

Integration with Existing Apps

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

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

[Android \(Kotlin\)](#)[Android \(Java\)](#)[iOS \(Objective-C\)](#)[iOS \(Swift\)](#)

Key Concepts

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

1. Set up React Native dependencies and directory structure.
2. Understand what React Native components you will use in your app.
3. Add these components as dependencies using CocoaPods.
4. Develop your React Native components in JavaScript.
5. Add a `RCTRootView` to your iOS app. This view will serve as the container for your React Native component.
6. Start the React Native server and run your native application.
7. Verify that the React Native aspect of your application works as expected.

Prerequisites

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

1. Set up directory structure

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

2. Install JavaScript dependencies

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

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

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

npm **Yarn**

```
yarn add react-native
```

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

```
warning "react-native@0.52.2" has unmet peer dependency "react@16.2.0".
```

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

npm **Yarn**

```
yarn add react@version_printed_above
```

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

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

3. Install CocoaPods

CocoaPods is a package management tool for iOS and macOS development. We use it to add the actual React Native framework code locally into your current project.

We recommend installing CocoaPods using [Homebrew](#).

```
$ brew install cocoapods
```

It is technically possible not to use CocoaPods, but that would require manual library and linker additions that would overly complicate this process.

Adding React Native to your app

Assume the app for integration is a 2048 game. Here is what the main menu of the native application looks like without React Native.

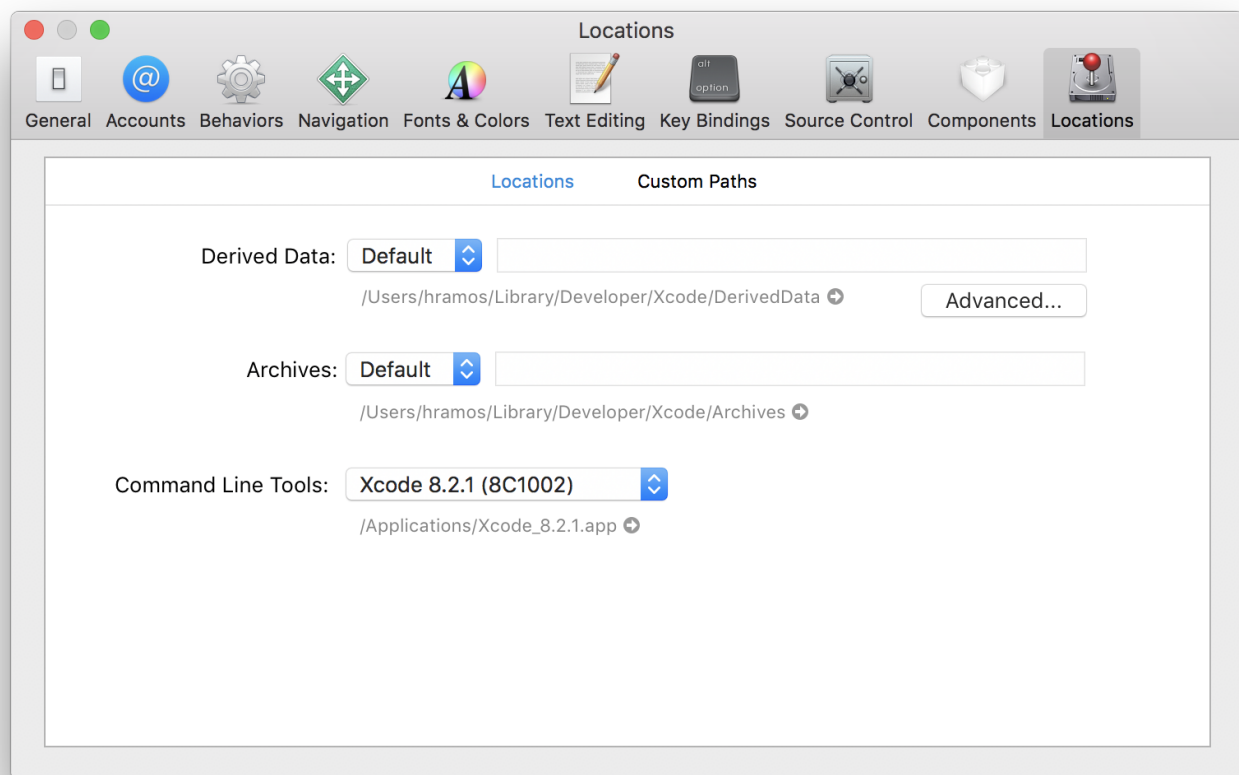


[Play Game](#)



Command Line Tools for Xcode

Install the Command Line Tools. Choose **Settings... (or Preferences...)** in the Xcode menu. Go to the Locations panel and install the tools by selecting the most recent version in the Command Line Tools dropdown.



Configuring CocoaPods dependencies

Before you integrate React Native into your application, you will want to decide what parts of the React Native framework you would like to integrate. We will use CocoaPods to specify which of these "subspecs" your app will depend on.

The list of supported `subspec`s is available in `/node_modules/react-native/React.podspec`. They are generally named by functionality. For example, you will generally always want the `core` `subspec`. That will get you the `AppRegistry`, `StyleSheet`, `View` and other core React Native libraries. If you want to add the React Native `Text` library (e.g., for `<Text>` elements), then you will need the `RCTText` `subspec`. If you want the `Image` library (e.g., for `<Image>` elements), then you will need the `RCTImage` `subspec`.

You can specify which subspecs your app will depend on in a Podfile file. The easiest way to create a Podfile is by running the CocoaPods `init` command in the `/ios` subfolder of your project:

```
$ pod init
```

The Podfile will contain a boilerplate setup that you will tweak for your integration purposes.

The Podfile version changes depending on your version of `react-native`. Refer to <https://react-native-community.github.io/upgrade-helper/> for the specific version of Podfile you should be using.

Ultimately, your Podfile should look something similar to this: [Podfile Template](#)

After you have created your Podfile, you are ready to install the React Native pod.

```
$ pod install
```

You should see output such as:

```
Analyzing dependencies
Fetching podspec for `React` from `../node_modules/react-native`
Downloading dependencies
Installing React (0.62.0)
Generating Pods project
Integrating client project
Sending stats
Pod installation complete! There are 3 dependencies from the Podfile and 1 total pod installed.
```

If this fails with errors mentioning `xcrun`, make sure that in Xcode in **Settings... (or Preferences...) > Locations** the Command Line Tools are assigned.

If you get a warning such as *"The swift-2048 [Debug] target overrides the FRAMEWORK_SEARCH_PATHS build setting defined in Pods/Target Support Files/Pods-swift-2048/Pods-swift-2048.debug.xcconfig. This can lead to problems with the CocoaPods installation"*, then make sure the Framework Search Paths in Build Settings for both Debug and Release only contain `$(inherited)`.

Code integration

Now we will actually modify the native iOS application to integrate React Native. For our 2048 sample app, we will add a "High Score" screen in React Native.

The React Native component

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

1. Create a `index.js` file

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

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

2. Add your React Native code

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

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

const RNHighScores = ({scores}) => {
  const contents = scores.map(score => (
    <Text key={score.name}>
      {score.name}:{score.value}
      {'\n'}
    </Text>
  ));
};
```

```

    return (
      <View style={styles.container}>
        <Text style={styles.highScoresTitle}>
          2048 High Scores!
        </Text>
        <Text style={styles.scores}>{contents}</Text>
      </View>
    );
  };

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#FFFFFF',
  },
  highScoresTitle: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
  scores: {
    textAlign: 'center',
    color: '#333333',
    marginBottom: 5,
  },
});

// Module name
AppRegistry.registerComponent('RNHighScores', () => RNHighScores);

```

RNHighScores is the name of your module that will be used when you add a view to React Native from within your iOS application.

The Magic: RCTRootView

Now that your React Native component is created via `index.js`, you need to add that component to a new or existing `ViewController`. The easiest path to take is to optionally create an event path to your component and then add that component to an existing `ViewController`.

We will tie our React Native component with a new native view in the `ViewController` that will actually contain it called `RCTRootView` .

1. Create an Event Path

You can add a new link on the main game menu to go to the "High Score" React Native page.



2. Event Handler

We will now add an event handler from the menu link. A method will be added to the main `ViewController` of your application. This is where `RCTRootView` comes into play.

When you build a React Native application, you use the Metro bundler to create an `index.bundle` that will be served by the React Native server. Inside `index.bundle` will be our `RNHighScore` module. So, we need to point our `RCTRootView` to the location of the `index.bundle` resource (via `NSURL`) and tie it to the module.

We will, for debugging purposes, log that the event handler was invoked. Then, we will create a string with the location of our React Native code that exists inside the `index.bundle`. Finally, we will create the main `RCTRootView`. Notice how we provide `RNHighScores` as the `moduleName` that we created above when writing the code for our React Native component.

First import the React library.

```
import React
```

The `initialProperties` are here for illustration purposes so we have some data for our high score screen. In our React Native component, we will use `this.props` to get access to that data.

```
@IBAction func highScoreButtonTapped(sender : UIButton) {
    NSLog("Hello")
    let jsCodeLocation = URL(string: "http://localhost:8081/index.bundle?platform=ios")
    let mockData:NSDictionary = ["scores":
        [
            ["name":"Alex", "value":"42"],
            ["name":"Joel", "value":"10"]
        ]
    ]

    let rootView = RCTRootView(
        bundleURL: jsCodeLocation,
        moduleName: "RNHighScores",
        initialProperties: mockData as [NSObject : AnyObject],
        launchOptions: nil
    )
}
```

```

let vc = UIViewController()
vc.view = rootView
self.present(vc, animated: true, completion: nil)
}

```

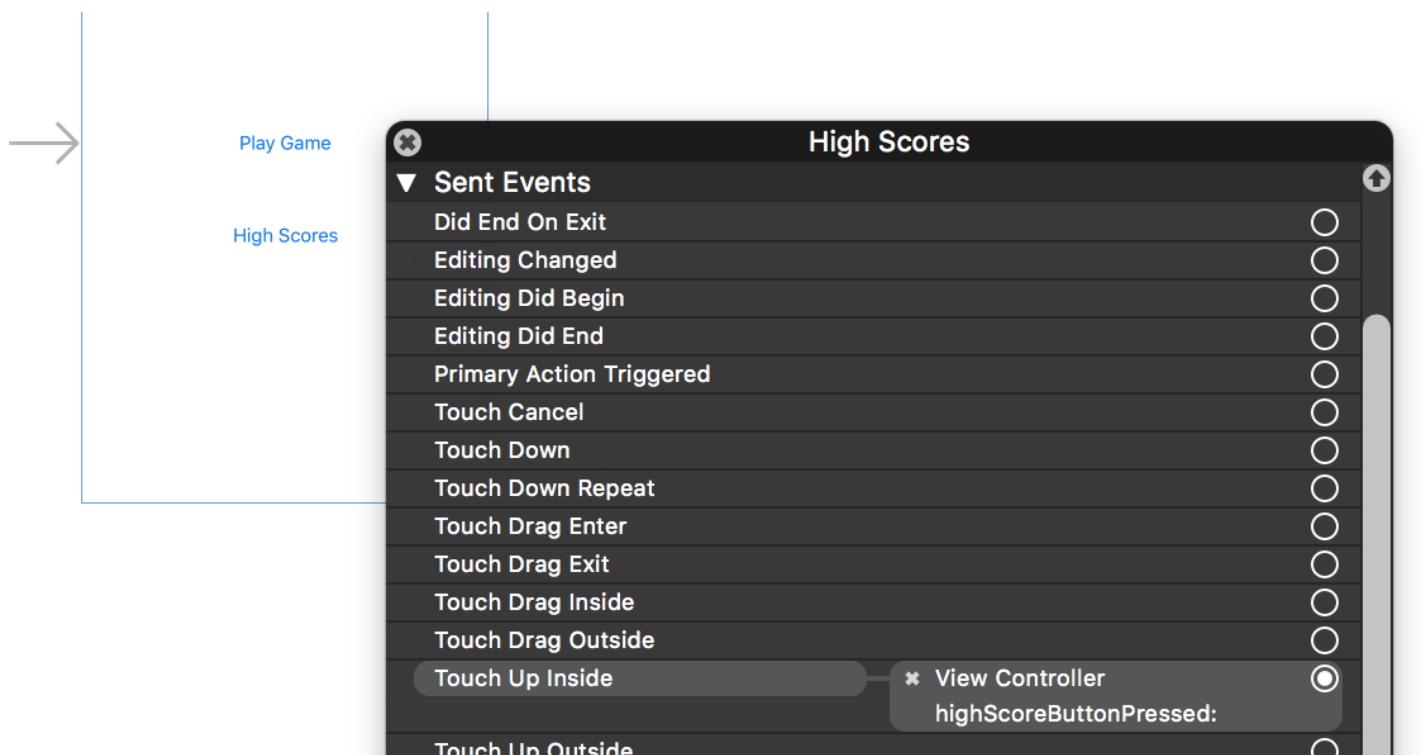
Note that `RCTRootView bundleURL` starts up a new JSC VM. To save resources and simplify the communication between RN views in different parts of your native app, you can have multiple views powered by React Native that are associated with a single JS runtime. To do that, instead of using `RCTRootView bundleURL`, use

`RCTBridge initWithBundleURL` to create a bridge and then use `RCTRootView initWithBridge`.

When moving your app to production, the `NSURL` can point to a pre-bundled file on disk via something like `let mainBundle = NSBundle(URLForResource: "main" withExtension:"jsbundle")`. You can use the `react-native-xcode.sh` script in `node_modules/react-native/scripts/` to generate that pre-bundled file.

3. Wire Up

Wire up the new link in the main menu to the newly added event handler method.



One of the easier ways to do this is to open the view in the storyboard and right click on the new link. Select something such as the `Touch Up Inside` event, drag that to the storyboard and then select the created method from the list provided.

3. Window Reference

Add an window reference to your `AppDelegate.swift` file. Ultimately, your `AppDelegate` should look something similar to this:

```
import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {

    // Add window reference
    var window: UIWindow?

    func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }

    ....
}
```

Test your integration

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

1. Add App Transport Security exception

Apple has blocked implicit cleartext HTTP resource loading. So we need to add the following our project's `Info.plist` (or equivalent) file.

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSExceptionDomains</key>
```

```
<dict>
  <key>localhost</key>
  <dict>
    <key>NSTemporaryExceptionAllowsInsecureHTTPLoads</key>
    <true/>
  </dict>
</dict>
</dict>
```

App Transport Security is good for your users. Make sure to re-enable it prior to releasing your app for production.

2. Run the packager

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

npm **Yarn**

```
yarn start
```

3. Run the app

If you are using Xcode or your favorite editor, build and run your native iOS application as normal. Alternatively, you can run the app from the command line using following command from the root directory of your React Native project:

npm **Yarn**

```
yarn ios
```

In our sample application, you should see the link to the "High Scores" and then when you click on that you will see the rendering of your React Native component.

Here is the *native* application home screen:



Carrier

8:21 PM

[Play Game](#)[High Scores](#)

Here is the *React Native* high score screen:



Carrier

8:21 PM



2048 High Scores!

Alex:42

Joel:10




If you are getting module resolution issues when running your application please see [this GitHub issue](#) for information and possible resolution. [This comment](#) seemed to be the latest possible resolution.

Now what?

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

Is this page useful?  

 Edit this page

Last updated on **Jun 21, 2023**