, create: { type: 'linear', property: 'opacity' }, update: { type: 'spring', springDamping: 0.4 }, delete: { type: 'linear', property: 'opacity' } }</code>

easeInEaseOut

Shortcut to bind `configureNext()` methods with `Presets.easeInEaseOut`.

linear

Shortcut to bind `configureNext()` methods with `Presets.linear`.

spring

Shortcut to bind `configureNext()` methods with `Presets.spring`.

ListDataSource

Provides efficient data processing and access to the `ListView` component. A `ListDataSource` is created with functions for extracting data from the input blob, and comparing elements (with default implementations for convenience). The input blob can be as simple as an array of strings, or an object with rows nested inside section objects.

To update the data in the datasource, use `cloneWithRows` (or `cloneWithRowsAndSections` if you care about sections). The data in the data source is immutable, so you can't modify it directly. The clone methods suck in the new data and compute a diff for each row so `ListView` knows whether to re-render it or not.

In this example, a component receives data in chunks, handled by `_onDataArrived`, which concats the new data onto the old data and updates the data source. We use `concat` to create a new array - mutating `this._data`, e.g. with `this._data.push(newRowData)`, would be an error. `_rowHasChanged` understands the shape of the row data and knows how to efficiently compare it.

```
getInitialState: function() {
  var ds = new ListView.DataSource({rowHasChanged: this._rowHasChanged});
  return {ds};
},
_onDataArrived(newData) {
  this._data = this._data.concat(newData);
  this.setState({
    ds: this.state.ds.cloneWithRows(this._data)
  });
}
```

Methods

- `constructor`
- `cloneWithRows`
- `cloneWithRowsAndSections`
- `getRowCount`
- `getRowAndSectionCount`
- `rowShouldUpdate`
- `getRowData`
- `getRowIDForFlatIndex`
- `getSectionIDForFlatIndex`
- `getSectionLengths`
- `sectionHeaderShouldUpdate`
- `getSectionHeaderData`

Reference

Methods

constructor()

```
constructor(params);
```

You can provide custom extraction and `hasChanged` functions for section headers and rows. If absent, data will be extracted with the `defaultGetRowData` and `defaultGetSectionHeaderData` functions.

The default extractor expects data of one of the following forms:

```
{ sectionID_1: { rowID_1: <rowData1>, ... }, ... }
```

or

```
{ sectionID_1: [ <rowData1>, <rowData2>, ... ], ... }
```

or

```
[ [ <rowData1>, <rowData2>, ... ], ... ]
```

The constructor takes in a `params` argument that can contain any of the following:

- `getRowData(dataBlob, sectionID, rowID);`
- `getSectionHeaderData(dataBlob, sectionID);`
- `rowHasChanged(prevRowData, nextRowData);`
- `sectionHeaderHasChanged(prevSectionData, nextSectionData);`

cloneWithRows()

```
cloneWithRows(dataBlob, rowIdentities);
```

Clones this `ListViewDataSource` with the specified `dataBlob` and `rowIdentities`. The `dataBlob` is just an arbitrary blob of data. At construction an extractor to get the interesting information was defined (or the default was used).

The `rowIdentities` is a 2D array of identifiers for rows. ie. `[['a1', 'a2'], ['b1', 'b2', 'b3'], ...]`. If not provided, it's assumed that the keys of the section data are the row identities.

Note: This function does NOT clone the data in this data source. It simply passes the functions defined at construction to a new data source with the data specified. If you wish to maintain the existing data you must handle merging of old and new data separately and then pass that into this function as the `dataBlob`.

cloneWithRowsAndSections()

```
cloneWithRowsAndSections(dataBlob, sectionIdentities, rowIdentities);
```

This performs the same function as the `cloneWithRows` function but here you also specify what your `sectionIdentities` are. If you don't care about sections you should safely be able to use `cloneWithRows`.

`sectionIdentities` is an array of identifiers for sections. ie. `['s1', 's2', ...]`. The identifiers should correspond to the keys or array indexes of the data you wish to include. If not provided, it's assumed that the keys of `dataBlob` are the section identities.

Note: this returns a new object!

```
const dataSource = ds.cloneWithRowsAndSections({
  addresses: ['row 1', 'row 2'],
  phone_numbers: ['data 1', 'data 2'],
}, ['phone_numbers']);
```

getRowCount()

```
getRowCount();
```

Returns the total number of rows in the data source.

If you are specifying the `rowIdentities` or `sectionIdentities`, then `getRowCount` will return the number of rows in the filtered data source.

getRowAndSectionCount()

```
getRowAndSectionCount();
```

Returns the total number of rows in the data source (see `getRowCount` for how this is calculated) plus the number of sections in the data.

If you are specifying the `rowIdentities` or `sectionIdentities`, then `getRowAndSectionCount` will return the number of rows & sections in the filtered data source.

rowShouldUpdate()

```
rowShouldUpdate(sectionIndex, rowIndex);
```

Returns if the row is dirtied and needs to be rerendered

getRowData()

```
getRowData(sectionIndex, rowIndex);
```

Gets the data required to render the row.

getRowIDForFlatIndex()

```
getRowIDForFlatIndex(index);
```

Gets the rowID at index provided if the dataSource arrays were flattened, or null of out of range indexes.

getSectionIDForFlatIndex()

```
getSectionIDForFlatIndex(index);
```

Gets the sectionID at index provided if the dataSource arrays were flattened, or null for out of range indexes.

getSectionLengths()

```
getSectionLengths();
```

Returns an array containing the number of rows in each section

sectionHeaderShouldUpdate()

```
sectionHeaderShouldUpdate(sectionIndex);
```

Returns if the section header is dirtied and needs to be rerendered

getSectionHeaderData()

```
getSectionHeaderData(sectionIndex);
```

Gets the data required to render the section header

NetInfo

NetInfo exposes info about online/offline status

```
NetInfo.getConnectionInfo().then((connectionInfo) => {
  console.log('Initial, type: ' + connectionInfo.type + ', effectiveType: ' + connectionInfo.effectiveType);
});
function handleFirstConnectivityChange(connectionInfo) {
  console.log('First change, type: ' + connectionInfo.type + ', effectiveType: ' + connectionInfo.effectiveType);
  NetInfo.removeEventListener(
    'connectionChange',
    handleFirstConnectivityChange
  );
}
NetInfo.addEventListener(
  'connectionChange',
  handleFirstConnectivityChange
);
```

ConnectionType enum

`ConnectionType` describes the type of connection the device is using to communicate with the network.

Cross platform values for `ConnectionType` :

- `none` - device is offline
- `wifi` - device is online and connected via wifi, or is the iOS simulator
- `cellular` - device is connected via Edge, 3G, WiMax, or LTE
- `unknown` - error case and the network status is unknown

Android-only values for `ConnectionType` :

- `bluetooth` - device is connected via Bluetooth
- `ethernet` - device is connected via Ethernet
- `wimax` - device is connected via WiMAX

EffectiveConnectionType enum

Cross platform values for `EffectiveConnectionType` :

- `2g`
- `3g`
- `4g`
- `unknown`

Android

To request network info, you need to add the following line to your app's `AndroidManifest.xml` :

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

Methods

- `addEventListener`
- `removeEventListener`
- `getConnectionInfo`
- `isConnectionExpensive`

Properties

- `isConnected`
-

Reference

Methods

`addEventListener()`

```
NetInfo.addEventListener(eventName, handler);
```

Adds an event handler.

Parameters:

Name	Type	Required	Description
eventName	enum(connectionChange, change)	Yes	The change event name.
handler	function	Yes	Listener function.

Supported events:

- `connectionChange` : Fires when the network status changes. The argument to the event handler is an object with keys:
 - `type` : A `ConnectionType` (listed above)
 - `effectiveType` : An `EffectiveConnectionType` (listed above)
 - `change` : This event is deprecated. Listen to `connectionChange` instead. Fires when the network status changes. The argument to the event handler is one of the deprecated connectivity types listed above.
-

`removeEventListener()`

```
NetInfo.removeEventListener(eventName, handler);
```

Removes the listener for network status changes.

Parameters:

Name	Type	Required	Description

eventName	enum(connectionChange, change)	Yes	The change event name.
handler	function	Yes	Listener function.

getConnectionInfo()

```
NetInfo.getConnectionInfo();
```

Returns a promise that resolves to an object with `type` and `effectiveType` keys whose values are a `ConnectionType` and an `EffectiveConnectionType`, respectively.

isConnectionExpensive()

```
NetInfo.isConnectionExpensive();
```

Available on Android. Detect if the current active connection is metered or not. A network is classified as metered when the user is sensitive to heavy data usage on that connection due to monetary costs, data limitations or battery/performance issues.

```
NetInfo.isConnectionExpensive()
  .then(isConnectionExpensive => {
    console.log('Connection is ' + (isConnectionExpensive ? 'Expensive' : 'Not Expensive'));
  })
  .catch(error => {
    console.error(error);
});
});
```

Properties

isConnected

Available on all platforms. Asynchronously fetch a boolean to determine internet connectivity.

```
NetInfo.isConnected.fetch().then(isConnected => {
  console.log('First, is ' + (isConnected ? 'online' : 'offline'));
});
function handleFirstConnectivityChange(isConnected) {
  console.log('Then, is ' + (isConnected ? 'online' : 'offline'));
  NetInfo.isConnected.removeEventListener(
    'connectionChange',
    handleFirstConnectivityChange
  );
}
NetInfo.isConnected.addEventListener(
  'connectionChange',
  handleFirstConnectivityChange
);
```


PanResponder

`PanResponder` reconciles several touches into a single gesture. It makes single-touch gestures resilient to extra touches, and can be used to recognize simple multi-touch gestures.

By default, `PanResponder` holds an `InteractionManager` handle to block long-running JS events from interrupting active gestures.

It provides a predictable wrapper of the responder handlers provided by the [gesture responder system](#). For each handler, it provides a new `gestureState` object alongside the native event object:

```
onPanResponderMove: (event, gestureState) => {}
```

A native event is a synthetic touch event with the following form:

- `nativeEvent`
 - `changedTouches` - Array of all touch events that have changed since the last event
 - `identifier` - The ID of the touch
 - `locationX` - The X position of the touch, relative to the element
 - `locationY` - The Y position of the touch, relative to the element
 - `pageX` - The X position of the touch, relative to the root element
 - `pageY` - The Y position of the touch, relative to the root element
 - `target` - The node id of the element receiving the touch event
 - `timestamp` - A time identifier for the touch, useful for velocity calculation
 - `touches` - Array of all current touches on the screen

A `gestureState` object has the following:

- `stateID` - ID of the gestureState- persisted as long as there at least one touch on screen
- `moveX` - the latest screen coordinates of the recently-moved touch
- `moveY` - the latest screen coordinates of the recently-moved touch
- `x0` - the screen coordinates of the responder grant
- `y0` - the screen coordinates of the responder grant
- `dx` - accumulated distance of the gesture since the touch started
- `dy` - accumulated distance of the gesture since the touch started
- `vx` - current velocity of the gesture
- `vy` - current velocity of the gesture
- `numberActiveTouches` - Number of touches currently on screen

Basic Usage

```
class ExampleComponent extends Component {
  constructor(props) {
    super(props)
    this._panResponder = PanResponder.create({
      // Ask to be the responder:
      onStartShouldSetPanResponder: (evt, gestureState) => true,
      onStartShouldSetPanResponderCapture: (evt, gestureState) => true,
      onMoveShouldSetPanResponder: (evt, gestureState) => true,
```

```

onMoveShouldSetPanResponderCapture: (evt, gestureState) => true,
onPanResponderGrant: (evt, gestureState) => {
  // The gesture has started. Show visual feedback so the user knows
  // what is happening!

  // gestureState.d{x,y} will be set to zero now
},
onPanResponderMove: (evt, gestureState) => {
  // The most recent move distance is gestureState.move{X,Y}

  // The accumulated gesture distance since becoming responder is
  // gestureState.d{x,y}
},
onPanResponderTerminationRequest: (evt, gestureState) => true,
onPanResponderRelease: (evt, gestureState) => {
  // The user has released all touches while this view is the
  // responder. This typically means a gesture has succeeded
},
onPanResponderTerminate: (evt, gestureState) => {
  // Another component has become the responder, so this gesture
  // should be cancelled
},
onShouldBlockNativeResponder: (evt, gestureState) => {
  // Returns whether this component should block native components from becoming the JS
  // responder. Returns true by default. Is currently only supported on android.
  return true;
},
);
};

render() {
  return (
    <View {...this._panResponder.panHandlers} />
  );
}
}

```

Working Example

To see it in action, try the [PanResponder example in RNTester](#)

Methods

- [create](#)
-

Reference

Methods

[create\(\)](#)

```
static create(config)
```

`@param {object} config` Enhanced versions of all of the responder callbacks that provide not only the typical `ResponderSyntheticEvent`, but also the `PanResponder` gesture state. Simply replace the word `Responder` with `PanResponder` in each of the typical `onResponder*` callbacks. For example, the `config` object would look like:

- `onMoveShouldSetPanResponder: (e, gestureState) => {...}`
- `onMoveShouldSetPanResponderCapture: (e, gestureState) => {...}`
- `onStartShouldSetPanResponder: (e, gestureState) => {...}`
- `onStartShouldSetPanResponderCapture: (e, gestureState) => {...}`
- `onPanResponderReject: (e, gestureState) => {...}`
- `onPanResponderGrant: (e, gestureState) => {...}`
- `onPanResponderStart: (e, gestureState) => {...}`
- `onPanResponderEnd: (e, gestureState) => {...}`
- `onPanResponderRelease: (e, gestureState) => {...}`
- `onPanResponderMove: (e, gestureState) => {...}`
- `onPanResponderTerminate: (e, gestureState) => {...}`
- `onPanResponderTerminationRequest: (e, gestureState) => {...}`
- `onShouldBlockNativeResponder: (e, gestureState) => {...}`

In general, for events that have capture equivalents, we update the `gestureState` once in the capture phase and can use it in the bubble phase as well.

Be careful with `onStartShould*` callbacks. They only reflect updated `gestureState` for start/end events that bubble/capture to the Node. Once the node is the responder, you can rely on every start/end event being processed by the gesture and `gestureState` being updated accordingly. (`numberActiveTouches`) may not be totally accurate unless you are the responder.

PixelRatio

PixelRatio class gives access to the device pixel density.

Fetching a correctly sized image

You should get a higher resolution image if you are on a high pixel density device. A good rule of thumb is to multiply the size of the image you display by the pixel ratio.

```
var image = getImage({
  width: PixelRatio.getPixelSizeForLayoutSize(200),
  height: PixelRatio.getPixelSizeForLayoutSize(100),
});
<Image source={image} style={{width: 200, height: 100}} />
```

Pixel grid snapping

In iOS, you can specify positions and dimensions for elements with arbitrary precision, for example 29.674825. But, ultimately the physical display only have a fixed number of pixels, for example 640×960 for iPhone 4 or 750×1334 for iPhone 6. iOS tries to be as faithful as possible to the user value by spreading one original pixel into multiple ones to trick the eye. The downside of this technique is that it makes the resulting element look blurry.

In practice, we found out that developers do not want this feature and they have to work around it by doing manual rounding in order to avoid having blurry elements. In React Native, we are rounding all the pixels automatically.

We have to be careful when to do this rounding. You never want to work with rounded and unrounded values at the same time as you're going to accumulate rounding errors. Having even one rounding error is deadly because a one pixel border may vanish or be twice as big.

In React Native, everything in JavaScript and within the layout engine works with arbitrary precision numbers. It's only when we set the position and dimensions of the native element on the main thread that we round. Also, rounding is done relative to the root rather than the parent, again to avoid accumulating rounding errors.

Methods

- [get](#)
- [getFontSize](#)
- [getPixelSizeForLayoutSize](#)
- [roundToNearestPixel](#)

Reference

Methods

get()

```
static get()
```

Returns the device pixel density. Some examples:

- PixelRatio.get() === 1
 - mdpi Android devices
- PixelRatio.get() === 1.5
 - hdpi Android devices
- PixelRatio.get() === 2
 - iPhone 4, 4S
 - iPhone 5, 5C, 5S
 - iPhone 6, 7, 8
 - iPhone XR
 - xhdpi Android devices
- PixelRatio.get() === 3
 - iPhone 6 Plus, 7 Plus, 8 Plus
 - iPhone X, XS, XS Max
 - Pixel, Pixel 2
 - xxhdpi Android devices
- PixelRatio.get() === 3.5
 - Nexus 6
 - Pixel XL, Pixel 2 XL
 - xxxhdpi Android devices

getFontSize()

```
static getFontSize()
```

Returns the scaling factor for font sizes. This is the ratio that is used to calculate the absolute font size, so any elements that heavily depend on that should use this to do calculations.

If a font scale is not set, this returns the device pixel ratio.

Currently this is only implemented on Android and reflects the user preference set in Settings > Display > Font size, on iOS it will always return the default pixel ratio. @platform android

getPixelSizeForLayoutSize()

```
static getPixelSizeForLayoutSize(layoutSize)
```

Converts a layout size (dp) to pixel size (px).

Guaranteed to return an integer number.

roundToNearestPixel()

```
static roundToNearestPixel(layoutSize)
```

Rounds a layout size (dp) to the nearest layout size that corresponds to an integer number of pixels. For example, on a device with a PixelRatio of 3, `PixelRatio.roundToNearestPixel(8.4) = 8.33`, which corresponds to exactly $(8.33 * 3) = 25$ pixels.

Settings

`Settings` serves as a wrapper for `NSUserDefaults`, a persistent key-value store available only on iOS.

Methods

- `get`
 - `set`
 - `watchKeys`
 - `clearWatch`
-

Reference

Methods

`get()`

```
static get(key)
```

Get the current value for a key in `NSUserDefaults`.

`set()`

```
static set(settings)
```

Set one or more values in `NSUserDefaults`.

`watchKeys()`

```
static watchKeys(keys, callback)
```

Subscribe to be notified when the value for any of the keys specified by the `keys` array changes in `NSUserDefaults`. Returns a `watchId` number that may be used with `clearWatch()` to unsubscribe.

`clearWatch()`

```
static clearWatch(watchId)
```

`watchId` is the number returned by `watchKeys()` when the subscription was originally configured.

Shadow Props

Props

- `shadowColor`
 - `shadowOffset`
 - `shadowOpacity`
 - `shadowRadius`
-

Reference

Props

`shadowColor`

Sets the drop shadow color

Type	Required	Platform
<code>color</code>	No	iOS

`shadowOffset`

Sets the drop shadow offset

Type	Required	Platform
<code>object: {width: number,height: number}</code>	No	iOS

`shadowOpacity`

Sets the drop shadow opacity (multiplied by the color's alpha component)

Type	Required	Platform
<code>number</code>	No	iOS

`shadowRadius`

Sets the drop shadow blur radius

Type	Required	Platform

number

No

iOS

Share

Methods

- `share`
 - `sharedAction`
 - `dismissedAction`
-

Reference

Methods

`share()`

```
static share(content, options)
```

Open a dialog to share text content.

In iOS, Returns a Promise which will be invoked an object containing `action` , `activityType` . If the user dismissed the dialog, the Promise will still be resolved with action being `Share.dismissedAction` and all the other keys being undefined.

In Android, Returns a Promise which always be resolved with action being `share.sharedAction` .

Content

- `message` - a message to share
- `title` - title of the message

iOS

- `url` - an URL to share

At least one of URL and message is required.

Options

iOS

- `subject` - a subject to share via email
- `excludedActivityTypes`
- `tintColor`

Android

- `dialogTitle`
-

sharedAction()

```
static sharedAction()
```

The content was successfully shared.

dismissedAction()

```
static dismissedAction()
```

iOS Only. The dialog has been dismissed.

Basic Example

```
import React, {Component} from 'react'
import {Share, Button} from 'react-native'

class ShareExample extends Component {

  async onShare = () => {
    try {
      const result = await Share.share({
        message:
          'React Native | A framework for building native apps using React',
    })
    if (result.action === Share.sharedAction) {
      if (result.activityType) {
        // shared with activity type of result.activityType
      } else {
        // shared
      }
    } else if (result.action === Share.dismissedAction) {
      // dismissed
    }
  } catch (error) {
    alert(error.message);
  }
}

render() {
  return (
    <Button onPress={this.onShare}>Share</Button>
  );
}
}
```


StatusBarIOS

Use `StatusBar` for mutating the status bar.

Reference

StyleSheet

A StyleSheet is an abstraction similar to CSS StyleSheets

Create a new StyleSheet:

```
const styles = StyleSheet.create({
  container: {
    borderRadius: 4,
    borderWidth: 0.5,
    borderColor: '#d6d7da',
  },
  title: {
    fontSize: 19,
    fontWeight: 'bold',
  },
  activeTitle: {
    color: 'red',
  },
});
```

Use a StyleSheet:

```
<View style={styles.container}>
  <Text style={[styles.title, this.props.isActive && styles.activeTitle]} />
</View>
```

Code quality:

- By moving styles away from the render function, you're making the code easier to understand.
- Naming the styles is a good way to add meaning to the low level components in the render function.

Methods

- `setStyleAttributePreprocessor`
- `create`
- `flatten`

Properties

- `hairlineWidth`
- `absoluteFill`
- `absoluteFillObject`

Reference

Methods

setStyleAttributePreprocessor()

```
static setStyleAttributePreprocessor(property, process)
```

WARNING: EXPERIMENTAL. Breaking changes will probably happen a lot and will not be reliably announced. The whole thing might be deleted, who knows? Use at your own risk.

Sets a function to use to pre-process a style property value. This is used internally to process color and transform values. You should not use this unless you really know what you are doing and have exhausted other options.

create()

```
static create(obj)
```

Creates a StyleSheet style reference from the given object.

flatten

```
static flatten(style)
```

Flattens an array of style objects, into one aggregated style object. Alternatively, this method can be used to lookup IDs, returned by `StyleSheet.register`.

NOTE: Exercise caution as abusing this can tax you in terms of optimizations. IDs enable optimizations through the bridge and memory in general. Referring to style objects directly will deprive you of these optimizations.

Example:

```
var styles = StyleSheet.create({
  listItem: {
    flex: 1,
    fontSize: 16,
    color: 'white',
  },
  selectedListItem: {
    color: 'green',
  },
});

StyleSheet.flatten([styles.listItem, styles.selectedListItem]);
// returns { flex: 1, fontSize: 16, color: 'green' }
```

Alternative use:

```
var styles = StyleSheet.create({
```

```

listItem: {
  flex: 1,
  fontSize: 16,
  color: 'white',
},
selectedListItem: {
  color: 'green',
},
});

StyleSheet.flatten(styles.listItem);
// return { flex: 1, fontSize: 16, color: 'white' }
// Simply styles.listItem would return its ID (number)

```

This method internally uses `StyleSheetRegistry.getStyleByID(style)` to resolve style objects represented by IDs. Thus, an array of style objects (instances of `StyleSheet.create()`), are individually resolved to, their respective objects, merged as one and then returned. This also explains the alternative use.

Properties

hairlineWidth

```

var styles = StyleSheet.create({
  separator: {
    borderBottomColor: '#bbb',
    borderBottomWidth: StyleSheet.hairlineWidth,
  },
});

```

This constant will always be a round number of pixels (so a line defined by it can look crisp) and will try to match the standard width of a thin line on the underlying platform. However, you should not rely on it being a constant size, because on different platforms and screen densities its value may be calculated differently.

A line with hairline width may not be visible if your simulator is downscaled.

absoluteFill

A very common pattern is to create overlays with position absolute and zero positioning (`position: 'absolute', left: 0, right: 0, top: 0, bottom: 0`), so `absoluteFill` can be used for convenience and to reduce duplication of these repeated styles.

absoluteFillObject

Sometimes you may want `absoluteFill` but with a couple tweaks - `absoluteFillObject` can be used to create a customized entry in a `StyleSheet`, e.g.:

```

const styles = StyleSheet.create({
  wrapper: {
    ...StyleSheet.absoluteFillObject,
    top: 10,
  }
});

```

```
    backgroundColor: 'transparent',  
},  
});
```

Systrace

Methods

- `installReactHook`
 - `setEnabled`
 - `isEnabled`
 - `beginEvent`
 - `endEvent`
 - `beginAsyncEvent`
 - `endAsyncEvent`
 - `counterEvent`
 - `attachToRelayProfiler`
 - `swizzleJSON`
 - `measureMethods`
 - `measure`
-

Reference

Methods

`installReactHook()`

```
static installReactHook(useFiber)
```

`setEnabled()`

```
static setEnabled(enabled)
```

`isEnabled()`

```
static isEnabled()
```

`beginEvent()`

```
static beginEvent(profileName?, args?)
```

beginEvent/endEvent for starting and then ending a profile within the same call stack frame.

endEvent()

```
static endEvent()
```

beginAsyncEvent()

```
static beginAsyncEvent(profileName?)
```

beginAsyncEvent/endAsyncEvent for starting and then ending a profile where the end can either occur on another thread or out of the current stack frame, eg await the returned cookie variable should be used as input into the endAsyncEvent call to end the profile.

endAsyncEvent()

```
static endAsyncEvent(profileName?, cookie?)
```

counterEvent()

```
static counterEvent(profileName?, value?)
```

Register the value to the profileName on the systrace timeline.

attachToRelayProfiler()

```
static attachToRelayProfiler(relayProfiler)
```

Relay profiles use await calls, so likely occur out of current stack frame therefore async variant of profiling is used.

swizzleJSON()

```
static swizzleJSON()
```

This is not called by default due to performance overhead, but it's useful for finding traces which spend too much time in JSON.

measureMethods()

```
static measureMethods(object, objectName, methodNames)
```

Measures multiple methods of a class. For example, the following will return the `parse` and `stringify` methods of the JSON class: `Systrace.measureMethods(JSON, 'JSON', ['parse', 'stringify']);`

`@param object` `@param objectName` `@param methodNames` Map from method names to method display names.

measure()

```
static measure(objName, fName, func)
```

Returns a profiled version of the input function. For example, you can: `JSON.parse = Systrace.measure('JSON', 'parse', JSON.parse);`

`@param objName` `@param fName` `@param {function} func` `@return {function}` replacement function

Text Style Props

Props

- `textShadowOffset`
 - `color`
 - `fontSize`
 - `fontStyle`
 - `fontWeight`
 - `lineHeight`
 - `textAlign`
 - `textDecorationLine`
 - `textShadowColor`
 - `fontFamily`
 - `textShadowRadius`
 - `includeFontPadding`
 - `textAlignVertical`
 - `fontVariant`
 - `letterSpacing`
 - `textDecorationColor`
 - `textDecorationStyle`
 - `textTransform`
 - `writingDirection`
-

Reference

Props

`textShadowOffset`

Type	Required
<code>object: {width: number,height: number}</code>	No

`color`

Type	Required
<code>color</code>	No

`fontSize`

Type	Required
number	No

fontStyle

Type	Required
enum('normal', 'italic')	No

fontWeight

Specifies font weight. The values 'normal' and 'bold' are supported for most fonts. Not all fonts have a variant for each of the numeric values, in that case the closest one is chosen.

Type	Required
enum('normal', 'bold', '100', '200', '300', '400', '500', '600', '700', '800', '900')	No

lineHeight

Type	Required
number	No

textAlign

Specifies text alignment. The value 'justify' is only supported on iOS and fallbacks to `left` on Android.

Type	Required
enum('auto', 'left', 'right', 'center', 'justify')	No

textDecorationLine

Type	Required
enum('none', 'underline', 'line-through', 'underline line-through')	No

textShadowColor

Type	Required
color	No

fontFamily

Type	Required
string	No

textShadowRadius

Type	Required
number	No

includeFontPadding

Set to `false` to remove extra font padding intended to make space for certain ascenders / descenders. With some fonts, this padding can make text look slightly misaligned when centered vertically. For best results also set `textAlignVertical` to `center`. Default is true.

Type	Required	Platform
bool	No	Android

textAlignVertical

Type	Required	Platform
enum('auto', 'top', 'bottom', 'center')	No	Android

fontVariant

Type	Required	Platform
array of enum('small-caps', 'oldstyle-nums', 'lining-nums', 'tabular-nums', 'proportional-nums')	No	iOS

letterSpacing

Type	Required	Platform
number	No	iOS

textDecorationColor

Type	Required	Platform
color	No	iOS

textDecorationStyle

Type	Required	Platform
enum('solid', 'double', 'dotted', 'dashed')	No	iOS

textTransform

Type	Required
enum('none', 'uppercase', 'lowercase', 'capitalize')	No

writingDirection

Type	Required	Platform
enum('auto', 'ltr', 'rtl')	No	iOS

TimePickerAndroid

Opens the standard Android time picker dialog.

Example

```
try {
  const {action, hour, minute} = await TimePickerAndroid.open({
    hour: 14,
    minute: 0,
    is24Hour: false, // Will display '2 PM'
  });
  if (action !== TimePickerAndroid.dismissedAction) {
    // Selected hour (0-23), minute (0-59)
  }
} catch ({code, message}) {
  console.warn('Cannot open time picker', message);
}
```

Methods

- `open`
 - `timeSetAction`
 - `dismissedAction`
-

Reference

Methods

`open()`

```
static open(options)
```

Opens the standard Android time picker dialog.

The available keys for the `options` object are:

- `hour` (0-23) - the hour to show, defaults to the current time
- `minute` (0-59) - the minute to show, defaults to the current time
- `is24Hour` (boolean) - If `true`, the picker uses the 24-hour format. If `false`, the picker shows an AM/PM chooser. If undefined, the default for the current locale is used.
- `mode` (enum('clock', 'spinner', 'default')) - set the time picker mode
 - 'clock': Show a time picker in clock mode.
 - 'spinner': Show a time picker in spinner mode.
 - 'default': Show a default time picker based on Android versions.

Returns a Promise which will be invoked an object containing `action`, `hour` (0-23), `minute` (0-59) if the user picked a time. If the user dismissed the dialog, the Promise will still be resolved with `action` being `TimePickerAndroid.dismissedAction` and all the other keys being undefined. **Always** check whether the `action` before reading the values.

timeSetAction()

```
static timeSetAction()
```

A time has been selected.

dismissedAction()

```
static dismissedAction()
```

The dialog has been dismissed.

ToastAndroid

This exposes the native ToastAndroid module as a JS module. This has a function 'show' which takes the following parameters:

1. String message: A string with the text to toast
2. int duration: The duration of the toast. May be ToastAndroid.SHORT or ToastAndroid.LONG

There is also a function `showWithGravity` to specify the layout gravity. May be ToastAndroid.TOP, ToastAndroid.BOTTOM, ToastAndroid.CENTER.

The 'showWithGravityAndOffset' function adds on the ability to specify offset These offset values will translate to pixels.

Basic usage:

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

ToastAndroid.show('A pikachu appeared nearby !', ToastAndroid.SHORT);
ToastAndroid.showWithGravity(
  'All Your Base Are Belong To Us',
  ToastAndroid.SHORT,
  ToastAndroid.CENTER,
);
ToastAndroid.showWithGravityAndOffset(
  'A wild toast appeared!',
  ToastAndroid.LONG,
  ToastAndroid.BOTTOM,
  25,
  50,
);
```

Advanced usage:

The ToastAndroid API is imperative and this might present itself as an issue, but there is actually a way(hack) to expose a declarative component from it. See an example below:

```
import React, {Component} from 'react';
import {View, Button, ToastAndroid} from 'react-native';

// a component that calls the imperative ToastAndroid API
const Toast = (props) => {
  if (props.visible) {
    ToastAndroid.showWithGravityAndOffset(
      props.message,
      ToastAndroid.LONG,
      ToastAndroid.BOTTOM,
      25,
      50,
    );
    return null;
  }
  return null;
};
```

```

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      visible: false,
    };
  }

  handleButtonPress = () => {
    this.setState(
      {
        visible: true,
      },
      () => {
        this.hideToast();
      },
    );
  };

  hideToast = () => {
    this.setState({
      visible: false,
    });
  };

  render() {
    return (
      <View style={styles.container}>
        <Toast visible={this.state.visible} message="Example" />
        <Button title="Toggle Modal" onPress={this.handleButtonPress} />
      </View>
    );
  }
}

```

Methods

- `show`
- `showWithGravity`
- `showWithGravityAndOffset`

Properties

- `SHORT`
 - `LONG`
 - `TOP`
 - `BOTTOM`
 - `CENTER`
-

Reference

Methods

`show()`

```
static show(message, duration)
```

showWithGravity()

```
static showWithGravity(message, duration, gravity)
```

showWithGravityAndOffset()

```
static showWithGravityAndOffset(message, duration, gravity, x0ffset, y0ffset)
```

Properties

SHORT

```
ToastAndroid.SHORT;
```

LONG

```
ToastAndroid.LONG;
```

TOP

```
ToastAndroid.TOP;
```

BOTTOM

```
ToastAndroid.BOTTOM;
```

CENTER

```
ToastAndroid.CENTER;
```

Transforms

Props

- `decomposedMatrix`
 - `rotation`
 - `scaleX`
 - `scaleY`
 - `transform`
 - `transformMatrix`
 - `translateX`
 - `translateY`
-

Reference

Props

`decomposedMatrix`

Deprecated. Use the `transform` prop instead.

Type	Required
<code>DecomposedMatrixPropType</code>	No

`rotation`

Type	Required
<code>deprecatedPropType(ReactPropTypes.number, 'Use the transform prop instead.')</code>	No

`scaleX`

Type	Required
<code>deprecatedPropType(ReactPropTypes.number, 'Use the transform prop instead.')</code>	No

`scaleY`

Type	Required
<code>deprecatedPropType(ReactPropTypes.number, 'Use the transform prop instead.')</code>	No

transform

`transform` accepts an array of transformation objects. Each object specifies the property that will be transformed as the key, and the value to use in the transformation. Objects should not be combined. Use a single key/value pair per object.

The rotate transformations require a string so that the transform may be expressed in degrees (deg) or radians (rad). For example:

```
transform([{ rotateX: '45deg' }, { rotateZ: '0.785398rad' }])
```

The skew transformations require a string so that the transform may be expressed in degrees (deg). For example:

```
transform([{ skewX: '45deg' }])
```

Type	Required
array of object: {perspective: number}, ,object: {rotate: string}, ,object: {rotateX: string}, ,object: {rotateY: string}, ,object: {rotateZ: string}, ,object: {scale: number}, ,object: {scaleX: number}, ,object: {scaleY: number}, ,object: {translateX: number}, ,object: {translateY: number}, ,object: {skewX: string}, ,object: {skewY: string}	No

transformMatrix

Deprecated. Use the `transform` prop instead.

Type	Required
TransformMatrixPropType	No

translateX

Type	Required
deprecatedPropType(ReactPropTypes.number, 'Use the transform prop instead.')	No

translateY

Type	Required
deprecatedPropType(ReactPropTypes.number, 'Use the transform prop instead.')	No

Vibration

The Vibration API is exposed at `vibration.vibrate()`. The vibration is asynchronous so this method will return immediately.

There will be no effect on devices that do not support Vibration, eg. the simulator.

Note for Android: add `<uses-permission android:name="android.permission.VIBRATE"/>` to `AndroidManifest.xml`

The vibration duration in iOS is not configurable, so there are some differences with Android. In Android, if `pattern` is a number, it specifies the vibration duration in ms. If `pattern` is an array, those odd indices are the vibration duration, while the even ones are the separation time.

In iOS, invoking `vibrate(duration)` will just ignore the duration and vibrate for a fixed time. While the `pattern` array is used to define the duration between each vibration. See below example for more.

Repeatable vibration is also supported, the vibration will repeat with defined pattern until `cancel()` is called.

Example:

```
const DURATION = 10000
const PATTERN = [1000, 2000, 3000]

Vibration.vibrate(DURATION)
// Android: vibrate for 10s
// iOS: duration is not configurable, vibrate for fixed time (about 500ms)

Vibration.vibrate(PATTERN)
// Android: wait 1s -> vibrate 2s -> wait 3s
// iOS: wait 1s -> vibrate -> wait 2s -> vibrate -> wait 3s -> vibrate

Vibration.vibrate(PATTERN, true)
// Android: wait 1s -> vibrate 2s -> wait 3s -> wait 1s -> vibrate 2s -> wait 3s -> ...
// iOS: wait 1s -> vibrate -> wait 2s -> vibrate -> wait 3s -> vibrate -> wait 1s -> vibrate -> wait 2s -> vibrate -> wait 3s -> vibrate -> ...

Vibration.cancel()
// Android: vibration stopped
// iOS: vibration stopped
```

Methods

- `vibrate`
- `cancel`

Reference

Methods

`vibrate()`

```
Vibration.vibrate(pattern: number, Array<number>, repeat: boolean)
```

Trigger a vibration with specified `pattern`.

Parameters:

Name	Type	Required	Description
pattern	number or Array\	Yes	Vibration pattern, accept a number or an array of numbers. Default to 400ms.
repeat	boolean	No	Repeat vibration pattern until cancel(), default to false.

cancel()

```
Vibration.cancel();
```

Stop vibration.

```
Vibration.cancel()
```

VibrationIOS

NOTE: `VibrationIOS` is being deprecated. Use `Vibration` instead.

The Vibration API is exposed at `VibrationIOS.vibrate()`. On iOS, calling this function will trigger a one second vibration. The vibration is asynchronous so this method will return immediately.

There will be no effect on devices that do not support Vibration, eg. the iOS simulator.

Vibration patterns are currently unsupported.

Methods

- `vibrate`
-

Reference

Methods

`vibrate()`

```
static vibrate()
```

`@deprecated`

View Style Props

Props

- [Layout Props](#)
 - [Shadow Props](#)
 - [Transforms](#)
 - `borderRightColor`
 - `backfaceVisibility`
 - `borderBottomColor`
 - `borderBottomEndRadius`
 - `borderBottomLeftRadius`
 - `borderBottomRightRadius`
 - `borderBottomStartRadius`
 - `borderBottomWidth`
 - `borderColor`
 - `borderEndColor`
 - `borderLeftColor`
 - `borderLeftWidth`
 - `borderRadius`
 - `backgroundColor`
 - `borderRightWidth`
 - `borderStartColor`
 - `borderStyle`
 - `borderTopColor`
 - `borderTopEndRadius`
 - `borderTopLeftRadius`
 - `borderTopRightRadius`
 - `borderTopStartRadius`
 - `borderTopWidth`
 - `borderWidth`
 - `opacity`
 - `elevation`
-

Reference

Props

`borderRightColor`

Type	Required
<code>color</code>	No

backfaceVisibility

Type	Required
enum('visible', 'hidden')	No

borderBottomColor

Type	Required
color	No

borderBottomEndRadius

Type	Required
number	No

borderBottomLeftRadius

Type	Required
number	No

borderBottomRightRadius

Type	Required
number	No

borderBottomStartRadius

Type	Required
number	No

borderBottomWidth

Type	Required
number	No

borderColor

Type	Required
color	No

borderEndColor

Type	Required
color	No

borderLeftColor

Type	Required
color	No

borderLeftWidth

Type	Required
number	No

borderRadius

Type	Required
number	No

backgroundColor

Type	Required
color	No

borderRightWidth

Type	Required
number	No

borderStartColor

Type	Required
color	No

borderStyle

Type	Required
enum('solid', 'dotted', 'dashed')	No

borderTopColor

Type	Required
color	No

borderTopEndRadius

Type	Required
number	No

borderTopLeftRadius

Type	Required
number	No

borderTopRightRadius

Type	Required
number	No

borderTopStartRadius

Type	Required
number	No

borderTopWidth

Type	Required
number	No

borderWidth

Type	Required
number	No

opacity

Type	Required
number	No

elevation

(Android-only) Sets the elevation of a view, using Android's underlying [elevation API](#). This adds a drop shadow to the item and affects z-order for overlapping views. Only supported on Android 5.0+, has no effect on earlier versions.

Type	Required	Platform
number	No	Android

ImageBackground

A common feature request from developers familiar with the web is `background-image`. To handle this use case, you can use the `<ImageBackground>` component, which has the same props as `<Image>`, and add whatever children to it you would like to layer on top of it.

You might not want to use `<ImageBackground>` in some cases, since the implementation is very simple. Refer to `<ImageBackground>`'s [source code](#) for more insight, and create your own custom component when needed.

Note that you must specify some width and height style attributes.

Example

```
return (
  <ImageBackground source={...} style={{width: '100%', height: '100%'}}>
    <Text>Inside</Text>
  </ImageBackground>
);
```

Props

- `Image` props...
- `style`
- `imageStyle`
- `imageRef`

Reference

Props

style

Type	Required
view styles	No

imageStyle

Type	Required
image styles	No

imageRef

Allows to set a reference to the inner `Image` component

Type	Required
Ref	No

Native Modules Setup

Native modules are usually distributed as npm packages, except that on top of the usual Javascript they will include some native code per platform. To understand more about npm packages you may find [this guide](#) useful.

To get set up with the basic project structure for a native module we will use a third party tool [react-native-create-library](#). You can go ahead further and dive deep into how that library works, for our needs we will just need:

```
$ npm install -g react-native-create-library
$ react-native-create-library MyLibrary
```

Where `MyLibrary` is the name you would like for the new module. After doing this you will navigate into `MyLibrary` folder and install the npm package to be locally available for your computer by doing:

```
$ npm install
```

After this is done you can go to your main react app folder (which you created by doing `react-native init MyApp`)

- add your newly created module as a dependency in your package.json
- run `npm install` to bring it along from your local npm repository.

After this, you will be able to continue to native-modules-ios or native-module-android to add in some code. Make sure to read the README.md within your `MyLibrary` Directory for platform-specific instructions on how to include the project.

Out-of-Tree Platforms

React Native is not just for Android and iOS - there are community-supported projects that bring it to other platforms, such as:

- [React Native Windows](#) - React Native support for Microsoft's Universal Windows Platform (UWP) and the Windows Presentation Foundation (WPF)
- [React Native DOM](#) - An experimental, comprehensive port of React Native to the web. (Not to be confused with [React Native Web](#), which has different goals)
- [React Native Desktop](#) - A project aiming to bring React Native to the Desktop with Qt's QML. A fork of [React Native Ubuntu](#), which is no longer maintained.
- [React Native macOS](#) - An experimental React Native fork targeting macOS and Cocoa

Creating your own React Native platform

Right now the process of creating a React Native platform from scratch is not very well documented - one of the goals of the upcoming re-architecture ([Fabric](#)) is to make maintaining a platform easier.

Bundling

As of React Native 0.57 you can now register your React Native platform with React Native's JavaScript bundler, [Metro](#). This means you can pass `--platform example` to `react-native bundle`, and it will look for JavaScript files with the `.example.js` suffix.

To register your platform with RNPM, your module's name must match one of these patterns:

- `react-native-example` - It will search all top-level modules that start with `react-native-`
- `@org/react-native-example` - It will search for modules that start with `react-native-` under any scope
- `@react-native-example/module` - It will search in all modules under scopes with names starting with `@react-native-`

You must also have an entry in your `package.json` like this:

```
{  
  "rnpm": {  
    "haste": {  
      "providesModuleNodeModules": ["react-native-example"],  
      "platforms": ["example"]  
    }  
  }  
}
```

`"providesModuleNodeModules"` is an array of modules that will get added to the Haste module search path, and `"platforms"` is an array of platform suffixes that will be added as valid platforms.

