

# Communication between native and React Native

In [Integrating with Existing Apps guide](#) and [Native UI Components guide](#) we learn how to embed React Native in a native component and vice versa. When we mix native and React Native components, we'll eventually find a need to communicate between these two worlds. Some ways to achieve that have been already mentioned in other guides. This article summarizes available techniques.

## Introduction

React Native is inspired by React, so the basic idea of the information flow is similar. The flow in React is one-directional. We maintain a hierarchy of components, in which each component depends only on its parent and its own internal state. We do this with properties: data is passed from a parent to its children in a top-down manner. If an ancestor component relies on the state of its descendant, one should pass down a callback to be used by the descendant to update the ancestor.

The same concept applies to React Native. As long as we are building our application purely within the framework, we can drive our app with properties and callbacks. But, when we mix React Native and native components, we need some specific, cross-language mechanisms that would allow us to pass information between them.

## Properties

Properties are the most straightforward way of cross-component communication. So we need a way to pass properties both from native to React Native, and from React Native to native.

## Passing properties from native to React Native

In order to embed a React Native view in a native component, we use `RCTRootView`. `RCTRootView` is a `UIView` that holds a React Native app. It also provides an interface between native side and the hosted app.

`RCTRootView` has an initializer that allows you to pass arbitrary properties down to the React Native app. The `initialProperties` parameter has to be an instance of `NSDictionary`. The dictionary is internally converted into a JSON object that the top-level JS component can reference.

```
NSArray *imageList = @[@"http://foo.com/bar1.png",
                      @"http://foo.com/bar2.png"];

NSDictionary *props = @{@"images" : imageList};

RCTRootView *rootView = [[RCTRootView alloc] initWithBridge:bridge
                                           moduleName:@"ImageBrowserApp"
                                           initialProperties:props];
```

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

export default class ImageBrowserApp extends React.Component {
  renderImage(imgURI) {
    return <Image source={{uri: imgURI}} />;
  }
  render() {
    return <View>{this.props.images.map(this.renderImage)}</View>;
  }
}
```

`RCTRootView` also provides a read-write property `appProperties`. After `appProperties` is set, the React Native app is re-rendered with new properties. The update is only performed when the new updated properties differ from the previous ones.

```
NSArray *imageList = @[@"http://foo.com/bar3.png",
                      @"http://foo.com/bar4.png"];

rootView.appProperties = @{@"images" : imageList};
```

It is fine to update properties anytime. However, updates have to be performed on the main thread. You use the getter on any thread.

### NOTE

Currently, there is a known issue where setting `appProperties` during the bridge startup, the change can be lost. See <https://github.com/facebook/react-native/issues/20115> for more information.

There is no way to update only a few properties at a time. We suggest that you build it into your own wrapper instead.

## Passing properties from React Native to native

The problem exposing properties of native components is covered in detail in [this article](#). In short, export properties with `RCT_CUSTOM_VIEW_PROPERTY` macro in your custom native component, then use them in React Native as if the component was an ordinary React Native component.

## Limits of properties

The main drawback of cross-language properties is that they do not support callbacks, which would allow us to handle bottom-up data bindings. Imagine you have a small RN view that you want to be removed from the native parent view as a result of a JS action. There is no way to do that with props, as the information would need to go bottom-up.

Although we have a flavor of cross-language callbacks ([described here](#)), these callbacks are not always the thing we need. The main problem is that they are not intended to be passed as properties. Rather, this mechanism allows us to trigger a native action from JS, and handle the result of that action in JS.

## Other ways of cross-language interaction (events and native modules)

As stated in the previous chapter, using properties comes with some limitations. Sometimes properties are not enough to drive the logic of our app and we need a solution that gives more flexibility. This chapter covers other communication techniques available in React Native. They can be used for internal communication (between JS and native layers in RN) as well as for external communication (between RN and the 'pure native' part of your app).

React Native enables you to perform cross-language function calls. You can execute custom native code from JS and vice versa. Unfortunately, depending on the side we are working on, we achieve the same goal in different ways. For native - we use events mechanism to schedule an execution of a handler function in JS, while for React Native we directly call methods exported by native modules.

## Calling React Native functions from native (events)

Events are described in detail in [this article](#). Note that using events gives us no guarantees about execution time, as the event is handled on a separate thread.

Events are powerful, because they allow us to change React Native components without needing a reference to them. However, there are some pitfalls that you can fall into while using them:

- As events can be sent from anywhere, they can introduce spaghetti-style dependencies into your project.
- Events share namespace, which means that you may encounter some name collisions. Collisions will not be detected statically, which makes them hard to debug.
- If you use several instances of the same React Native component and you want to distinguish them from the perspective of your event, you'll likely need to introduce identifiers and pass them along with events (you can use the native view's `reactTag` as an identifier).

The common pattern we use when embedding native in React Native is to make the native component's `RCTViewManager` a delegate for the views, sending events back to JavaScript via the bridge. This keeps related event calls in one place.

## Calling native functions from React Native (native modules)

Native modules are Objective-C classes that are available in JS. Typically one instance of each module is created per JS bridge. They can export arbitrary functions and constants to React Native. They have been covered in detail in [this article](#).

The fact that native modules are singletons limits the mechanism in the context of embedding. Let's say we have a React Native component embedded in a native view and we want to update the native, parent view. Using the native module mechanism, we would export a function that not only takes expected arguments, but also an identifier of the parent native view. The identifier would be used to retrieve a reference to the parent view to update. That said, we would need to keep a mapping from identifiers to native views in the module.

Although this solution is complex, it is used in `RCTUIManager`, which is an internal React Native class that manages all React Native views.

Native modules can also be used to expose existing native libraries to JS. The [Geolocation library](#) is a living example of the idea.

### CAUTION

All native modules share the same namespace. Watch out for name collisions when creating new ones.

## Layout computation flow

When integrating native and React Native, we also need a way to consolidate two different layout systems. This section covers common layout problems and provides a brief description of mechanisms to address them.

## Layout of a native component embedded in React Native

This case is covered in [this article](#). To summarize, since all our native react views are subclasses of `UIView`, most style and size attributes will work like you would expect out of

the box.

## Layout of a React Native component embedded in native

### React Native content with fixed size

The general scenario is when we have a React Native app with a fixed size, which is known to the native side. In particular, a full-screen React Native view falls into this case. If we want a smaller root view, we can explicitly set `RCTRootView`'s frame.

For instance, to make an RN app 200 (logical) pixels high, and the hosting view's width wide, we could do:

#### SomeViewController.m

---

```
- (void)viewDidLoad
{
    [...]
    RCTRootView *rootView = [[RCTRootView alloc] initWithBridge:bridge
                                                         moduleName:appName
                                                         initialProperties:props];
    rootView.frame = CGRectMake(0, 0, self.view.width, 200);
    [self.view addSubview:rootView];
}
```

When we have a fixed size root view, we need to respect its bounds on the JS side. In other words, we need to ensure that the React Native content can be contained within the fixed-size root view. The easiest way to ensure this is to use Flexbox layout. If you use absolute positioning, and React components are visible outside the root view's bounds, you'll get overlap with native views, causing some features to behave unexpectedly. For instance, 'TouchableHighlight' will not highlight your touches outside the root view's bounds.

It's totally fine to update root view's size dynamically by re-setting its frame property. React Native will take care of the content's layout.

### React Native content with flexible size

In some cases we'd like to render content of initially unknown size. Let's say the size will be defined dynamically in JS. We have two solutions to this problem.

1. You can wrap your React Native view in a `ScrollView` component. This guarantees that your content will always be available and it won't overlap with native views.
2. React Native allows you to determine, in JS, the size of the RN app and provide it to the owner of the hosting `RCTRootView`. The owner is then responsible for re-laying out the subviews and keeping the UI consistent. We achieve this with `RCTRootView`'s flexibility modes.

`RCTRootView` supports 4 different size flexibility modes:

### `RCTRootView.h`

```
typedef NS_ENUM(NSInteger, RCTRootViewSizeFlexibility) {  
    RCTRootViewSizeFlexibilityNone = 0,  
    RCTRootViewSizeFlexibilityWidth,  
    RCTRootViewSizeFlexibilityHeight,  
    RCTRootViewSizeFlexibilityWidthAndHeight,  
};
```

`RCTRootViewSizeFlexibilityNone` is the default value, which makes a root view's size fixed (but it still can be updated with `setFrame:`). The other three modes allow us to track React Native content's size updates. For instance, setting mode to `RCTRootViewSizeFlexibilityHeight` will cause React Native to measure the content's height and pass that information back to `RCTRootView`'s delegate. An arbitrary action can be performed within the delegate, including setting the root view's frame, so the content fits. The delegate is called only when the size of the content has changed.

#### CAUTION

Making a dimension flexible in both JS and native leads to undefined behavior. For example - don't make a top-level React component's width flexible (with `flexbox`) while you're using `RCTRootViewSizeFlexibilityWidth` on the hosting `RCTRootView`.

Let's look at an example.

## FlexibleSizeExampleView.m

```
- (instancetype)initWithFrame:(CGRect)frame
{
    [...]

    _rootView = [[RCTRootView alloc] initWithBridge:bridge
        moduleName:@"FlexibilityExampleApp"
        initialProperties:@{ }];

    _rootView.delegate = self;
    _rootView.sizeFlexibility = RCTRootViewSizeFlexibilityHeight;
    _rootView.frame = CGRectMake(0, 0, self.frame.size.width, 0);
}

#pragma mark - RCTRootViewDelegate
- (void)rootViewDidChangeIntrinsicSize:(RCTRootView *)rootView
{
    CGRect newFrame = rootView.frame;
    newFrame.size = rootView.intrinsicContentSize;

    rootView.frame = newFrame;
}
```

In the example we have a `FlexibleSizeExampleView` view that holds a root view. We create the root view, initialize it and set the delegate. The delegate will handle size updates. Then, we set the root view's size flexibility to `RCTRootViewSizeFlexibilityHeight`, which means that `rootViewDidChangeIntrinsicSize:` method will be called every time the React Native content changes its height. Finally, we set the root view's width and position. Note that we set there height as well, but it has no effect as we made the height RN-dependent.

You can checkout full source code of the example [here](#).

It's fine to change root view's size flexibility mode dynamically. Changing flexibility mode of a root view will schedule a layout recalculation and the delegate `rootViewDidChangeIntrinsicSize:` method will be called once the content size is known.



**NOTE**




React Native layout calculation is performed on a separate thread, while native UI view updates are done on the main thread. This may cause temporary UI inconsistencies between native and React Native. This is a known problem and our team is working on synchronizing UI updates coming from different sources.

#### NOTE

React Native does not perform any layout calculations until the root view becomes a subview of some other views. If you want to hide React Native view until its dimensions are known, add the root view as a subview and make it initially hidden (use `UIView`'s `hidden` property). Then change its visibility in the delegate method.

Is this page useful?  

 Edit this page

Last updated on **Aug 29, 2023**