programming

React Native

Dotan Nahum

Programming React Native

Dotan Nahum

This book is for sale at http://leanpub.com/programming-react-native

This version was published on 2016-07-01

Contents

About Me

My name is Dotan Nahum, and my story with computing started somewhere around 95'.

While I had an Apple *IIe* since early childhood, I really started out at 14, in the then very vibrant demoscene[1]. I got to join a few groups and found one group, and most of my coding was in Assembly and C over crappy 14k internet connection, and an ISP ccount that I managed to share with someone. I spent my days building game mods (when it wasn't that popular like today) and nights debugging other people's painfully opaque assembly code using hard-core tools, like SoftICE; I guess this was what forged my personality as a hacker.

During the dotcom boom I did work in C, Perl, Python, VB, Java, to end up deeply in .NET in 2005, building a .NET based startup in 2007 and finally joining an enterprise to be the tech lead at around 2008. If you check, in a hindsight, that timeline of technologies matches each technology's tipping point (sounds unbelievable, but once, Java was very unusable).

From there, I moved to the Facebook-era web boom, and joined an Adtech company to serve as the infrastructure one-man-show. With the years, I joined a division that built a browser, formed a division that built a rich content recommendation-based Android lock-screen which competed with Facebook Home (we started way before Facebook Home), and 3 years after that, I moved to be the CTO of a company building a platform for DIY mobile apps: a sort-of Wix for native apps.

Throughout that time, I kept doing open-source[2]. I started at around 99', when most people didn't know what that was.

The reason I'm telling you all of this is that I spent all of my good years and free time, for better or worst, getting to be familiar with the high-level and low-level of an enormous set of technologies whether it is backend, desktop, or mobile.

Today, you find me buried in another stack of technologies: React and React Native. It's not the first cross-platform silver-bullet mobile development stack I've tried either. The mobile platform I've talked about being a CTO of? That's built on Cordova, and later Native iOS and Native Android, and it generated around 2 million mobile apps to date. I've tried Xamarin, Appcelerator, and Ruby Motion. I've also tried CrossWalk and Cordova. I've also used Cocos-2dx for gaming. And Unity.

[1] https://en.wikipedia.org/wiki/Demoscene

[2] https://github.com/jondot

You want to know what was the general theme of these all? Simply, one of these three:

1. Bad performance
2. Bad API, docs, and/or support
3. Mismatch and friction against upstream mobile SDK (Android and iOS)
4. Broken or stale samples, APIs

Off the bat, React Native in its 0.19 version, and less than a year of being cross-platform (i.e. when React Native for Android came about), already does better on all of these accounts.

About This Book

Writing a book about a bleeding-edge technology is difficult. Some one said: “If you build with bleeding edge, expect there to be some blood spilt”. While I wouldn’t go that far, I would say this:

- React Native updated 6 versions while writing this book (several months)
- The first version of React Native had a completely different project structure, and the book samples looked completely different
- APIs were broken each time and again, and the samples got rewritten until they felt a little dirty

At first, this book looked completely different. It used to look like this:

- Introduction
- React principles
- React Native getting started
- A step-by-step build out of the showcase app
- Advanced topics

I soon found out that this structure is highly unreliable, since changes in React Native and the Javascript ecosystem doomed it to change very frequently.

How To Build a Book for Bleeding Edge?

Well, this is what I did, eventually.

1. Do away with step-by-step and assume readers are coming with experience and have a hacker mentality. From my experience writing this book, I can definitely say that the beginner “walkthrough” and “learning” type books that are out there right now, by now, are probably worth nothing (unless they get updated every two weeks).
2. Throw away every Javascript component and library that didn’t directly serve the purpose of React Native out the door. This meant:
3. Testing doesn’t appear in this book (at the time of writing the way to test is still being invented. Currently people use Mocha, a custom React Native mock components, and Enzyme)

4. A Flux framework is not included, however the codebase clearly resembles a flavor-less Flux
5. Focus on: *why*, *context*, *how*, and *patterns*. Focus on things that will remain, while the bleeding edge dust settle even a year from now.
6. Invest more time in advanced, low-level topics, because these will change less frequently than user-facing API surface-area.

How To Read This Book?

With that out of the way, you can read this book cover to cover, or you can skim it, or you can just read the advanced chapters and rely on the various up-to-date tutorials out there that go over the basics of building a React Native. I maintain the Awesome-React-Native[3] list which hold many of these, so that you can just go that and pick a tutorial or a sample app you like.

[3] https://github.com/jondot/awesome-react-native

Introduction

About Me

The Revolution Has Begun

Cross Platform Mobile

React

React Native

Why This Time?

Book Style

Intended Audience

What This Book Is

What This Book Is Not

Breezing Through React

Getting Started

Project Layout

React Native is a cross-platform SDK for mobile development. As such, you'll need to know how a project that the React Native CLI generates looks like in terms of key files, folders and so on; but you'll undoubtedly also need to get to know the native sides specific to each platform; iOS and Android, and you can be certain that each of these are completely different in terms of build tooling, dependency management, resources management and project conventions.

We'll try to cover *all* of these here in a sufficient way that will carry you through your future React Native projects. My advise to you, as a React Native programmer is of acceptance. For example, if you are originally an iOS developer, be tolerant towards the Android development environment, tooling and project structure, and the other way around. It's not easy, because some things are built better than others and you might have grown accustomed to using the better things your entire career.

The React Native Project

First, take a moment to set up your environment according to the latest best practices[4] on the official React Native documentation site. That will walk you through setting up Node.js, the Android SDK, watchman, flow, and additional tools that make the React Native stack great for development. Next, to start the discussion, let's generate a React Native project that we'll eventually throw away:

```
1 $ react-native init projectlayout
```

Wait a bit, and we'll get the following superstructure:

[4] https://facebook.github.io/react-native/docs/getting-started.html

```
.
├── android/
├── ios/
├── index.android.js
├── index.ios.js
└── package.json
```

On the first level of the project tree there is no complexity at all, and I think this is great. React Native does not force any convention or project structure and only exposes the *bare* essentials such as our `package.json` which is used for dependency and project managment via `npm`, and the two starting points for iOS and Android - `index.ios.js` and `index.android.js`. Neatly besides these an `android/` and `ios/` folders are placed and we'll touch these very shortly.

To read up on Javascript dependency management with `npm` look at this[5] and or a more friendly getting started[6] guide.

Both `index.ios.js` and `index.android.js` are *hardwired* into each native project's bootstrap code, and each platform knows how to filter its own platform-specific code by the `.ios` or `.android` infixes. In other words, a file tagged with `file.ios.js` will never be visible to Android when importing `file` and the other way around for the `.android` infix.

Now would be a good time to read more about Platform Specific Code[7] at the official documentation site.

Where possible, I will refer you to the official documentation. There is no point re-articulating something that is already perfectly and freely written elsewhere. Being an indy book allows me to make the content here to be just about the subject and avoid resorting to page-inflating tactics the big publishers use.

Running Your Project

By now, you can run both Android and iOS projects from the command line:

```
$ react-native run-ios
```

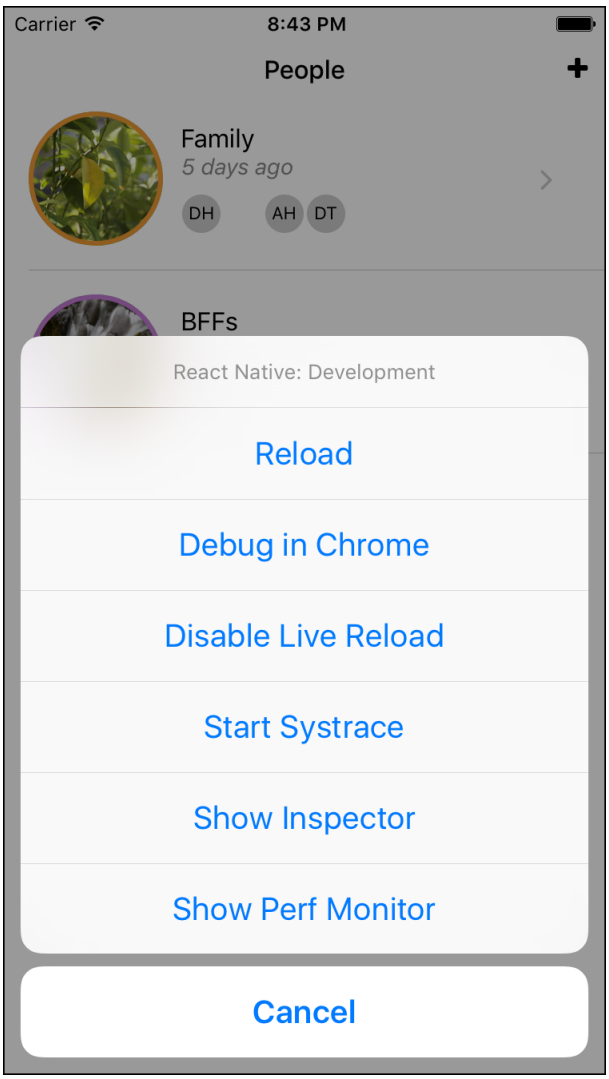
For Android, make sure your `ANDROID_HOME` environment variable is already set for the session or the user and run like so:

[5] https://docs.npmjs.com/files/package.json
[6] https://docs.npmjs.com/getting-started/using-a-package.json
[7] https://facebook.github.io/react-native/docs/platform-specific-code.html#content

```
$ export ANDROID_HOME=<your-android-home-path>
$ react-native run-android
```

To get a hold of your Android home path, you might need to recall where you installed the Android SDK distribution to if you did it manually, or if you are using Android Studio take a look at `File->Project Structure->Android SDK`.

Next up, you should be able to change Javascript code and reload easily with `ctrl-cmd-Z` (shake gesture) on iOS simulator which opens the debug action sheet, or the menu button on Android which does the same. For now, also feel free to explore the rest of this menu, we'll get back to it further along.



The Development Menu

If you've noticed, you didn't compile or run any native code, and if you played with the "Live Reload" option (you should), then you get to see your app automatically reflect changes in your code once you hit "Save" in your editor. This magic happens because in development mode React Native

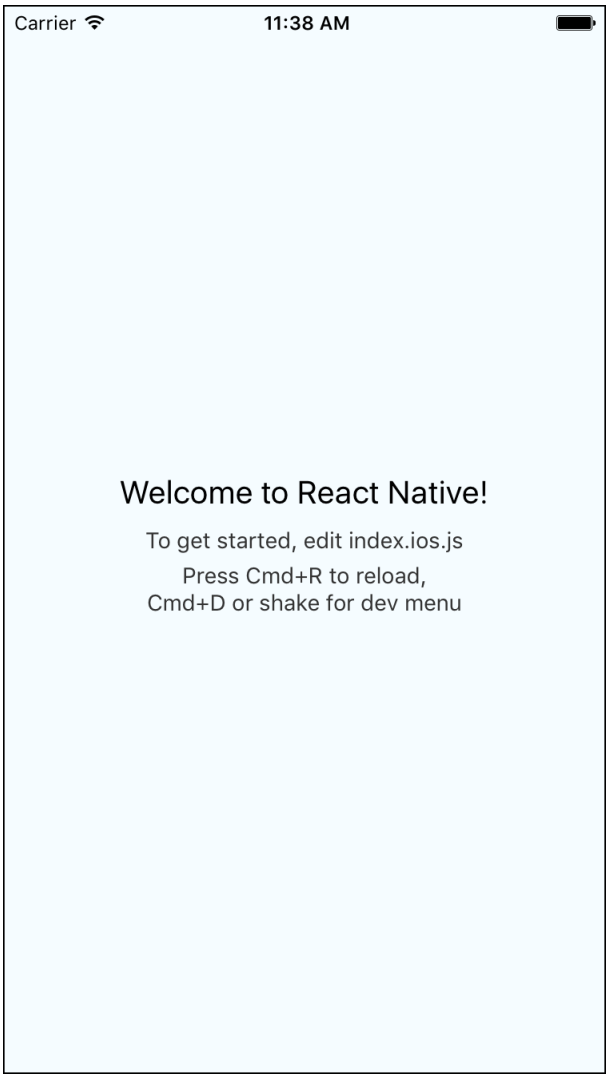
communicates with a small packager server and is working in an "RPC[8]" mode. This server that you might have noticed popping up uninvited every time you run your project (don't kill it!) watches your source files, and communicates changes back to the native app that runs on your device or simulator.

Making Changes

Now let's fiddle a bit with the default generated project. Run your iOS simulator like so:

```
1 $ react-native run-ios
```

This will take a bit, and you'll see the iOS simulator fire up and a terminal with the React Native packager pop up as well. You should see the following:



[8] https://en.wikipedia.org/wiki/Remote_procedure_call

Now edit `index.ios.js`, locate the main component called `projectlayout` and edit it so that you end up with this, essentially removing a bunch of components:

```
class projectlayout extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to React Native!
        </Text>
      </View>
    );
  }
}
```

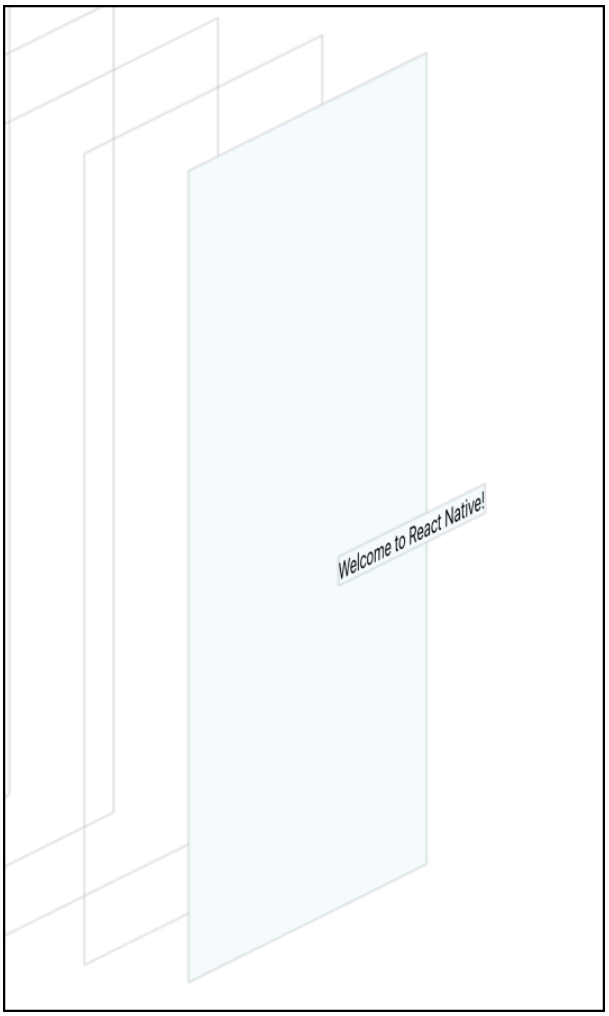
Hit `Cmd-R` when you're facing the iOS simulator to reload (or, even better hit `Ctrl-Cmd-Z` and pick "Enable Live Reload"). When content reloaded, we'll end up with this:

Let’s see how this looks like from the iOS platform’s point of view. Is this really native?

Go to your terminal and at your project working directory do this:

```
1 open ios/projectlayout.xcodeproj
```

Xcode will open up, now hit `Cmd-R` to run the same project via Xcode. After the app runs on the iOS simulator, go back to Xcode and through the main menu hit `Debug -> View Debugging -> Capture View Hierarchy`. After a couple seconds you’ll end up with a 3D model of the view hierarchy, grab and tilt it, and this is what we get:



Looks amazingly clean. From my experience there are plenty of chances for a cross platform native SDK to really trash the view hierarchy:

- View abstractions made to make life easier for platform developers - in this case you get a tree with meaningless wrapper and holder components which the platform developers use to track the tree itself. Doesn’t exist with React Native, and you get a raw, flat, tree which is very important for performance.
- Logical abstractions made to make life easier for platform documenters - in this case the platform makers choose to build their own custom components and that you use these exclusively because it gives them more control over how the platform performs at-large and also easier documentation story. React Native doesn’t attempt to solve that and gives you access to a limited set of controls and each platform’s bare native components.
- Tree hierarchy errors - simply because such platform may generate native code from your abstract Javascript code, it may do so without much intelligence and generate huge obfuscated view trees. Again -

> React Native doesn't do that , more over it uses React to help compute and render trees and diffs.

You'll find that I'll talk a lot about the *X-factor* that holds React Native apart from every cross-platform mobile SDK that we've seen so far - that's because I've used quite a few of those and the scars still show. React Native makes a real difference and we've just seen another one of these factors.

Bundling Your App

Not relevant for the immediate future, but very relevant for when you want to ship your application, you don't really want to rely on a local server when you run on a real device on a user's home network. Because of that, React Native provides a command line utility to *bundle* your app for production (or, offline) work:

```
$react-native bundle
Options:
  --entry-file        Path to the root JS file, either absolute or relative t\
o JS root                                   [required]
  --platform          Either "ios" or "android"
  --transformer       Specify a custom transformer to be used (absolute path)\
                                            [default: "/Users/Dotan/projects/\
programming-react-native-samples/projectlayout/node_modules/react-native/pack\
ager/transformer.js"]
  --dev               If false, warnings are disabled and the bundle is minif\
ied                                         [default: true]
  --prepack           If true, the output bundle will use the Prepack format.\
                                            [default: false]
  --bridge-config     File name of a a JSON export of __fbBatchedBridgeConfig\
. Used by Prepack. Ex. ./bridgeconfig.json
  --bundle-output     File name where to store the resulting bundle, ex. /tmp\
/groups.bundle                              [required]
  --bundle-encoding   Encoding the bundle should be written in (https://nodej\
s.org/api/buffer.html#buffer_buffer).       [default: "utf8"]
  --sourcemap-output  File name where to store the sourcemap file for resulti\
ng bundle, ex. /tmp/groups.map
  --assets-dest       Directory name where to store assets referenced in the \
bundle
  --verbose           Enables logging                                       \
                                            [default: false]
```

You will use that tool to create your `main.jsbundle` which will be packaged with your application binaries (`ipa` on iOS and `apk` on Android), so for iOS for example - provide the following arguments: `--platform ios`, `--entry-file index.ios.js`, and `--bundle-output ios/main.jsbundle`. Like so:

```
$ react-native bundle --entry-file index.ios.js --platform ios --bundle-outpu\
t ios/main.jsbundle
bundle: Created ReactPackager
bundle: start
bundle: finish
bundle: Writing bundle output to: ios/main.jsbundle
bundle: Closing client
Assets destination folder is not set, skipping...
bundle: Done writing bundle output
```

The same process holds for Android (of course, replace `ios` with `android` where appropriate and make sure the bundle output goes into the Android project).

Needless to say, running from the command line is very convenient and creates a “zen” flow while developing, since not only you’re not bothered with the slowness of handling a GUI (pointing and clicking) but you can also automate command line tasks. However, there comes a time when you’ll need some more fire power, to debug the native code, add your own native code, or set up the application shell, resources or permissions. For all of these, you’ll need to at least get to know each of your individual iOS or Android projects and tooling.

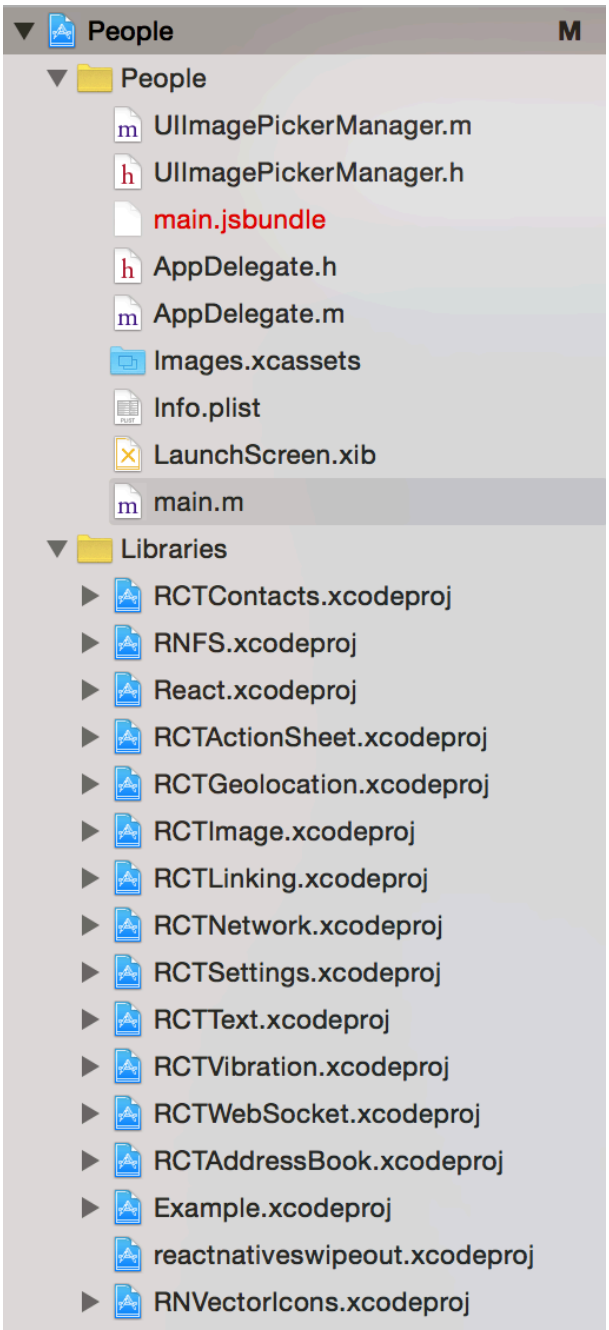
The Native iOS Project

Below is a dump of the tree that our `react-native init` command generated for the iOS part of our project.

```
├── ios
│   ├── projectlayout
│   │   ├── AppDelegate.h
│   │   ├── AppDelegate.m
│   │   ├── Base.lproj
│   │   │   └── LaunchScreen.xib
│   │   ├── Images.xcassets
│   │   │   └── AppIcon.appiconset
│   │   │       └── Contents.json
│   │   ├── Info.plist
│   │   └── main.m
│   ├── projectlayout.xcodeproj
│   │   ├── project.pbxproj
│   │   └── xcshareddata
│   │       └── xcschemes
│   │           └── projectlayout.xcscheme
│   └── projectlayoutTests
```

```
18 |       ├── Info.plist
19 |       └── projectlayoutTests.m
```

Our main touch points will be `AppDelegate.m` for bootstrapping code (in case we want to customize our loading mechanism, more on that later), various `plist`s and our `xcassets` for assets. As expected from a cross-platform framework like React Native, there's not much native surface area to handle.



Xcode Project

The Xcode project layout does not map directly to the directory structure

we just reviewd. Xcode keeps a layer of metadata for that. As you can see there's the `Libraries` group which does not exist physically, but does exist in the project layout within Xcode. `Libraries` is where you will add new libraries from community (or own) projects that you want to use that have a native part to them (we'll see how to do this in the *Using Community Components* subtopic), and `main.jsbundle` is a file that holds all of your "compiled" application code statically when you want to ship your app to production.

Let's take a detour and view the bootstrapping code iOS uses in order to load the actual Javascript application.

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOption\
s:(NSDictionary *)launchOptions
{
  NSURL *jsCodeLocation;
  jsCodeLocation = [NSURL URLWithString:@"http://localhost:8081/index.ios.bun\
dle?platform=ios&dev=true"];
//   jsCodeLocation = [[NSBundle mainBundle] URLForResource:@"main" withExten\
sion:@"jsbundle"];

  RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:jsCodeLocati\
on
                                                      moduleName:@"projectlay\
out"
                                               initialProperties:nil
                                                   launchOptions:launchOption\
s];

  self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
  UIViewController *rootViewController = [UIViewController new];
  rootViewController.view = rootView;
  self.window.rootViewController = rootViewController;
  [self.window makeKeyAndVisible];
  return YES;
}
@end
```

I've removed the very helpful comments from this file to save pages, so please do find the time to read it verbatim from your own source tree. This is a general application events module, designed to handle application centric events such as `didFinishLaunchingWithOptions` here, and events from notifications, background events and so on. React Native plugs into `didFinishLaunchingWithOptions` and configures the main `window` and `rootView`.

The important part is the `jsCodeLocation` resolution. In our case, it is rigged to an IP address, pointing to and assuming this is your local development

machine. In other instances this will be the actual bundle file we discussed before, and if you like to keep things tidy you'll want to make a compilation target, environment variable, or pragma to switch between the two (such as "development" and "release"). For now, keep it as is.

Other than that, if you're a stranger to iOS, I recommend taking a day to get to know Xcode[9]. As far as I know, if you factor the amount of code and apps that has been generated on that platform, then it is an IDE that packs so much "air time" that it is probably only second to Visual Studio.

Android is a completely different beast. Let's take a look at that now.

The Native Android Project

Here's the listing for our generated React Native project, focusing on the `android/` directory.

```
├── android
│   ├── app
│   │   ├── build.gradle
│   │   ├── proguard-rules.pro
│   │   ├── react.gradle
│   │   └── src
│   │       └── main
│   │           ├── AndroidManifest.xml
│   │           └── java
│   │               └── com
│   │                   └── projectlayout
│   │                       └── MainActivity.java
│   ├── build.gradle
│   ├── gradle
│   │   └── wrapper
│   │       ├── gradle-wrapper.jar
│   │       └── gradle-wrapper.properties
│   ├── gradle.properties
│   ├── gradlew
│   ├── gradlew.bat
│   └── settings.gradle
```

Android made the move from Eclipse and Ant a year or two ago, into the new and shiny Android Studio based on the Intellij platform, and less shiny (but very powerful) Gradle project. If you're coming from Java, then Gradle replaces Ant, Maven, and other different Java dependency management solutions. It provides endless flexibility and power in terms of dependency

[9] http://www.raywenderlich.com/tutorials

management and project configuration, while exposing a fluffy DSL (with the help of Groovy, the JVM based language).

This is why there's as many gradle files as normal source files in your generated project directory (is that a good or a bad thing, then? :-). Take a moment to explore the contents of `settings.gradle`, `app/build.gradle` and `app/react.gradle` as these are the key files you'll be handling during development with anything that regards dependency management - in other words, through these we'll configure external libraries, components and custom project compilation tasks and settings.

To make sense of it, remember these points:

- `settings.gradle` - holds pointers to various subprojects we'll want to configure. Plainly, this mostly will be external components with native parts; we'll point to their `node_modules` path so that Gradle will consider their Java projects as part of the general build. This in addition to definitions within `app/build.gradle` will resolve any "symbol undefined" errors.
- `app/build.gradle` - this is where we will specify our app's dependencies; external Jars we'd like to use (in case we're developing native parts), libraries and components from the community and so on. We'll also want to configure Android specific items there such as the SDK version, compilation target and so on. This only makes sense if you already are familiar with the Android platform. If you're not, please, spend a day building a hello-world app[10] with native Android.
- `app/react.gradle` - Gradle lets you run custom tasks of your own. It will show them through the command line. If you `$ cd android && ./gradlew tasks` you'll be able to take a look at every task available to you and within those tasks there will be `react` tasks. The `react.gradle` file is what defines them; as such you can also tweak (but probably will never need to) what's inside: namely the `bundleDebugJsAndAssets` and `bundleReleaseJsAndAssets` tasks.

Our `MainActivity.java` file is the parallel to iOS's `AppDelegate.m` file, which is nice to see; it is elegant. Let's take a look:

[10] https://developer.android.com/training/basics/firstapp/index.html

```
public class MainActivity extends ReactActivity {
    @Override
    protected String getMainComponentName() {
        return "projectlayout";
    }

    @Override
    protected boolean getUseDeveloperSupport() {
        return BuildConfig.DEBUG;
    }

    @Override
    protected List<ReactPackage> getPackages() {
      return Arrays.<ReactPackage>asList(
        new MainReactPackage()
      );
    }
}
```

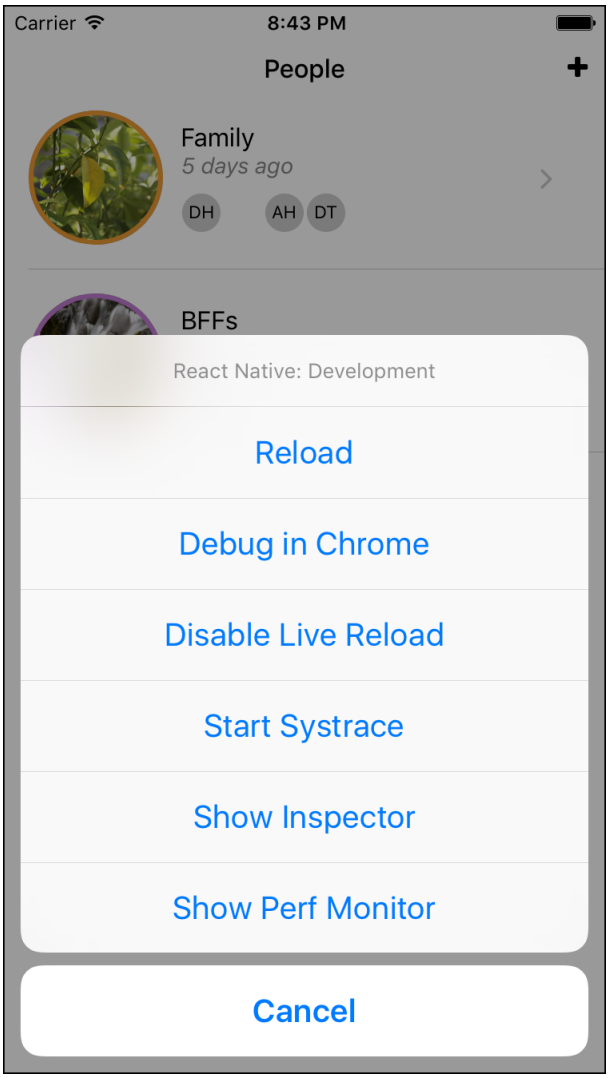
Removing comments, we don't see much. Most of the work revolves around `getPackages` where you'll be often required to add custom components' own packages (in case they have a native part). React Native's native extension model and infrastructure is fascinating so don't worry - we deep dive into this and into building your own native components (UI and modules) later, so hold on tight.

Tooling

We went over the React Native CLI, and it covers the run and build phases of the typical development cycle very well. In addition since we're really handling three different codebases: Javascript, and hopefully very little Objective-C, and Java, we would be wise to have a look at the various debugging and tooling around those.

Simulator and Emulator Developer Tools

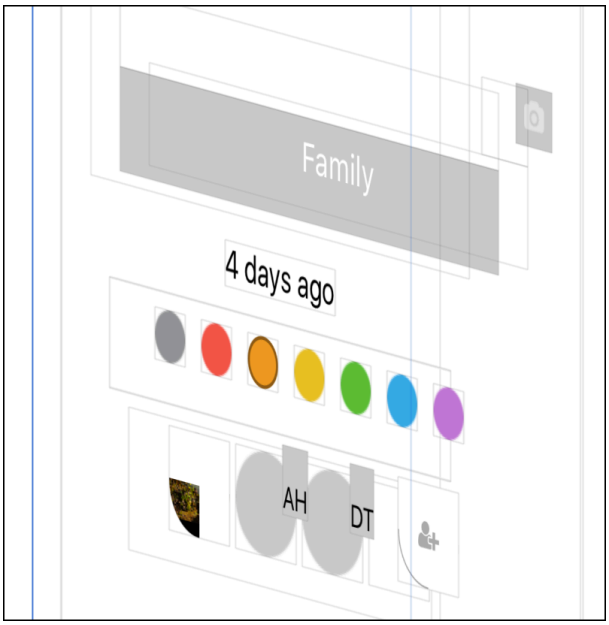
Since its early days React Native provided an unmatched developer experience on mobile if you factor the fact that it was such a young project. Error messages that are so helpful it feels telepathic, built-in developer tools, debugging with Chrome, and inspector and profiling tooling right there in your device; if you happened to deal with performance problems before, then you know having the profiling tools right there in your app and in your device is priceless (external tools may introduce overhead, and obviously trying to measure performance on a simulator is a no-go).



Built-in Debug Tooling

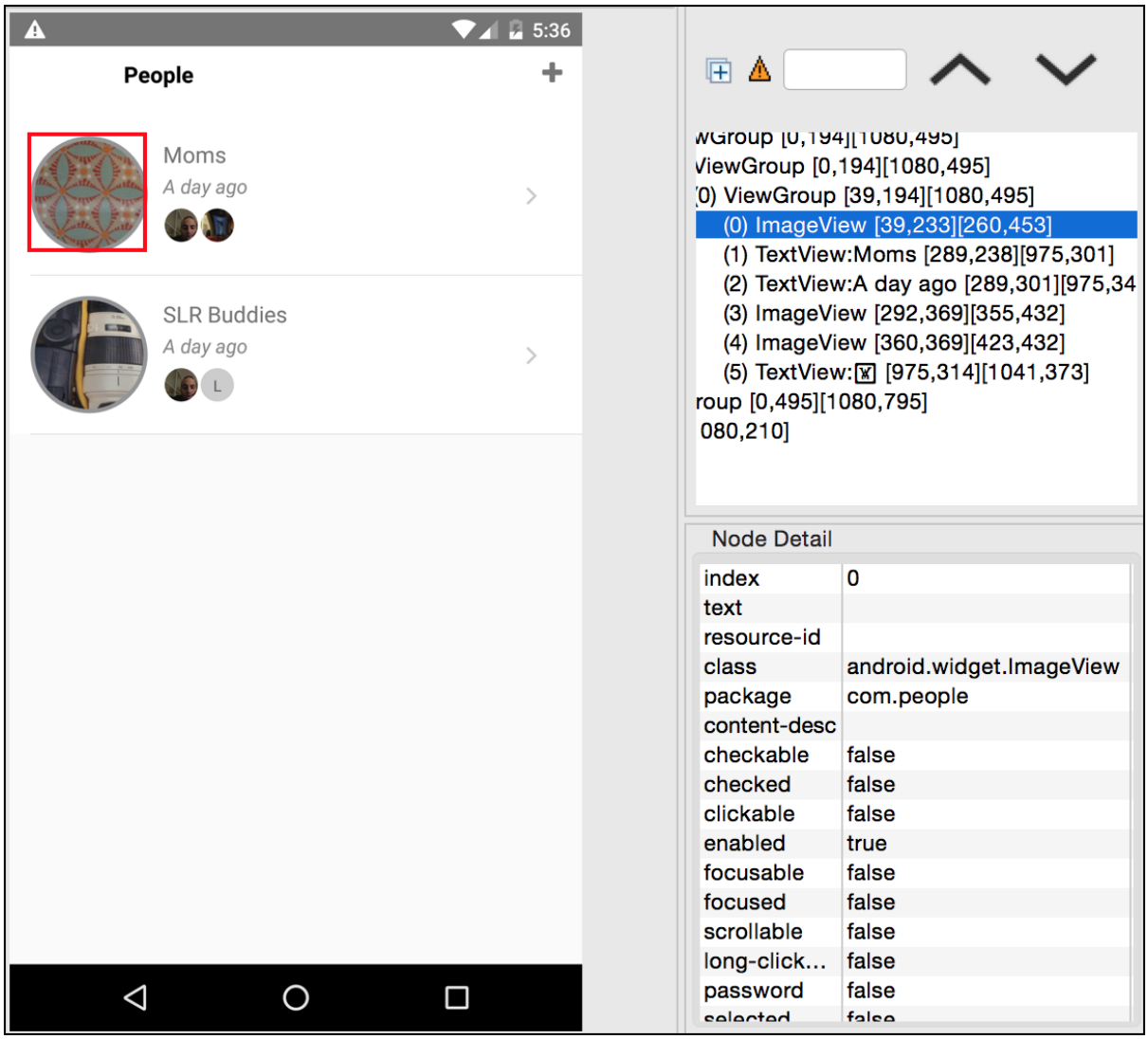
The inspector tool lets you hit-test live UI areas and see a breakdown of their properties and hierarchy, just like the Chrome inspection tool (go in your Chrome browser and hit `Shift-Cmd-C` and then try clicking inside a website somewhere).

On iOS, you can combine that with Xcode amazing visual debugging tools (while the app is running through Xcode `Cmd-R`, go to `Debug -> View Debugging -> Capture View Hierarchy`), you can dump the native visual tree like this:



Visual Breakdown of Components

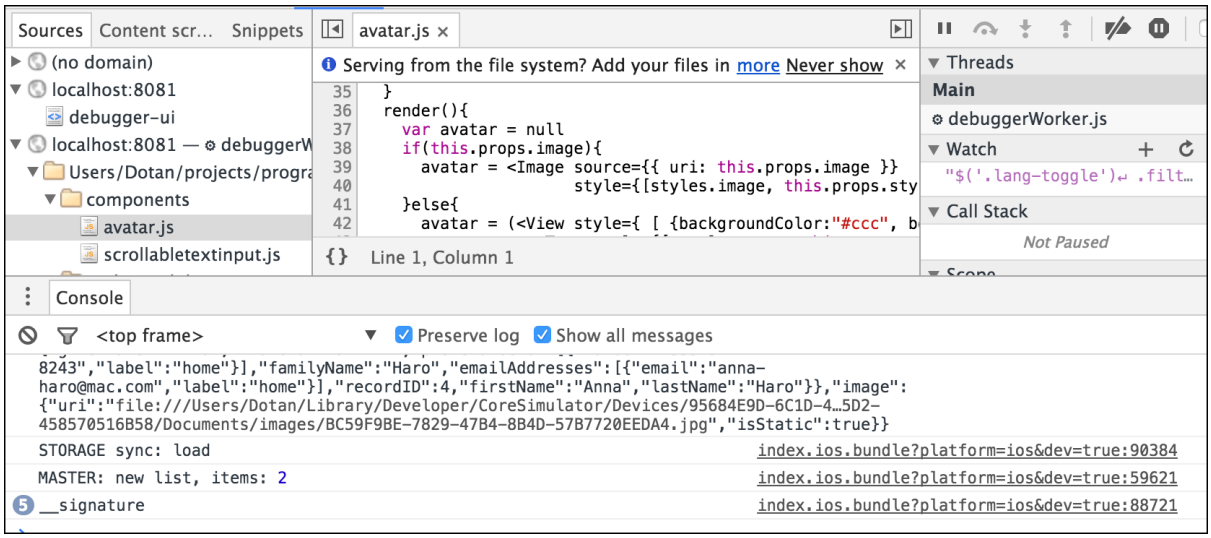
And on Android, you can combine that with DDMS (Android's hard core tooling for performance, debugging, analysis and so on), by selecting the device in the devices menu and clicking on the icon which looks like a stack of mobile devices (don't ask me why…):



Android's DDMS UI Layout Dump

This technique lets you perform a vice maneuver where you can attack a visual problem both on the Native and Javascript side and use each to complement the other for missing information.

Finally, we can use Chrome as a debugger for all of our React Native Javascript code:



Chrome Developer Tools

This means taking a look at log messages, setting breakpoints in actual React code, watches, stepping through and more. There's even a way to dump a trace with systrace and Google's profiling visualization tools but let's leave the spooky stuff for a different kind of book or blog post, chances are you won't need that kind of low level performance optimization help :)

We didn't talk about the simplest thing possible to do - logging. In real life you'll be using `console.log` a lot, and the output is watchable through Xcode itself while it runs the application, or in the case of Android, either with Android Studio (the Logcat widget window) or through the command line with `adb`, the Android super tool (really, `adb` is really useful to get to know, especially if you're an Android user yourself; you can automate a lot of your daily tasks with it). Run this on a spare terminal window:

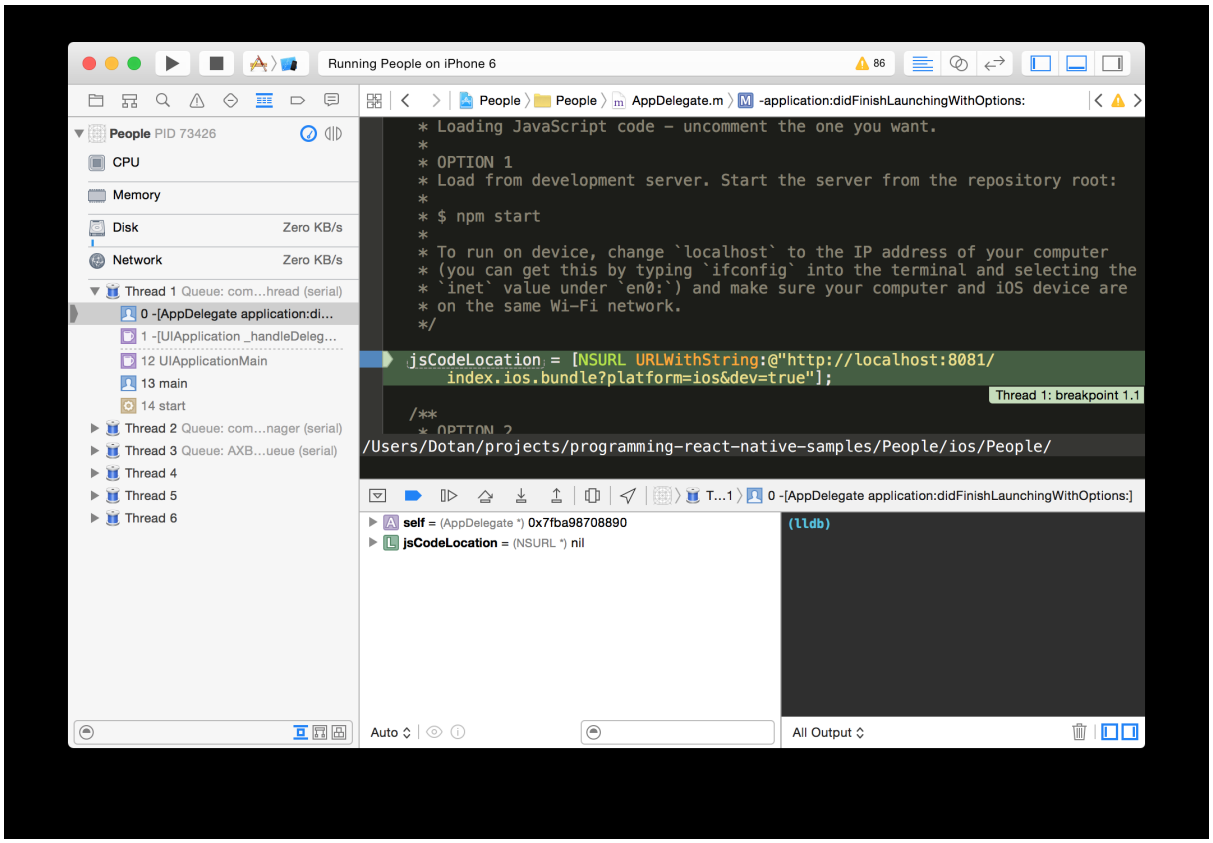
```
1 $ANDROID_HOME/platform-tools/adb logcat *:S ReactNative:V ReactNativeJS:V
```

If you dislike both for viewing live logs, we've just learned that you can use Chrome for debugging and also viewing logs - try that instead.

Native Debugging

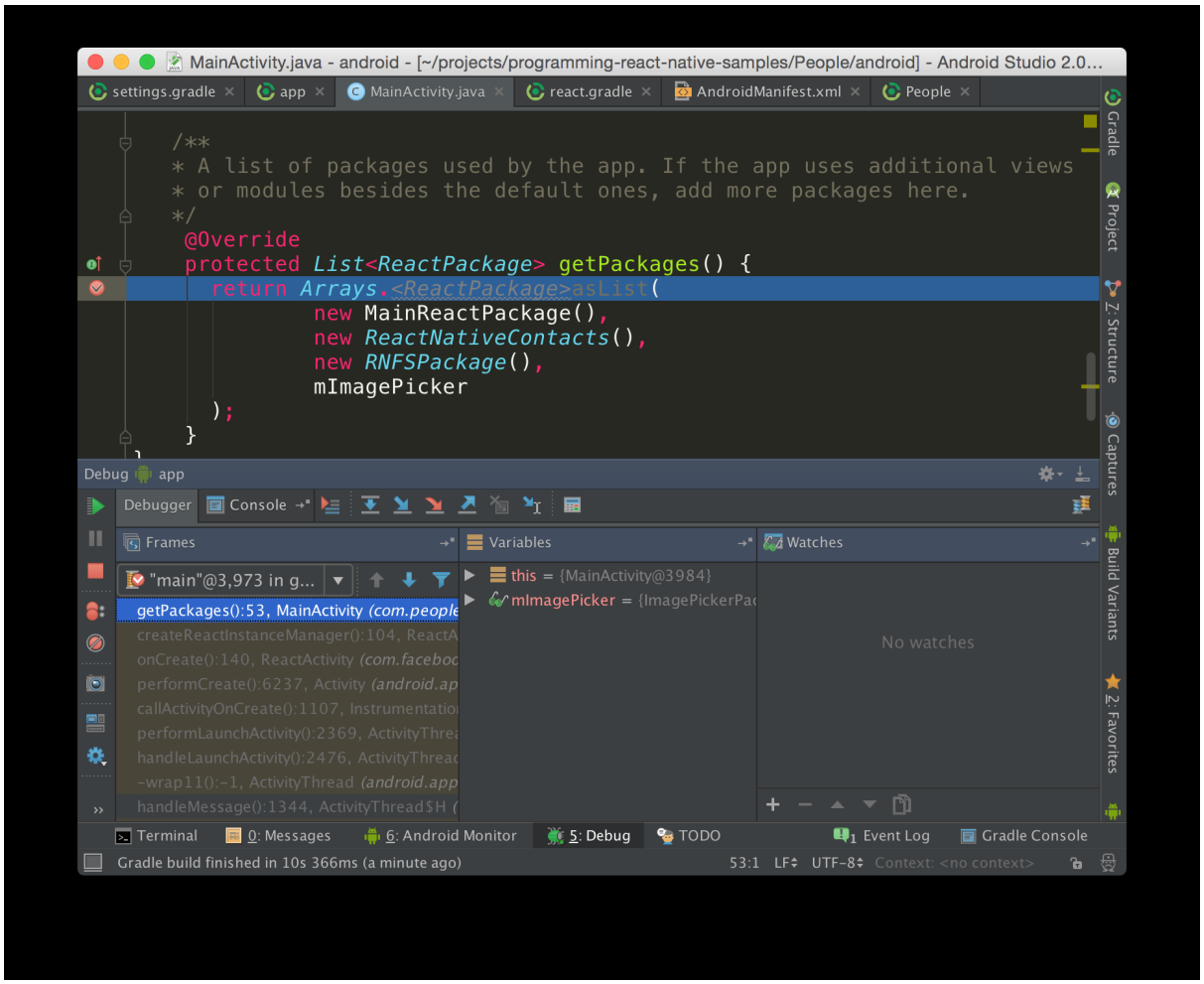
There were a time where I would cringe before covering such a topic. Native debugging on a cross-platform mobile SDK. However, this is not such a time, and the React Native team has done such a great job that all I have to say is this: if you're running your app and you use a native API through Javascript, and you ran the app via the IDE (Xcode or Android Studio) by simply hitting "Run", and you set a breakpoint on any native code - it will break into that code, and it will make total sense (i.e. local variables, contexts, threads, etc.). Kudos!

Here's how it looks on Xcode:



Debugging with Xcode

And its Android counterpart, Android Studio:



Debugging with Android Studio

So, if you recall back when we discussed the `run-android` and `run-ios` subcommands on the `react-native` CLI, this means that there *still is* great value for using the IDE behind each mobile platform, so don't give up on it so quickly.

Summary

In this chapter we covered the React Native project layout, built in tooling, the native platforms tooling and how it all fits. We also discussed some techniques you can use to improve your development experience, in addition to what tooling is available to do that.

All in all, the best thing for you to do now is to `init` a couple more projects, trash them down by tweaking with everything that looks tweakable, run them via Xcode or Android Studio, set breakpoints, and explore these great IDEs. Really, I can't remember a time where IDEs were this great – and I'm a Vim user!. So, feel free to have a go at it and we'll meet back here for more.

Building React Native Components

In this chapter, we'll go through a short history of Javascript frameworks, highlight how React is different, introduce React Native Katas - a playful learning experience for React Native I've created, and follow along one of the Katas.

A Squashed History of Javascript Frameworks

React changed how we build Javascript apps. First, there was jQuery which abstracted out how we work against a browser. jQuery made sure we were able to manipulate DOM elements without worrying about the entire DOM at once (Document Object Model), traverse it, change it, stick events and callbacks onto it, and all with a magnificent API that surprisingly - up until that point in time, didn't really exist.

With HTML5 shifting in, changing browsers, standards, and development practices at around 2011, there was a need to manage even larger apps. If you still used jQuery for that, you would find yourself managing buckets of state spilled all around your code, and more importantly, all around the DOM - and unfortunately, that DOM belonged to the browser, and not you. And, you can bet each browser liked to have its own quirks around the DOM. The title "Front-end developer" emerged shortly after, because now you needed an entire person with a full body of skills to handle that kind of complexity.

And then started the era of frameworks. Front-end developers needed to refine their newly born tool-belt. They needed to do *efficient* work and to consolidate and abstract away tedious, repetitive tasks they were performing each day. Frameworks like Backbone.js[11], Batman.js[12], Ember.js[13], and later Angular.js[14] were all becoming mainstream in a span of just 2 years, creating a massive flood of ideas and solutions.

But also, an impossible mess. These frameworks abstracted out browsers, DOM work, introduced MVC on the client-side, but in my opinion, apart from maybe Ember.js, didn't really 'solve' client-side development. Until React came around.

[11] http://backbonejs.org
[12] http://batmanjs.org
[13] http://emberjs.com
[14] https://angularjs.org

React.js

At launch, React took a less ambitious path than any of its heavy-weight contenders such as Angular.js or Ember.js. On first look, there was only a view layer, some sort of event mechanism, and JSX - the React flavored markup language. Many thought React was a View layer replacement in the then common MVC architecture.

Then, on a second look and a year later, you'd see that React also handled state, component composition, styling, while performance optimizations and DOM manipulation were abstracted out and done for you. This is textbook API bonanza. The *pitfall of success*.

And, on a third look and a couple years later, we realized that by being minimal, React allowed for the community to evolve creatively and develop opinions and a massive ecosystem grew in a velocity never seen before. With new design patterns such as Flux and functional patterns such as Redux, all paying homage to things[15] that[16] already work in software for the last 30 years. Soon enough, even the enterprise-adopted, Google-backed framework Angular gets to experience a decline.

React Components

React allows you to make components in a very detached and independent way. Never before there was a Javascript framework that let you make declarative yet functional component in such a fun way. You can create views and not care about the browser or DOM, style and not care about CSS, and implement functionality in a completely encapsulated way not caring about any of your dependencies.

While the scope of this book was not to teach you React, I released it with a "something missing" feeling. I wanted you to experience the *playfulness* of building a mobile app that only React Native enables, even before diving into building a real app - and you *can't* do it in a book because it's not interactive - it's too "dry"; don't let any other book fool you.

That's why I created React Native Katas[17]. It is an open-source project that serves as a complementary learning experience for this book (and also a stand-alone experience), that you can go through yourself and have your friends try out - whether they're designers, beginner programmers, or just product people (with a minimal intro to Javascript, of course).

React Native Katas is a hands-on, immersive and playful experience, that teaches you about building React Native Components while *doing*. You can

[15] https://bitquabit.com/post/the-more-things-change/
[16] https://en.wikipedia.org/wiki/Lisp_(programming_language)
[17] https://github.com/jondot/ReactNativeKatas

check out the repository here[18].

We'll dedicate the rest of this chapter to show you how to work on just one Kata because the majority of the fun is not on these pages: it's on Github and in your editor. You can even ignore the rest of the chapter and have fantastic time playing with the Katas right now!

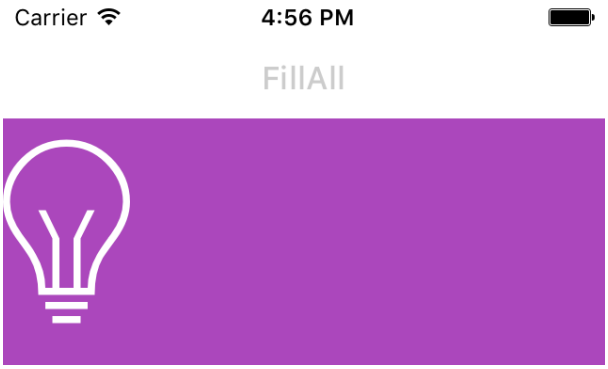
Completing a Kata

Let's go through the first Kata, just to test out the waters. I trust that you'll continue on your own after this. I assume you already have a working react-native setup[19].

- Clone project the project with `$ git clone https://github.com/jondot/ReactNativeKatas`
- `cd ReactNativeKatas` and `npm i` for installing the dependencies.
- Run the project via Xcode or `react-native run-ios`
- After running the project, turn on Live Reload (Ctrl+Cmd+Z for developer menu on Simulator)

You will then be faced with the first Kata, the test version or the "unsolved" version:

[18] https://github.com/jondot/ReactNativeKatas
[19] https://facebook.github.io/react-native/docs/getting-started.html

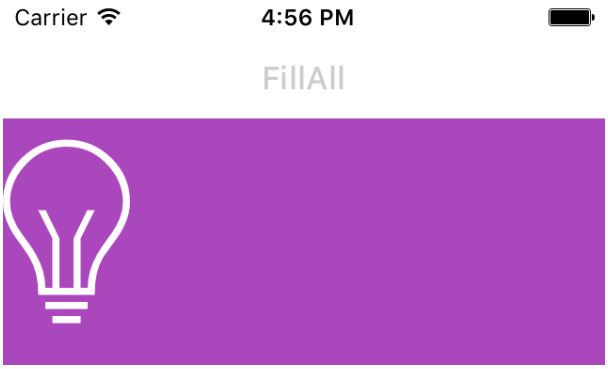


A Kata waiting to be solved

Tap on it anywhere, to see what the solution should look like:

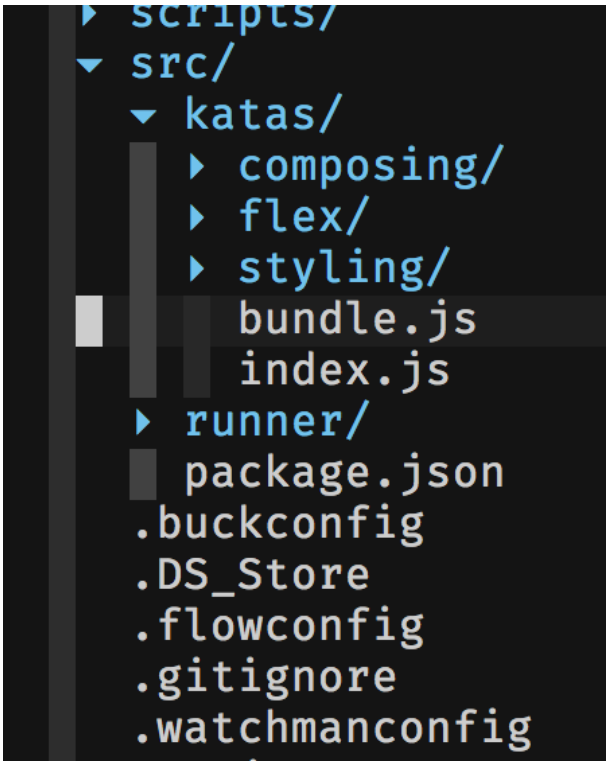
A solved Kata

And tap again to show the test version:



A Kata waiting to be solved

Next up, let's get orientated. Locate `bundle.js` with the relevant source tree, under `src/katas`. This will list all of your pending Katas in various subjects: *styling*, *flex*, and *composition*.



The bundle file

Opening the bundle, you'll find this:

```
// Katas TOC
//
// This is upside down (imports at the bot
// focus on what matters.
//

const bundle = [
  // flex
  kata(FillAllTest, FillAll),
  kata(DirectionTest, Direction),
  kata(AlignmentAxisTest, AlignmentAxis),
  kata(AlignmentTest, Alignment),
  kata(FlexSizeTest, FlexSize),
  kata(GriddingTest, Gridding),
```

Listing all Katas

And then, we see the first Kata is the `FillAll` Kata. We want to open the `.test` version of it, which is what we should edit.

```
// Filling and Centering
//
// Your goal is to make content dead center, and make background fill ev
// Hint: You have to combine multiple properties
//

const FillAll = (props)=>{
  return (
    <View style={styles.container}>
      <Icon style={{color:'white'}} name='ios-bulb-outline' size={120} /
    </View>
  )
}
```

Code for the test Kata

If you edit anything and save in that file, React Native will detect it and Live Reload will kick in and reload your app. The end result: your iOS simulator will refresh automatically. React Native Katas has a built-in mechanism to detect if you have solved your Kata properly, and will advance to the next Kata once you succeeded until all Katas are finished.

Let's try to solve the first Kata.

```
// Your goal is to make content dead center, and make background fill everyth\
ing
// Hint: You have to combine multiple properties
```

So let's fill everything. Flex is our layout model, so feel free to Google 'flex layout' now. Katas take into account you are not coming prepared, so you'll find yourself Googling a lot - this is exactly what we want you to do. This is how learning by practice is done :)

Pause.

After Googling for flex layout, I'm sure you found out about `flex:1`. This is the basic instruction for flex, to let the view under that style take over any realestate that it can, as long as there are no competing views with competing weights.

So, we'll change the styles to this:

```
const styles = StyleSheet.create({
  container: {
    backgroundColor: colors[0],
    flex: 1,
  },
  text:{
    color: 'white',
    textAlign:'center'
  }
});
```

Save, and watch the Kata being refreshed. Now the background covers everything but we are still seeing this Kata - so it means it wasn't solved. It *did* say that we also need to center everything. Google is your friend now, try 'flex centering'.

Pause.

Well, that's `justifyContent`, but I'm just telling you this now - and you're not even supposed to read this part of the chapter because it's full of spoilers. I urge you to go ahead and do the Katas solo.

Still here? Alright, let's keep going!

Fixing the styles again:

```
const styles = StyleSheet.create({
  container: {
    backgroundColor: colors[0],
    flex: 1,
    justifyContent: 'center',
  },
  text:{
    color: 'white',
    textAlign:'center'
  }
});
```

Save again. That made the content align to center, but not on *both* axis. The hint did say we have to *combine* a few styles to get that. Again, Google.

Pause.

Googling around, we find out that we're missing `alignItems` as well!

Surely enough, this is the *final* fix:

```
const styles = StyleSheet.create({
  container: {
    backgroundColor: colors[0],
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  text:{
    color: 'white',
    textAlign:'center'
  }
});
```

And we're done! There's a new Kata to solve.

A solved Kata

So we've just finished up a fun, *learn the hard way*, experience of building a React Native component. And now you're on your own (not really: you can always look at the code for the solution Kata!).

Go ahead and have fun, and when you're done you can go back to reading this book with more confidence than before! (or you can just keep reading, and do the Katas later. Or not do the Katas. Uhm.. you get the point :).

The People App

In this chapter we'll cover the *People App*, which is the app we'll use throughout the "work out" parts of the book. We'll dissect it, and then go over it bit by bit so that you will understand every decision and considerations you need to apply while building a React Native app.

We'll also cover what makes it tick - its features and technical building blocks, and we'll also discuss how far you can take it later on your own. In the following chapters we'll start setting up our environment, survey our app, and make a deep dive.

Features

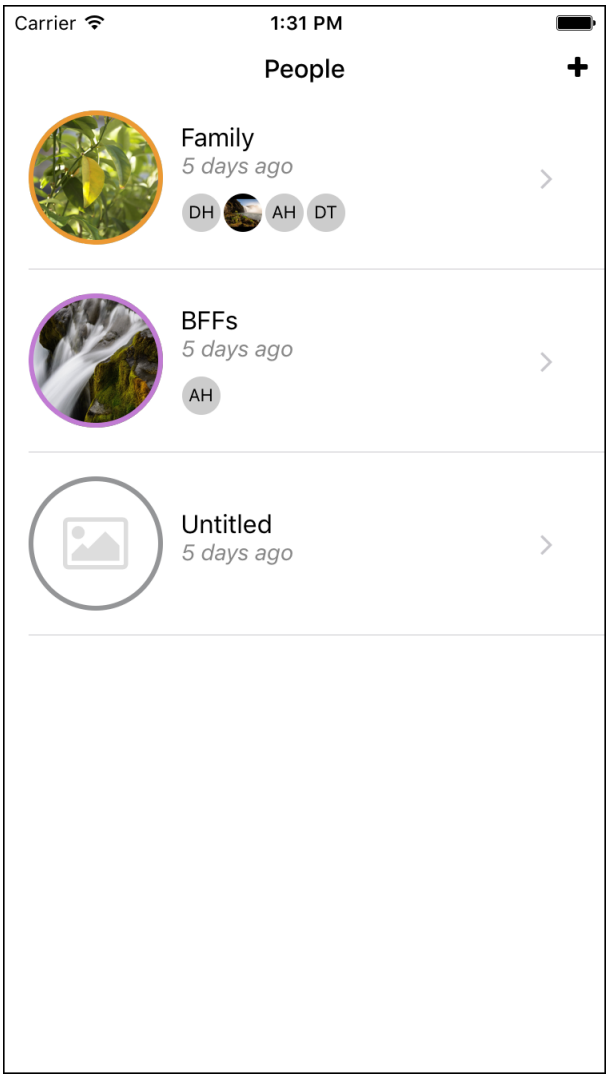
Let's go over our features:

- Create and manage groups of contacts
- Customize groups by color coding, custom image, etc.
- Pick an image from camera roll or take a live one
- Select contacts from the phone's existing address book
- Consistent UI for both iOS and Android with animations and routing
- Swipe to delete (iOS-like)

Product

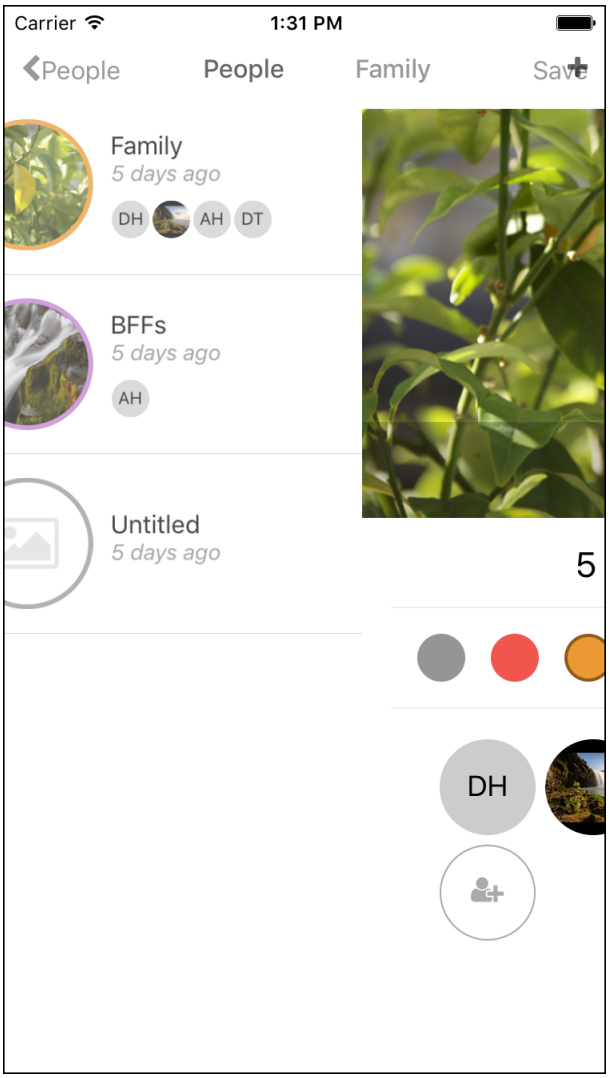
Let's breeze through the product, by looking at screen shots, grab on to something!

Our main app screen. This is the group list, or "Master" view in the master-detail relationship (more on this later). Notable visuals are color coding the main group avatar (or cover), a rather complex list cell layout with embedded contact photos and more.



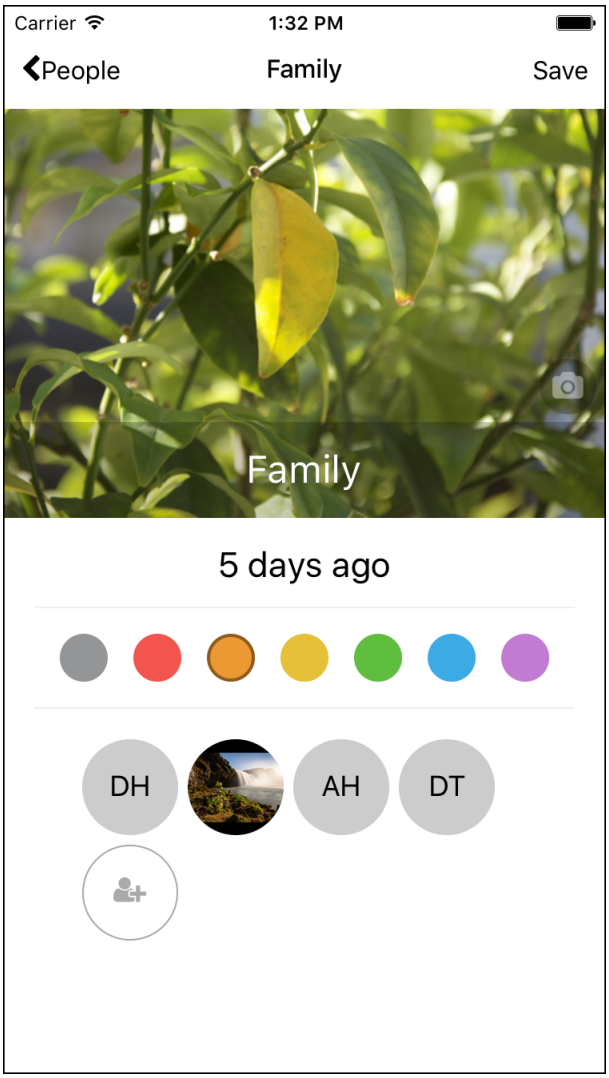
Groups List (Master)

Transitioning to the group editing screen, which is our detail view in the master-detail relationship. Animations run butter smooth, with typical left ease-out and fade-out followed by fade-in out of the box.



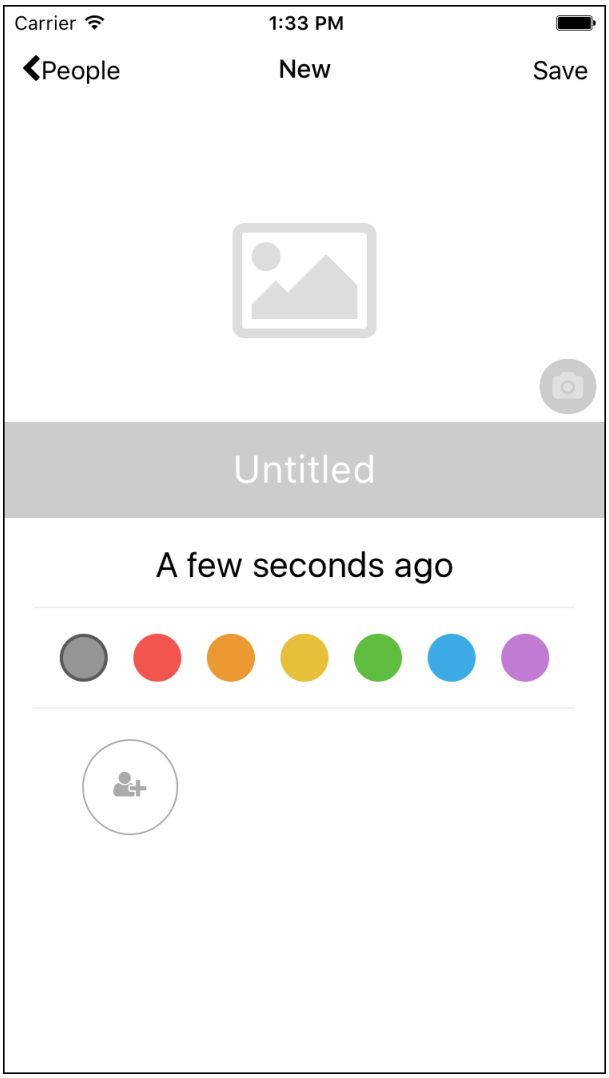
Transition to Detail

Our group screen ready to be edited. Typical iOS behavior dictate the “Save” right button on the navbar, we’ll carry this on to the Android version as well.



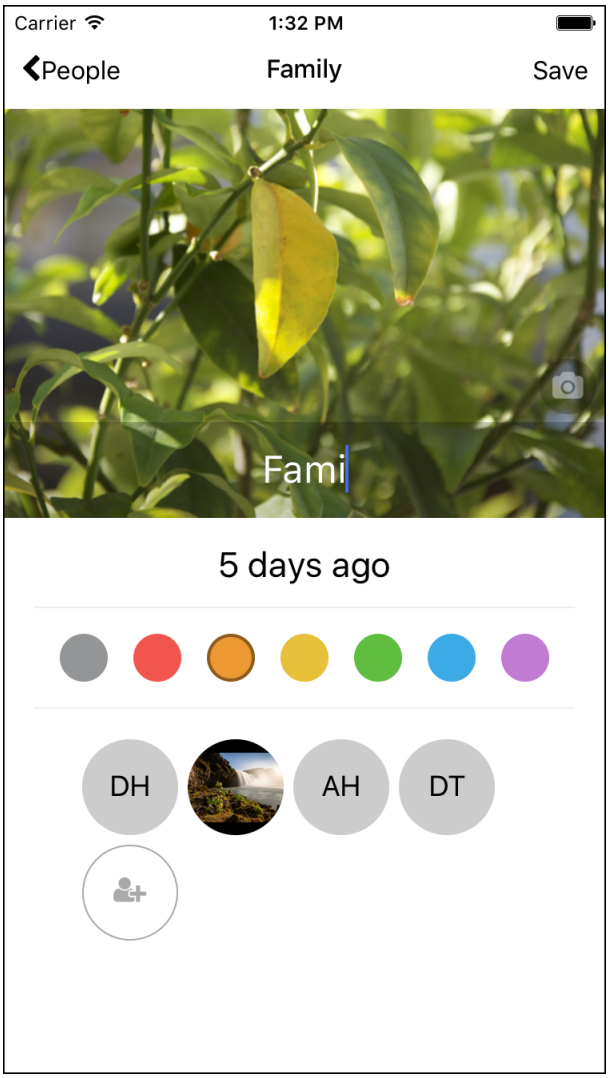
Group Screen (Detail)

And this is how the same group screen looks empty, when creating a new one.



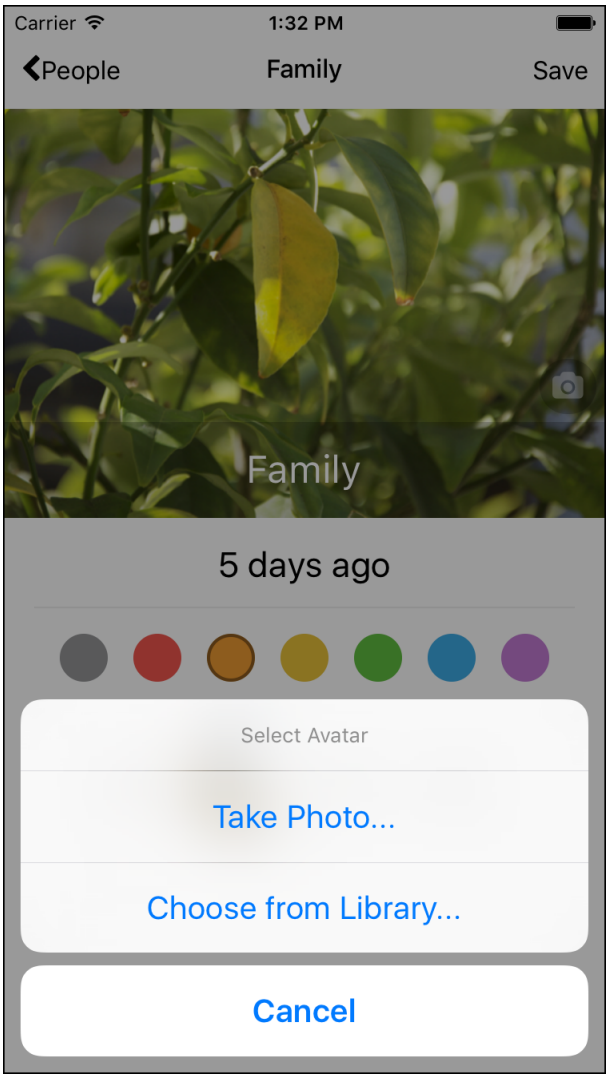
Group Screen, Empty (Detail)

We can edit the group name pretty easily with edit-in-place.



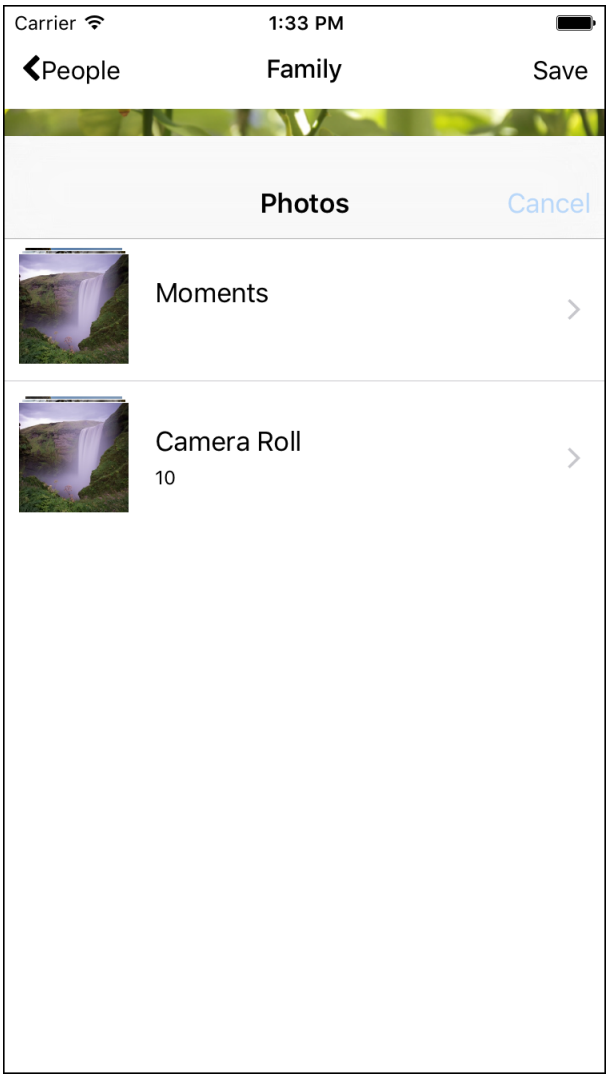
Editing Group Name

And pick an image using something that looks like the standard iOS actionsheet for selecting an image.



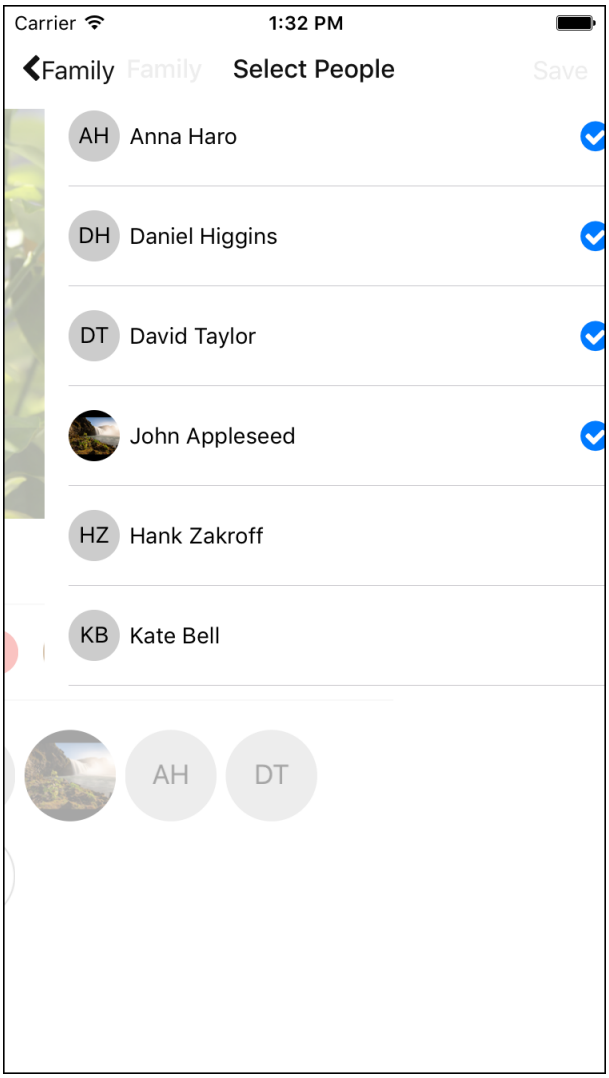
Pick Image

Then, we're picking an image from iOS' own camera roll. No need for custom UI to be developed here.



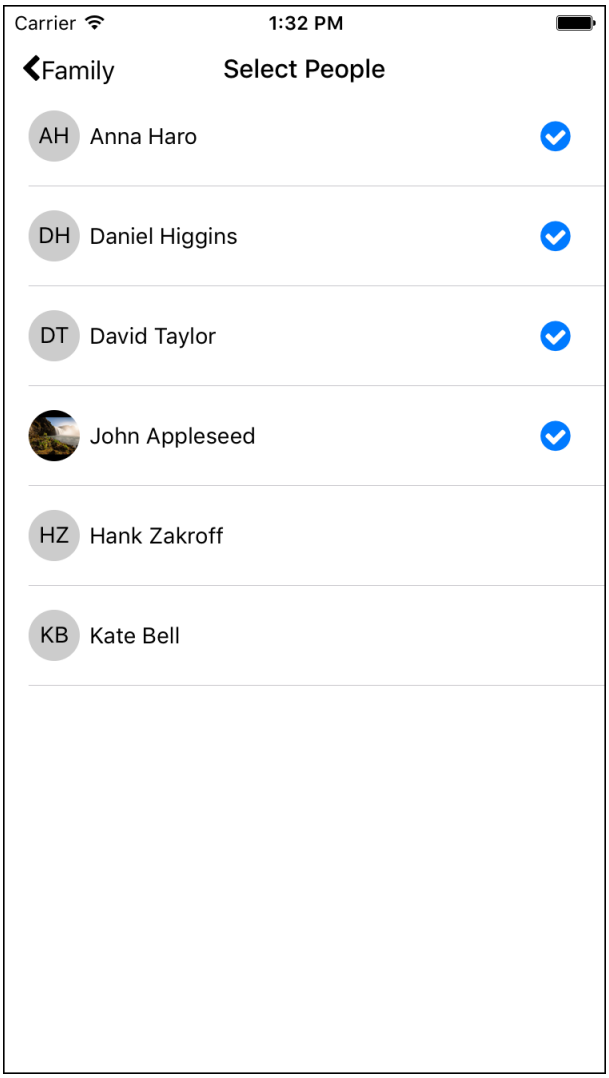
Select Image from Camera Roll

Hitting the person-plus icon to manage the contacts, we transition to the contacts screen.



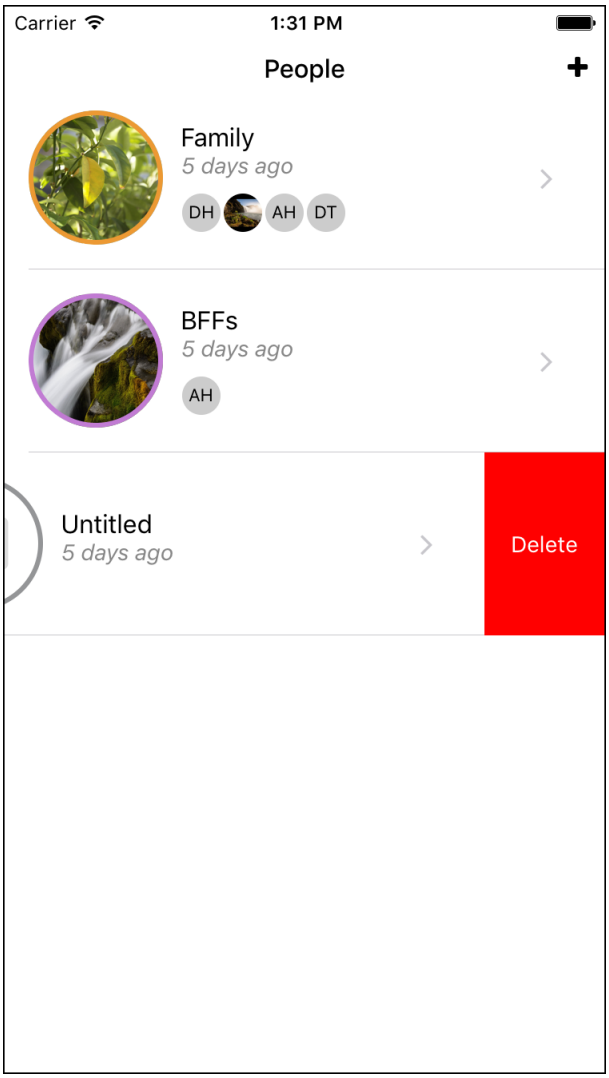
Transition to Contacts

And this is how the contacts screen looks like. It’s quite live and every contact we select or remove makes the whole view reorder itself based on selection.



Our Contacts Screen

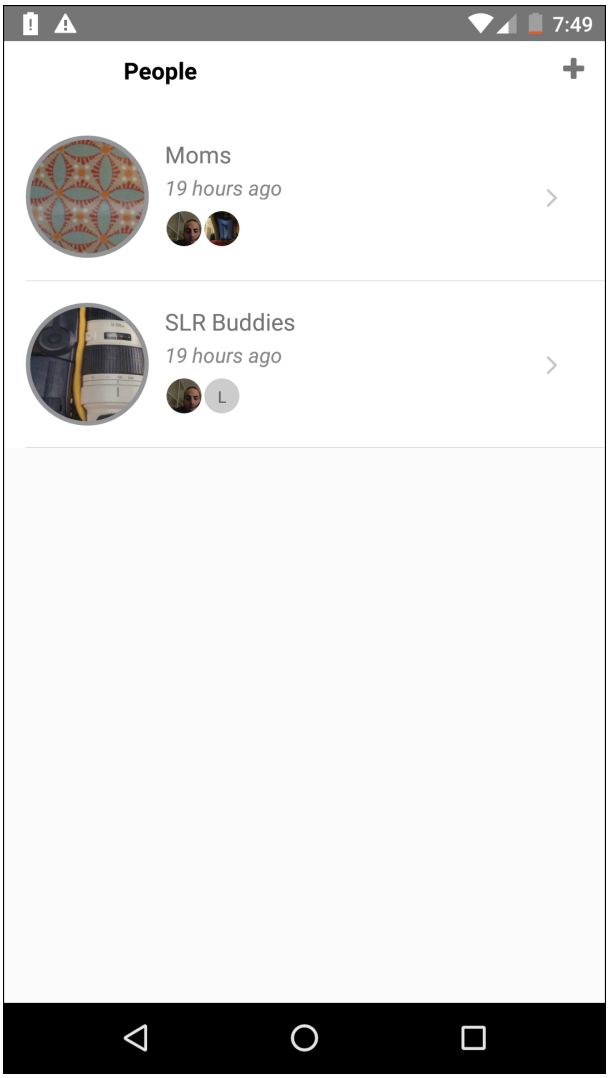
Back to our groups list, we show off the swipe to delete behavior from iOS, which we'll carry to Android as well.



Swipe to Delete

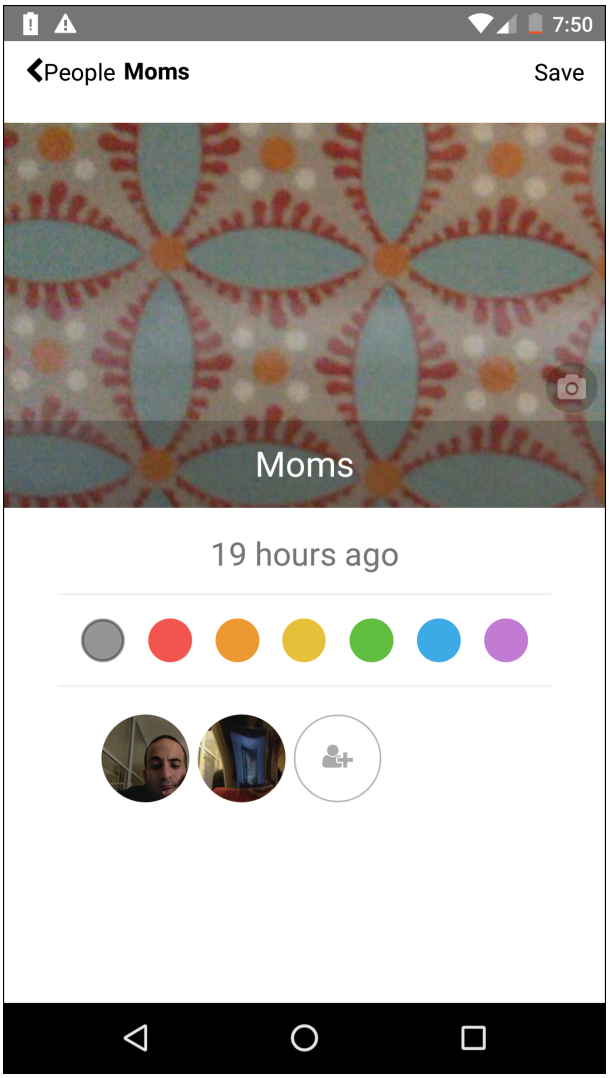
Just so you can believe me, let's go over the Android screen shots of the very same codebase. Virtually no effort was made to make the UI look and behave the same on Android, and this is as raw as it gets. With a few more hours of fine tuning both UIs could look the same by the pixel, but then again, should they look the same? Expect fun discussions with your product managers, but at least you have the power to do both ways!

Our groups list, on Android.



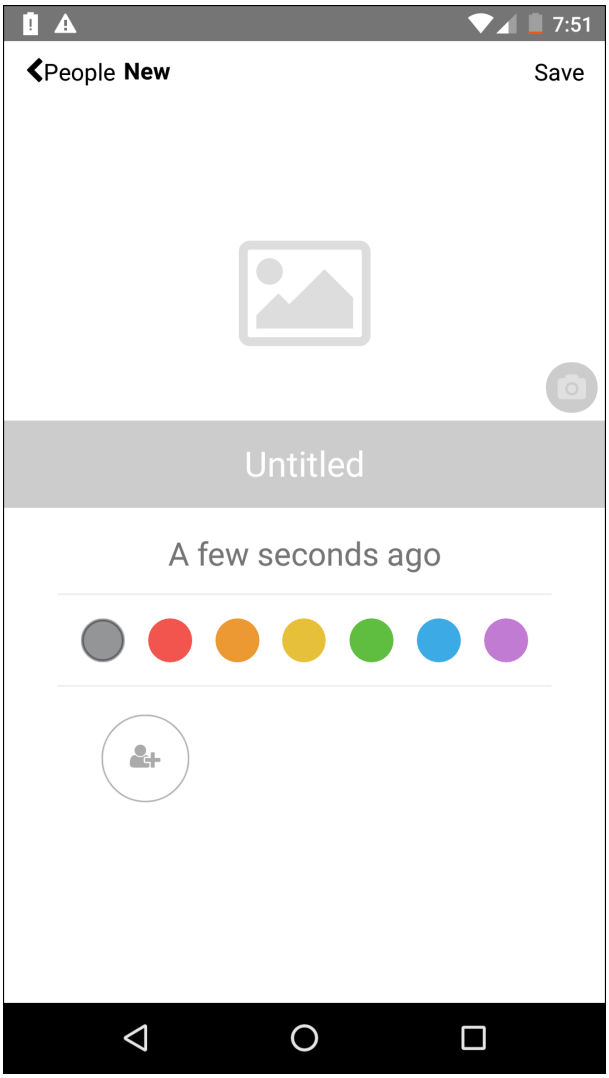
Android: Group List Screen (Master)

The group screen, on Android.



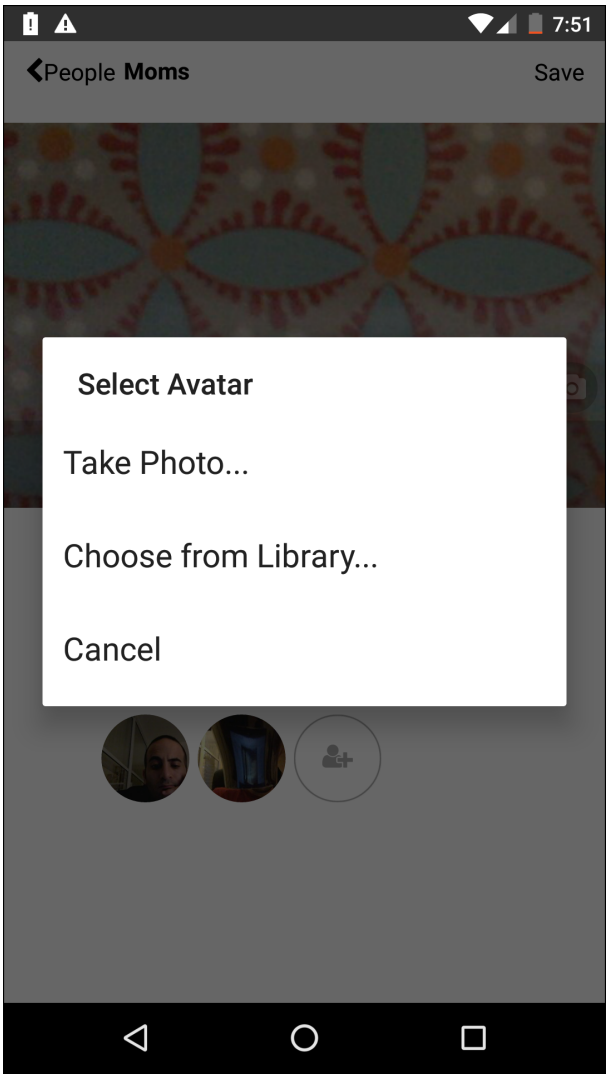
Android: Group Screen (Detail)

And here is how it looks like empty.



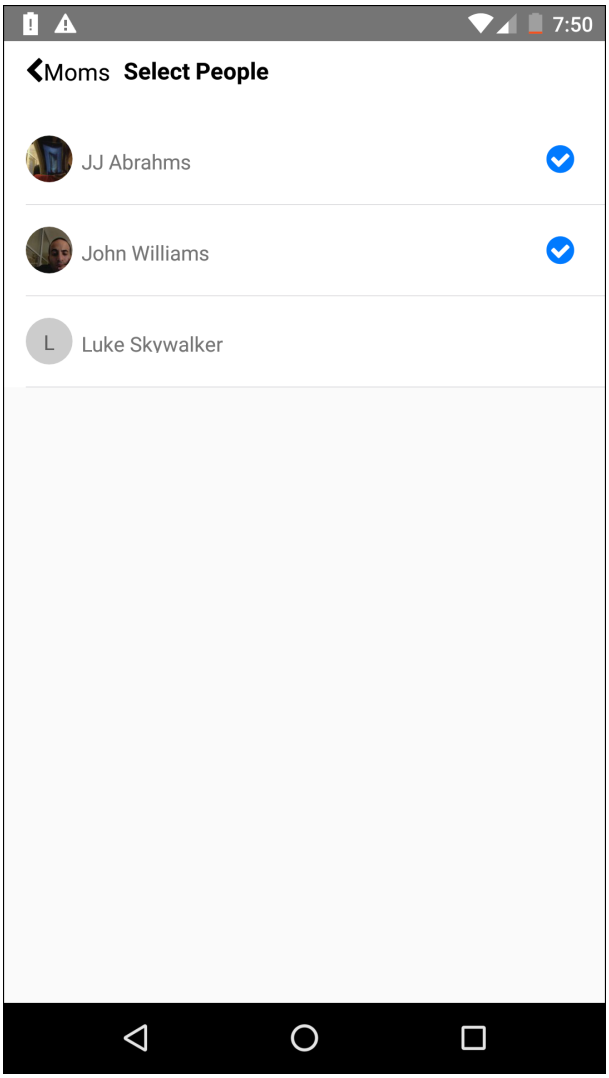
Android: Empty Group (Detail)

Picking an image.



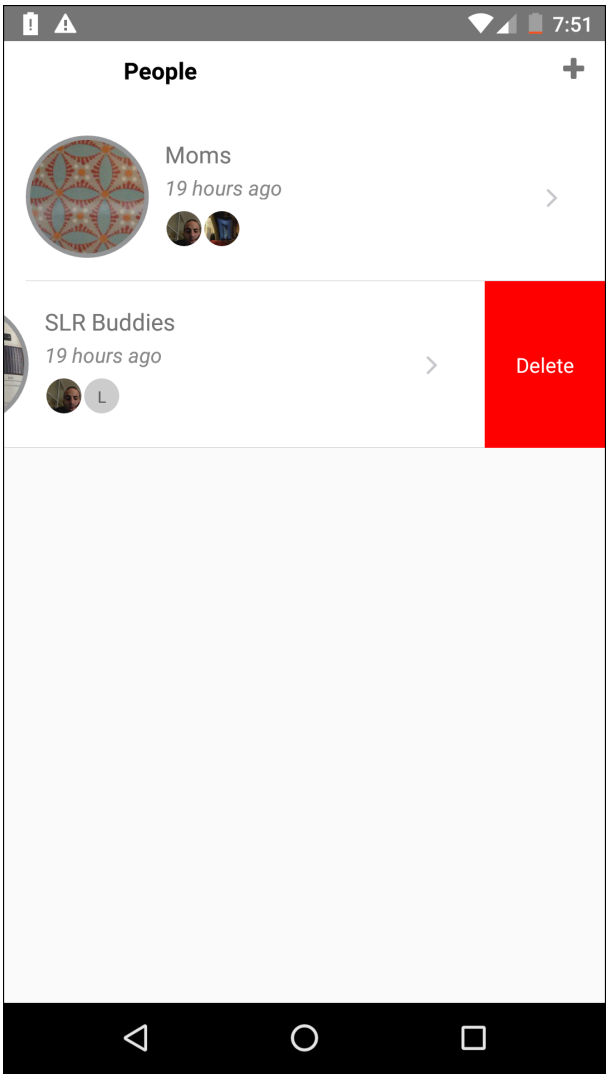
Android: Image Picker Sheet

And the contacts screen.



Android: Contacts Screen

Swipe to delete on Android.



Android: Swipe Menu

So, all of the Android screens are very much alike, you'll notice that the navigation bar is rendered a bit differently, which is very much expected. Many mobile platforms have elected to create different navigation systems and UI and it is so different both in mindset and execution that we'll cover navigation as a general subject later. For now, looking at these screen shots we can say that it is similar and that the differences are intentional.

Technical

In the previous sub topic we've covered how the application looks like and you might have felt a bit how it behaves from the transitions and the wording. Now, let's take a moment and cover the technical aspects of what we did, or what we need to do in order to get to that kind of an app.

First, we get React Native as a full blown off-the-shelf cross platform SDK. This means we have quite a few components[20] already when we start using

[20] https://github.com/facebook/react-native/tree/master/Libraries/Components

it. This allows us to compose our main screens, our list views (on group list screen, contacts screen), build the more intensive UI elements such as the color tag selection, various avatar boxes for contacts and groups and so on.

Next, we use several community contributed components to make implementing UI components easier:

- Swipeout[21] - to make the swipe-to-something experience in our groups list screen. In our case we only swipe to delete
- Listitem[22] - to render an iOS-like list item in terms of UI
- Vector icons[23] - to pack every free vector icon package to be available to us easily. This includes fontawesome and more
- Image picker[24] - to streamline the image picking actionsheet across both iOS and Android, using native UI where possible, and this is all transparent to us

In addition, we use several community contributed modules, that allow us to reach out into each platform's (iOS, Android) operating system services from React Native Javascript code:

- Contacts[25] - to grab all of the device's contacts in a structured manner
- FS[26] - to get access to important file system operations as well as metadata (Documents folder location for our app).

Internally we're doing a mini Flux pipeline with a store and an event emitter but don't pay too much attention to that. It seems that there is a lot of discussion about which Flux framework is the best, but it is way too much of an opinionated discussion to be able to be contained in this book. In real life projects you may want to pick alt.js[27] for its all-included approach or Redux[28] because it is a framework that lets you build your own framework.

We also didn't explicitly mention this - you can use almost all React.js Flux frameworks with React Native, which is quite a community accomplishment on the road to completely fuzzing the line between these two.

[21] https://www.npmjs.com/package/react-native-swipeout
[22] https://www.npmjs.com/package/react-native-listitem
[23] https://www.npmjs.com/package/react-native-vector-icons
[24] https://www.npmjs.com/package/react-native-image-picker
[25] https://www.npmjs.com/package/react-native-contacts
[26] https://www.npmjs.com/package/react-native-fs
[27] http://alt.js.org/
[28] https://github.com/rackt/redux

Potential (or: Homework)

The app we're building is completely functional, and is not a made-up or useless demo app like a weather app or yet-another todo list app. You can take this as a great starting point for any people based apps: meetups, chats, events, social network and so on. Some examples:

- By adding a calendar to the group view, this becomes an events app. Add a notification option as well and you're golden
- By converting the group view to a chat list and connecting it to a reactive backend like Firebase[29], you just made a chat app
- If you keep the group view as-is but make it more actionable, like making a tap on a contact's avatar call him or her, or record an action on a server which everyone is connected to, you've just made a variant of a social network.

It might help figuring out if you want to make any of these while going over the book or just follow the book as-is. You might also want to make it a two-pass, by finishing the book as-is and then hacking in your variant of the app and treating the book as a desk manual you pick up once in a while.

Summary

I hope you've got a glimpse of the power of React Native. We just covered an app that was built for two platforms. Unlike typical beginner level courses or books, we cover a real world app that can be made into real apps quite easily and serve a meaningful purpose (no more weather apps!). This is to show that finally - a native, performant, cross-platform app really *can* be built in 2016 - using React Native.

What you didn't get from this chapter so far, is that the experience of building such an app was easy, and even fun. I'm sure you'll get to feel it as we go along!

[29] http://firebase.com

Walkthrough

In this chapter, we'll be starting our breakdown of the *People* app. You'll get to understand the formula that most apps are adopting, the technical choices, the whys, and the hows of building a real world React Native app. We will then dissect each and every screen and understand how React and React Native play together, as well as supposedly non-development concerns such as styling and resource management.

Walkthough Style

Let's set some expectations and goals from the upcoming walkthrough. As it stands at the time of the writing, there are plenty of ways to go when you want to learn how to build an app. You can pick a Udemy or Egghead course or an online tutorial, or follow a booklet that takes you step by step, line by line from a blank project to a fully working app.

This book takes a more challenging approach, specifically we're taking the dissection approach - you've seen a fully working app, and we're going to take it apart, understand the whys and hows and extract the *distilled knowledge* you need in order to make your own app; that's because I know you will continue reading and continue learning through resources like Udemy and Egghead throughout your career - we never stop learning do we?

This approach also deals with the volatility of the Javascript world, the React world and the React Native project (which is still new). In other words, if I wrote a tutorial style book, chances are within 6 months you would lose all value for your money because by then either Javascript changed (ES6, ES7), React changed, or React Native changed. Instead, I've chosen to focus on core principles and patterns that *will* hold up through the test of time, and show you old ways as well as new ways to do things (Javascript, ES6, and so on) should you bump into old or new React Native code on the Web. This is a pragmatic guide, a distilled knowledge guide.

With that out of the way, let's start!

Technical Choices

As a reminder, the *People* app is an app about your people. It is a contacts management app on steroids which you can take and build your own apps upon. Currently it holds a list of groups of people, their contacts and several

eye candy features to promote easy lookup and personalization. If you'd like, you can skim the *The People App* chapter to get a full blown overview of the app as well as its product and infrastructural building blocks.

By now the Javascript world is known for its churn[30], and because of that, I feel that the first items we need to get off the table are the choices that we make for the code base we're about to dissect, so let's do that now.

Javascript

For our Javascript codebase, we use a mix of both ES5 (probably the Javascript everyone is familiar with), and ES6 (the sixth edition, introducing many state of the art construct into the language) interchangeably. This is to support the old and the new. Often times I was either looking at an old React style code I needed to support, but it looked strange to me because I happened to already forget the old way of doing things, and then again I looked at a new React code and the same thing happened because I wasn't yet familiar with another new way of doing things that was introduced in the mere two weeks I was doing back-end or native mobile code.

Many say that 2014-15 were the years where Javascript became a jungle; I'd like to show you both ways of doing things in React so you can get your bearings right when you feel you're stranded in that jungle.

React

The good news with React Native is that you get a fairly closed-box React. In my opinion, taking the whole weight of Javascript and React choices and decisions *in addition* to mobile development is way too much to bear, that if it were that way it would be a major turn-off.

With React Native you get the basic components you can build a whole world with and they're pretty baked out for you, and you get styling which is done inline. No hours of fiddling with Webpack (that's the reality at the time of writing) to make a complex build pipeline that extracts CSS into its own files, merges sprites and so on. More over, you get a daemon that watches your files and packages everything neatly already - so that's also something you don't need to configure. In addition there's a tight integration with tools like Flow[31] so even though React Native as a software project sees React - the framework - as upstream, that the whole setup feels quite modern and up to date.

Flux

Choosing and arguing about which Flux framework is best is all the rage these days. To keep our focus we'll define Flux as a structured way of doing

[30] http://www.breck-mckye.com/blog/2014/12/the-state-of-javascript-in-2015/
[31] http://flowtype.org/

event based application architecture, and in my eyes it is an evolution of message based event handling and I completely agree with this article[32].

To keep our focus, I've chosen to build a slim version of a Flux pipeline with a simple event emitter and a store. If you'd like to use a framework, you can check out alt.js[33] or Redux[34]. So how to choose for your own future projects? It depends. I hold the opinion that if it is a small enough app, just go with what ever saves you more work (*alt.js*), and if its an app that you'd want to maintain for a long while and that you see a considerably large roadmap for, choose something simple and small that you can build on later (*Redux*).

Folder Structure

We'll use the following folder structure for our project. In my opinion, a project's folder structure should be primarily shaped by the team that works with it, and only then it is shaped by the technology stack.

```
views/
  helpers/
  master/
    index.js
    styles.js
    other.js
components/
services/
[ any flux-oriented folders such as actions, stores, etc. ]
```

In our app, I've chosen to focus on React Native and not Flux, and this is why you won't see the typical mega-structure of folders that is driven by a Flux project. In our project we're holding a store in `services/` and making simple actions objects, and that translates to a single file and a framework such as `alt.js` to conform to Flux. Yay for simplicity!

For the `views/` part, we are holding the main component in an `index.js` file so that when we require `views/master` it will get pulled automatically, and then in our `index.js` file when we simply require `./styles` it will get fetched locally from that view's folder. In other words, we're keeping the styles neatly tucked along the component or view that uses it (we'll talk about styles and why they're actually a Javascript file later).

Other ways of doing this? You could hold a central place for styles, in a single file. You could replace this entire folder structure with one coming from vanilla Flux, or you could use a structure you're bringing from React.js if you're a React.js developer already - that would be a huge advantage.

[32] https://bitquabit.com/post/the-more-things-change/
[33] http://alt.js.org/
[34] https://github.com/rackt/redux

Testing

The combination of React and mobile for testing is amazing. In my opinion React is almost testable *by design*. Its functional nature lets your test be clean and free of setups for side-effects, and the fine grained way of building things into components supports unit testing very well. That covers all of the Java or Objective-C code that you may or may have not written along the years if you're a mobile developer, and that grew to be untestable because it underwent a process of rot, and got labeled as "untestable, we'll get to it some day".

Further along, there's integration testing, or automation testing. The mobile world is prospering with tooling for automation testing, even more so than what the Web exhibited in its golden ages of automation testing with tools like Selenium. Apple made their own in-house UI testing framework, Google pushed their own as well, and both are pretty good, and tools like Appium[35] that aim to provide a single testing codebase for both platforms are getting to a high level of quality – and we're here to enjoy it all.

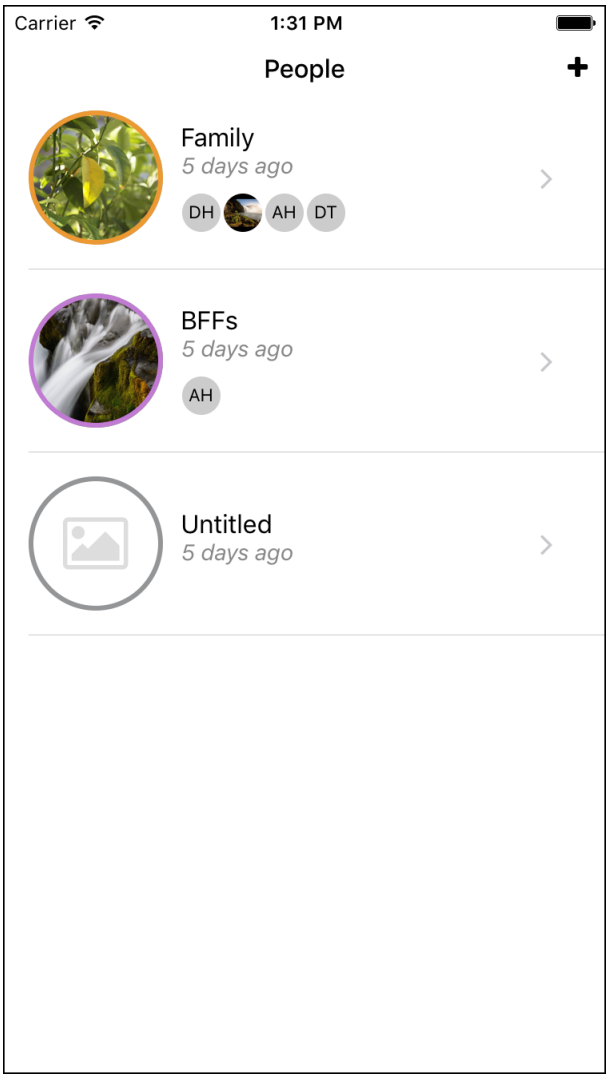
There is so much to say that there is simply way too much to cover and this topic should really make it into a book of its own. This is why in *this* book I've chosen not to take sides, and sadly we won't be covering that.

Dissecting Our First Screen

Let's take our first screen head on, dissect it and cover all of the material needed in order to build such a thing.

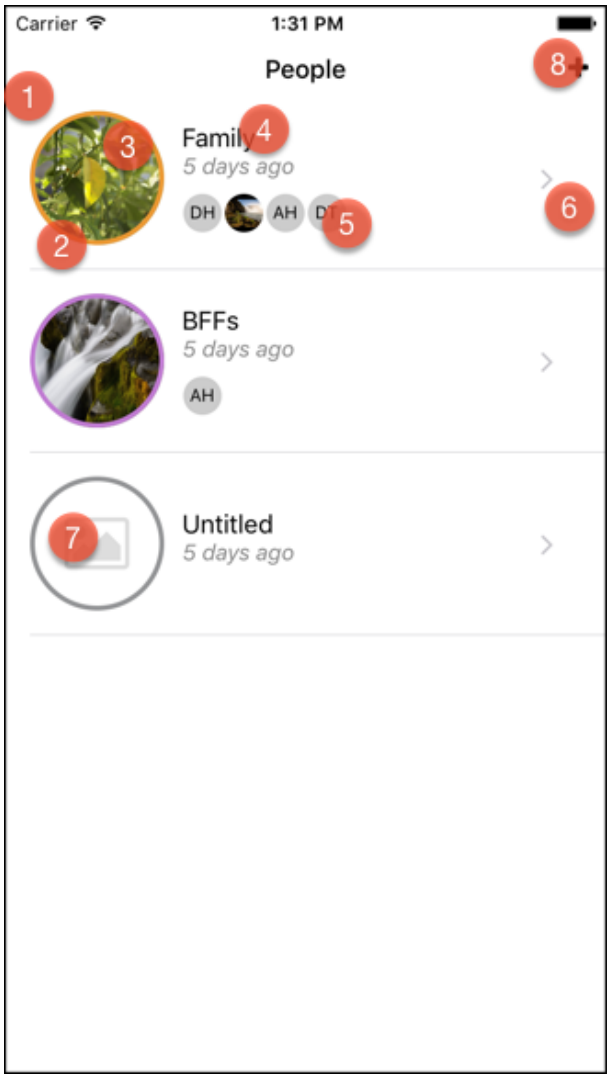
Here is how it looks like:

[35] http://appium.io



Main App Screen

Before we proceed, let’s see what we’re dealing with:



Main App Screen Annotated

1. The ListView[36] component React Native comes with. We'll cover this more in-depth shortly.
2. Our list view cell. Every great list view implementation allows you to make your own cell template (more on this later) and the React Native one is no different, since it uses the iOS and Android implementations which already do this. Once clicked, the cell will take us to the detail view and it will populate it based on the selected item.
3. The group avatar. The image is picked from camera roll or the image storage on the user's phone.
4. The group title, and timestamp. Both of these are picked from the JSON structure representing a group directly, and we use `moment.js` to get a human readable "time ago" render for the time.
5. The contacts list. These are rendered live per group, from the set contacts within each group. We make repeated use of the same

[36] https://facebook.github.io/react-native/docs/listview.html

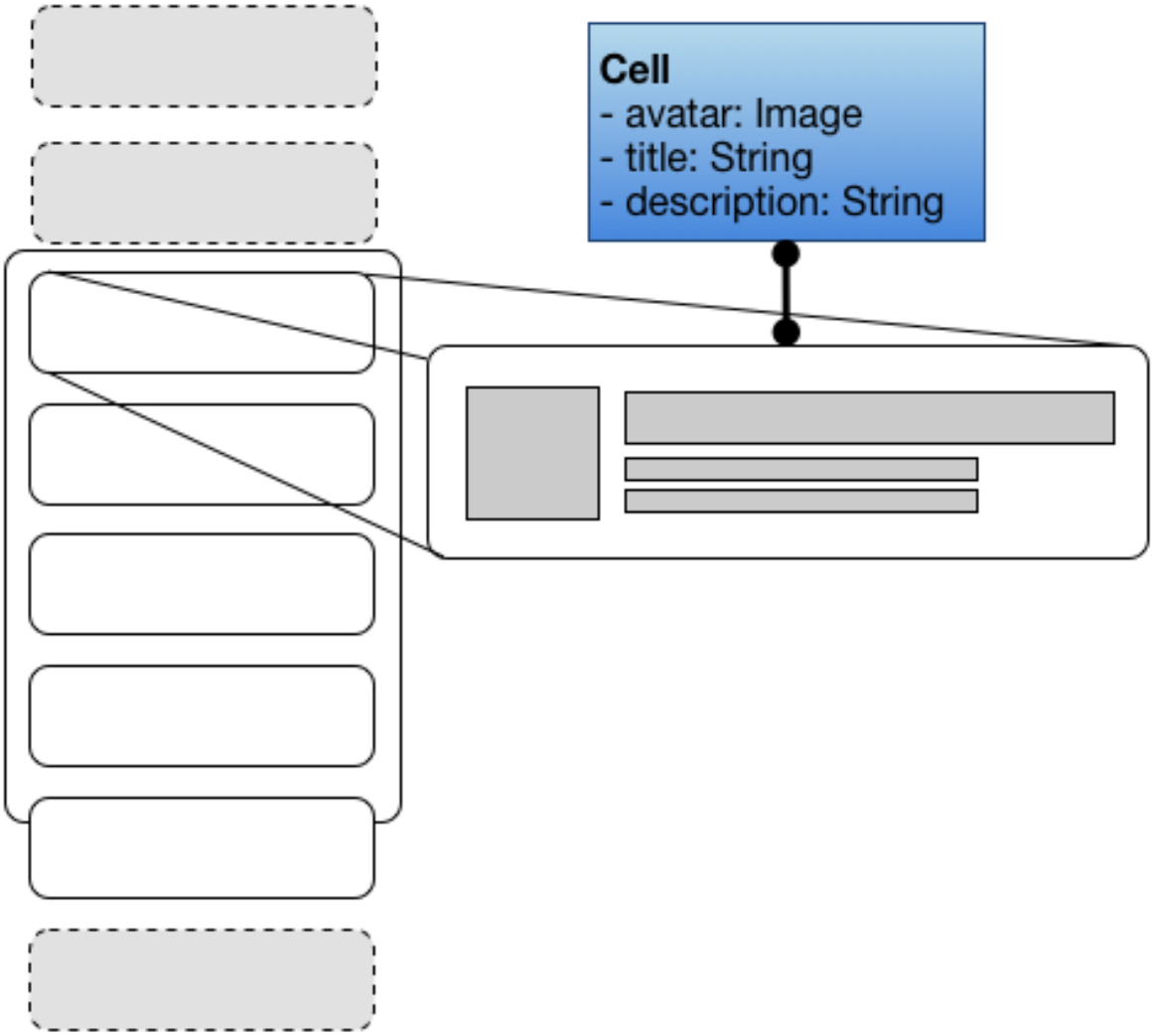
component for contacts avatars as well as group avatar here and in other places. We use Flex to make sure everything lays out correctly.
6. This chevron and general item design is a hint at iOS' default design for list items. We pick up the chevron symbol and other symbols throughout the app from the *vector-icons* community component (more on that later), so really it is a simple `<Icon>` element in implementation.
7. The UI knows how to render place holders when no image exists. We use React's ability to compute state and spit out a different UI tree when the data looks different.
8. This navigation button will create a new group and take us to the detail screen in a "New" routing state. We'll cover routing as well.

Before we continue, we *must* cover the list view, the pattern and what it means on mobile, and how it applies in React Native.

Mobile List Views

The list view is the workhorse of the mobile application. Regardless of React Native, if you take a moment to *really* look at apps on your phone, you'll find that a lot of them are simply pimped-up list views (on iOS it is the `UITableView` and on Android the `RecyclerView`): Gmail is simply managing lists (inbox, trash, custom filters), Twitter is managing a never ending list, and even Settings is really a static `UITableView` on iOS.

Every iOS or Android book about mobile development covers its own list view variant and today when both platforms matured - the same principles exist. I've managed to distill these principles as they are very important to know now that you're at the helm of *both* platforms, as we'll see in the following image and discussion.



Mechanics of a List View

So a list view is a container of items, often accompanied with a wrapping scroll functionality. It is responsible of the following:

- Layout - making each item render, measuring it and fitting it with the rest in a chosen layout (list, grid or what have you)
- Rendering - each cell is an independent component, which is a very good thing. You can build each cell by its own right as if it never needs to go into a list view eventually, and then plug it in as you see fit. It helps the development experience, composability and testability. In React Native such a cell is simply a React Native view - nothing new to learn! If you're coming from a native platform then this may ring a bell - on iOS you can either use the default cell that comes with `UITableView`, or design your own `xib` component which you will rig into your `UITableView` later. On Android you would design a new layout XML file for the cell and inflate that per cell.

- Recycling - on iOS and recently (depends how you define “recent”) on Android, list views are only holding a fixed amount of cells “live” proportional to the viewable area of the list view on screen, and they will trick you into thinking there is a bigger amount of cells by recycling off-screen cells and pulling them back into circulation as you scroll. This is why in the drawing some cells are white and some are gray.
- Lifecycle and events - it will hand out events for each cell in turn. With the React Native way you set events as you’d set them for any React Native component, and the lifecycle events are still the same as with any component
  - again, nothing new to learn!
- Data and adapters - since each list view is responsible of the concerns we just mentioned, it is crucial that it knows how the data looks like - how many items, their content for deferring the render per cell, and when content changes. Such a list view need not care about if the data is coming from a Web service or the local database. This is where the concept of adapters come into play; so much like Android’s adapters, the React Native list view has its own.

The Groups Screen

We can now continue to look at the main groups screen. Let’s dump the code here and discuss it bit by bit.

```
1  'use strict';
2
3  var React = require('react-native')
4  var {
5    ListView,
6  } = React
7
8  var Cell = require('./cell')
9
10 var Subscribable = require('Subscribable')
11 var R = React.createClass
12
13 var Master = R({
14   mixins: [Subscribable.Mixin],
15
16   store: function(){
17     return this.props.store
18   },
19
20   getInitialState: function() {
```

```
21      var ds = new ListView.DataSource({rowHasChanged:
22            (r1, r2) => {
23              //hack, need immutability on store for this to be detected
24              return true
25            }
26      })
27      var list = this.store().list()
28      return {
29        dataSource: ds.cloneWithRows(list),
30      }
31    },
32
33    componentDidMount: function() {
34      this.addListenerOn(this.store().events, 'change', this.onStoreChanged)
35    },
36
37    onStoreChanged: function(){
38      var list = this.store().list()
39      console.log("MASTER: new list, items:", list.length)
40      this.setState({dataSource: this.state.dataSource.cloneWithRows(
41        list
42      )})
43    },
44
45    didSelectRow:function(row){
46      this.props.navigator.push({
47        id: 'detail',
48        title: row.title,
49        props: { item: row,
50                 store: this.store(),
51                 navEvents: this.props.navEvents },
52      })
53      console.log("MASTER: selected", row.title)
54    },
55
56    didDeleteRow:function(row){
57       this.store().remove(row)
58    },
59
60    renderRow:function(row){
61      return (
62        <Cell key={row.id}
63              item={row}
64              onDelete={()=>this.didDeleteRow(row)}
65              onPress={()=> this.didSelectRow(row)}
66              />
```

```
67      )
68    },
69
70    render: function() {
71      return (
72        <ListView style={{paddingTop: 50, flex:1}}
73                  dataSource={this.state.dataSource}
74                  renderRow={this.renderRow}
75                  />
76      )
77    }
78  })
79
80  module.exports = Master
```

That's a lot of code, but surprisingly not too much. It's just logic and not much UI code, or so dubbed the "Smart Component[37]", and I like to call it "Controller View", so that people already familiar with MVC and other frameworks will have something to connect to. As such a view, we make sure the actual ownership of data takes place here (as opposed to within cells):

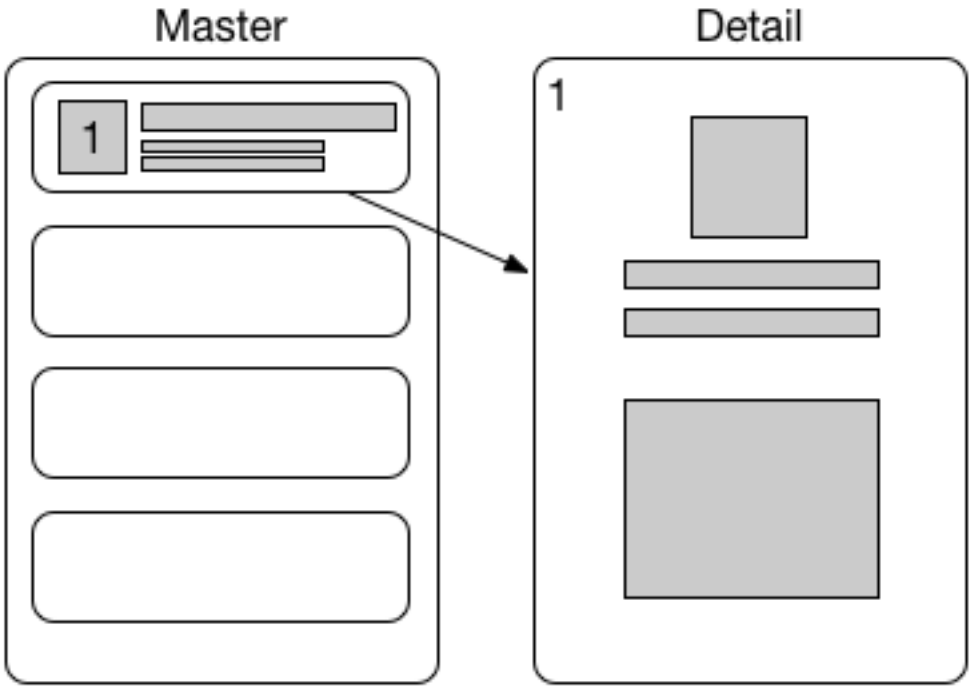
- Navigating to an item
- Deleting an item
- Data sync and notifications

I've also named this "Master" intentionally and not "GroupsView", and I've named the group cell just "Cell" and not "GroupCell" or "GroupItem". This is to push forward the idea that we're not doing anything special here, this is the same pattern all over again - Master-Detail. Let's side track to that now, and come back to the code listing afterwards.

Master-Detail

This pattern existed since the dawn of the ages[38], for mainframes, desktop applications, and now mobile apps. Here is an abstract drawing of what it means.

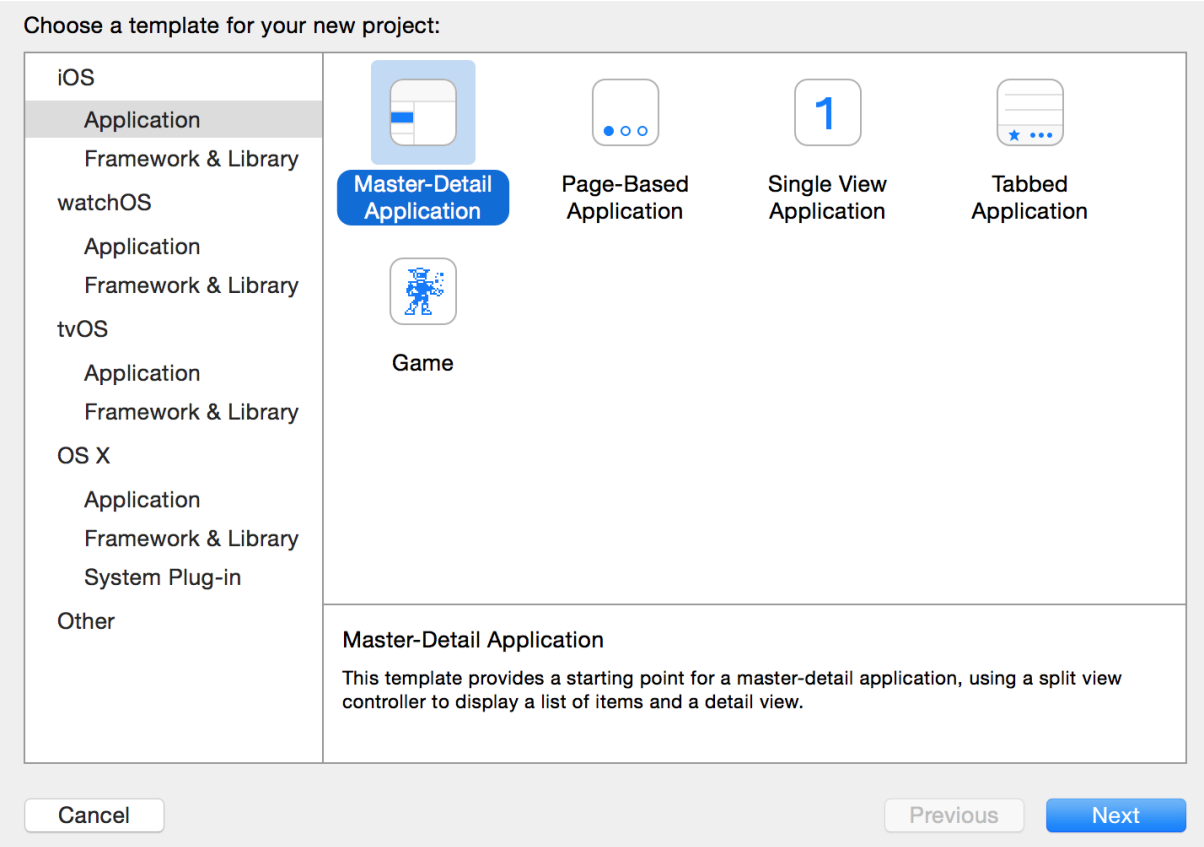
[37] https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.mh91y38yq
[38] https://en.wikipedia.org/wiki/Master%E2%80%93detail_interface



Typical Master-Detail

Basically we should always have a list of things - the *Master* view. And in React Native this will be our `ListView` component. Then, for each item, once selected, we render a *Detail* view. For our app, and in React Native, this will be a complete view we will transition to, handing it the data item from the list view in its props, via our routing stack (more on that later).

On iOS, you can generate a live master-detail based app fairly quickly:



Master-Detail Template with Xcode

Go ahead and try it, then run the app that was generated to play with it. It is essential that you develop the "Native" intuition for how things are done over at each platform even though you're focused at making a cross-platform app.

To wrap it up, in our app we're demonstrating these concepts:

- Master - the groups screen, at `master/index.js`
- Cell - the master cell component, at `master/cell.js`
- Detail - the edit group screen, at `detail/index.js`

For the rest of the code base we'll do away this pattern-ish way of naming things and call components in their semantic name. We're now ready to continue looking at the groups screen, or the *Master* view.

ListView and Our Master View

Since most of our *Master* view is `ListView` rigging and eventing, let's take a look at what it takes to make a list of objects rendered and reactive within a React Native `ListView`. As always, take a look at the official ListView docs[39] for a great overview of the `ListView` component.

[39] https://facebook.github.io/react-native/docs/listview.html

Let's pick apart our *Master* view.

```
'use strict';

var React = require('react-native')
var {
  ListView,
} = React

var Cell = require('./cell')

var Subscribable = require('Subscribable')
var R = React.createClass

var Master = R({
  mixins: [Subscribable.Mixin],

  store: function(){
    return this.props.store
  },
```

So this is how it starts, we're using the "old style" React conventions where we `React.createClass` to create a new component. The "new style" is the one encouraged by ES6, where we extend `React.Component` and make a proper ES6 `class` - we'll get to that as well. One of the reasons to use the "old style" is to be able to utilize existing *mixins* easily; not to say it isn't possible with ES6 based code, it is just a lot easier and co-exists with code you'll see around the Web as of the time of the writing.

Next is our store, which we'll get to later, and the `Subscribable` mixin, we'll get to those later as well.

Moving on towards the list view.

```
  getInitialState: function() {
    var ds = new ListView.DataSource({rowHasChanged:
          (r1, r2) => {
            //hack, need immutability on store for this to be detected
            return true
          }
    })
    var list = this.store().list()
    return {
      dataSource: ds.cloneWithRows(list),
    }
  },
```

As a proper React component, it is wise to provide a `getInitialState`, and that's a great place to set our `ListView` state. If you recall our discussion about list views, you'll remember that a list view expect some kind of abstract data source; this will be the following:

```
1     var ds = new ListView.DataSource({rowHasChanged:
2           (r1, r2) => {
3             //hack, need immutability on store for this to be detected
4             return true
5           }
6     })
```

Your `DataSource` is what takes a plain collection of items and makes it into a proper `ListView` data source. Here we're initializing it with a *change function* which is responsible for helping the `ListView` to detect changes in it. However, we must be careful here - if you're not doing proper immutable object work, don't expect it to work since you'll be changing contents of existing objects which the `ListView` already holds. So, given two rows, `r1` and `r2` it is supposed to compute if they are equal or not, and mostly that will mean comparing instance references. Otherwise, it would mean comparing object IDs or contents, and so on - but that is something you won't get out of, and you'll be paying a penalty for comparison.

The best possible way to do this the React way is to use immutability and an immutable collection framework such as Immutable.js[40]. Immutable collections, also called *Persistent Data Structures*, compose really well with React, and you can read more about here[41] and if you're drawn to academic papers there's this seminal work about the subject[42].

To conclude, when you use tools like *Immutable.js*, it is enough to just say `r1 === r2` and React Native will be able to pick up changes very efficiently. Otherwise, if your application architecture isn't based on immutability you may see that `ListView` would not be able to update itself, in which case I would try to force updates and `return true` instead of trying to manually create the concept of immutable collections without proper immutable collections (such as creating a new object manually when one changes) as long as you're feeling confident that there is no performance problem.

For this book, we have immutable collections implementation in a separate branch of the code; however to keep the discussion simple we'll need to keep away from that as we want to focus on the core techniques.

Moving on with the code listing, we see this:

[40] https://facebook.github.io/immutable-js/
[41] https://en.wikipedia.org/wiki/Persistent_data_structure
[42] http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf

```
  didSelectRow:function(row){
    this.props.navigator.push({
      id: 'detail',
      title: row.title,
      props: { item: row,
               store: this.store(),
               navEvents: this.props.navEvents },
    })
    console.log("MASTER: selected", row.title)
  },

  didDeleteRow:function(row){
     this.store().remove(row)
  },

  renderRow:function(row){
    return (
      <Cell key={row.id}
            item={row}
            onDelete={()=>this.didDeleteRow(row)}
            onPress={()=> this.didSelectRow(row)}
            />
    )
  },

  render: function() {
    return (
      <ListView style={{paddingTop: 50, flex:1}}
                dataSource={this.state.dataSource}
                renderRow={this.renderRow}
                />
    )
  }
})
```

We're skipping the store syncing functionality for now, until we cover the store itself. Let's start from bottom upwards. First, our `render` function is simply a declarative `ListView` render, with no logic or no complex UI composition at all; and that holds for a *Smart View* pattern quite well. We're doing *inline styles* for demonstration purposes - the purpose of these styles is to push the list view downwards below the navigation bar, and to make sure the container view flexes maximally along the view port (we'll get to styling later, but for now you can Google "flexbox" to get an idea of the layout model React Native uses).

We're supplying the `dataSource` we cooked up through the aptly named prop, and that is simply the data source that sits in our state. Additionally we

need to give it a `renderRow` prop, which is just a function returning the React Native view that is supposed to represent our list item, in our case - the `Cell`.

Moving on to `renderRow`, we're simply returning our `Cell` component with a few important props:

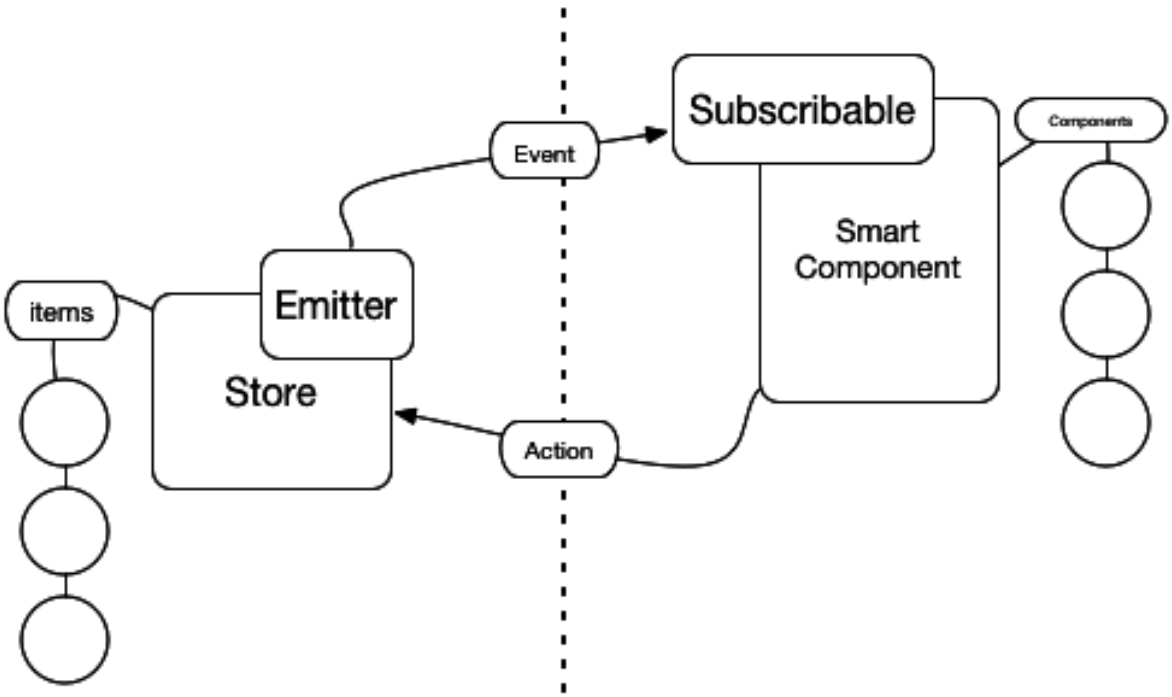
- `key` - this property is required for collection of components. It's how React Native knows how to track our views. Typically you will set it as 1:1 mapping of the identity of the object it renders. In our case it is simply `row.id`
- `onDelete` - this is our own custom prop. It signals that the user intends to delete this item
- `onPress` - this is our own custom prop. It signals that the user wants to interact with this item
- `item` - this is how this cell gets the data model it needs to render, our item.

Note, that the way each interaction method "knows" what item it needs to handle, is that we simply close over it with a lambda function! These interaction functions simply delegate to each and every interested party. In the case of deletion - the store gets notified about it, and in the case of navigation, well, the navigator gets notified about it. We'll get to both of these a bit later.

This completes our overview of the *Master* view. But we do have a couple of holes: our Flux store and navigator. Let's cover these now, and afterwards we can move on to the rest of the views.

The Store

Our store is simply a place where our data items are held and shaped. In addition it receives actions (in the form of methods) and dispatches back events. The way it inter relates with its environment is simple but powerful:



Our Flux-like Pipeline

Lets take a look how we make such a store by dropping the store listing and People class listing here fully annotated.

```
1  'use strict';
2
3  import Emitter from 'EventEmitter'
4  import _ from 'lodash'
5  import uuid from 'uuid-js'
6
7  class Group {
8    static withTitle(title){
9      return new Group(uuid.create(1).toString(), title, new Date)
10   }
11
12   static fromJSON(raw){
13     var h = JSON.parse(raw)
14     h.date = new Date(h.date)
15     return _.merge(new Group(), h)
16   }
17
18   constructor(id, title, date){
19     this.id = id
```

```
20      this.title = title
21      this.date = date
22      this.tag = "tag-none"
23      this.contacts = {}
24    }
25  }
```

So now we're using ES6, with the new `import` syntax and proper classes, static methods and a constructor. We could have also used default parameters[43], but let's keep it at that for simplicity.

The `Group` class is pretty simple, it holds our data and unlike holding a simple Javascript dictionary for data, we have an option to attach any logic we'd like directly into the `Group` class if it belongs there, which is classic OOP. We also make sure each Group holds a unique global ID, or a UUID, with the standard UUID V1[44], which is sufficient for our case.

We also take a reference to the `EventEmitter` supplied by React Native itself, no need to get one from an external community module.

Moving on to the store.

```
1  class Store {
2    constructor(){
3      this.items = []
4      this.events = new Emitter()
5    }
```

We set up our items and an instance of an emitter. This already states that we'll probably need just a single instance of Store to be traveling along our live objects, since we want everyone to register to the same event emitter, for otherwise it would nullify the purpose of a central dispatch event emitter.

```
1    load(list){
2      this.items = _.map(list, (raw)=>{
3        return Group.fromJSON(raw)
4      })
5      this.sort()
6      this.publish("change", {action: "load"})
7    }
```

This is how we deserialize our store, a raw JSON is supplied externally when the app boots up and sets up every central component such as the

[43] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/default_parameters
[44] https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_1_.28MAC_address_.26_date-time.29

store. So, we simply load that JSON as an array, and let the Group class decide how to deserialize pickled objects to live ones. Finally we sort it just in case we were handed unsorted data, and publish a change event. If you're looking at `_.map` and scratching your head about the _ symbol. Well, that's a regular variable, the lodash library (and before it, the *underscore* library) were very popular when the Backbone Javascript library made its debut (the dawn of single page Web apps) and its purpose was to bring the functional collection handling style to Javascript. So you'll find many functional collection operators such as *map*, *each*, *select*, *reject* and so on.

```
1   list(){
2     return this.items
3   }
4
5   find(id){
6     return _.find(this.items, this._byId(id))
7   }
8
9   new(){
10    return Group.withTitle('Untitled')
11  }
12
13
14  update(item){
15    _.remove(this.items, this._byId(item.id))
16    this.items.push(item)
17    this.sort()
18    this.publish("change", { action: "update", subject: item} )
19  }
20
21
22  remove(item){
23    console.log(`Removing ${item} (${item.id}) from ${this.items} (${_.map(th\
24 is.items, (i)=> i.id)})`)
25    var removed = _.remove(this.items, this._byId(item.id))
26    this.publish("change", { action: "remove", subject: item})
27  }
```

Next are your typical CRUD operations. We manipulate our internal `items` list as a response for any of the `update` or `remove` actions, since these are doing writes, we also publish the appropriate events. In addition we also provide querying with `list` and `find`, where I can *already* tell you that with React you only need `list` - React will know how to find out which of the items have changed for re-render, which is awesome. This is why frameworks like Redux don't bother with fine-grained change events or fine-grained access

to state, they simply hand over the entire store state and let React do the heavy lifting!

```
  sort(){
    this.items.sort((a,b)=>{
      return -1*(b.date - a.date)
    })
  }

  publish(event, ...args){
    this.events.emit(event, ...args)
  }

  _byId(id){
    return function(candidate){
      return candidate.id == id
    }
  }
}

module.exports = Store
```

Remaining are some housekeeping functions for sorting, publishing, and an internal finder function. Note that we export our module the traditional way and not with `export default` since at the time of this writing React Native doesn't support that yet.

Bootstrapping and Navigation

Let's dissect our entry point, in this case `index.ios.js` (which is similar to our `.android` one). At the head of the file, is our bootstrapping code. Here we'll need to initialize our main stores, navigation (we only got one of each through the entire app) and any housekeeping that's needed for bootstrapping the app. As preparation for wiring the navigation routes we import all of the relevant views as well.

```
'use strict';

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

import Master from './views/master'
import Detail from './views/detail'
import People from './views/people'
import Store from './services/store.js'
import Storage from './services/storage.js'
import Icon from 'react-native-vector-icons/FontAwesome'
import Emitter from 'EventEmitter'
import styles from './styles'
```

Import the UI parts that we'll use later in our navigation mapping, and each of our app's building blocks - the *Master*, *Detail*, and *People* smart views as well as our storage, styles, and navigation event emitter.

```
var store = new Store()
var storage = new Storage()
storage.load((err, items)=>{
  if(err){
    console.log("STORAGE: storage load error", err)
    return
  }
  store.load(items)
})

storage.syncWhenStoreChanges(store)
```

Store is our live object store that our app interacts with, and `Storage` is a service component that listens to our store, persisting it when it detects changes automatically for us (we'll review this simple component separately). This is a nice benefit of a central event dispatch - the store doesn't know how to persist itself, nor does it need to care about that. In the end, we end up with our store instance, `store`, which we'll happily plug into our routes as we map our navigation stack.

This is the same store instance that our *Master* view will get, and will subscribe to with the `Subscribable` mixin (I promised we'll tie that one up eventually).

Next up is a rather heavy subject, but its perhaps the most crucial to our application. I've dedicated a complete chapter for an in-depth look of React Native routing and navigation stack, so I hope you'll enjoy that subject on its own.

For our app, this is how the routing and navigation stack look like, dissected bit by bit.

```
1  var navEvents = new Emitter()
2  var _navigator
```

We'll need both of these. `navEvents` is a dispatch event emitter that is responsible to emit navigation events. For example, if someone tapped the 'Save' button and that button exist on the navigation bar, there is no guarantee a view will know about it because by design, our navigation bar, mapping, and routes are separate from any specific view concern. So this is the recommended way of doing things - keep a dedicate event emitter for communicating down navigation bar events. When we get to the `Detail` view, where you can edit a group, we'll see how it interacts with the `navEvents` emitter when it want to save itself. *Tip:* architecturally, you can also tie this communications up with another central entity to your app, can you guess which is it? (hint - your Store. Well, not really a hint :-)

We keep a `_navigator` variable global since as it happens (but you don't see it here yet), Android can navigate in ways that are different than iOS - via the hardware (or software) "Back" button. This is how we can grab our live navigator and wire it through our back button handling logic; this is described in more detail in the Routing and Navigation chapter.

Next up is our navigation and navigation route mapping. First, what's the sense of it all? (you can glimpse at the code listing below and come back here).

- `Navigator` - this is a react component. It is the main entry point for the things React Native needs to know in order to render a fully working and interactive navigation bar: styles, initial route, route rendering, navigation bar component, and within it a route mapper
- A route - is a simple dictionary. It would be best to follow a convention and shape it like below, with a dedicated route `id`, `title` and `props` which we'll pass down to our views directly (for more interesting patterns, jump to the Routing and Navigation chapter)
- An initial route - simply the first route that the app will emit. This in turn will invoke the `renderScene` function which will in turn instruct React Native which actual view component to render
- `renderScene` - a function that maps a route into an actual React Native component, full configured and ready to go. Here you'll typically take route IDs such as `master`, `detail` and perhaps more URI-like such as `app://users/1/carts/2/items` - you make the rules.

- `navigationBar` - with the `Navigator` you can supply any navigation bar component to render your actual UI for the navbar, with interactions included. Here we're taking the default provided navigation bar: `Navigator.NavigationBar` and applying our own `routeMapper` so that it will reflect the routes that are currently displayed meaningfully (more on this on the Routing and Navigation chapter).
- `routeMapper` - a special function that takes a route and reflects the parts that are typical to a navbar: the left button, the title, and the right button.

```
class Navigation extends React.Component {
  render(){
    return(
      <Navigator
        style={styles.container}
        initialRoute={{id:'master', title: "People", props:{ navEvents }}}
        renderScene={this.navigatorRenderScene}
        navigationBar={
          <Navigator.NavigationBar
            routeMapper={NavigationBarRouteMapper}
            style={styles.navBar}
          />
        }
        />
    )
  }
```

Following, this is how our `renderScene` function looks like.

```
  navigatorRenderScene(route, navigator){
     _navigator = navigator
     switch(route.id){
       case 'master':
         return <Master navigator={navigator}
                        store={store}
                        {...route.props}
                        />
       case 'detail':
         return <Detail navigator={navigator}
                        store={store}
                        {...route.props}
                        />
       case 'people':
         return <People navigator={navigator}
```

```
                        store={store}
                        {...route.props}
                        />
      }
    }
}
```

The React Native router and the `renderScene` function lets you completely separate the concerns in your app. The `Master` components has no idea who is the `Detail` components and so on. All of the routing is performed with an abstract `route` object, and this is the part of the stack that converts that abstraction into actual implementation detail. Note that we're passing the navigator and store to the views that need it - typically all views. This is easily DRY'd up with some convenience abstraction you can make but I'm keeping it extra verbatim so it would help to drive the idea at hand. A nice use of the new spread operator[45] is to pass down the props we get through the route.

Finally here is the piece of logic that makes the navigation bar render itself correctly as a reflection of the current route:

```
var NavigationBarRouteMapper = {
  LeftButton: function(route, navigator, index, navState) {
    if (index === 0) {
      return null
    }
    var previousRoute = navState.routeStack[index - 1]
    return (

      <TouchableOpacity
        onPress={() => navigator.pop()}
        style={styles.navBarLeftButton}>
        <Text style={[styles.navBarText, styles.navBarButtonText]}>
          <Icon name="chevron-left"
                size={18}/>
          {previousRoute.title}
        </Text>
      </TouchableOpacity>
    )
  },

  RightButton: function(route, navigator, index, navState) {
     switch(route.id){
       case 'master':
          return (
```

[45] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator

```
25              <TouchableOpacity
26                onPress={() => navigator.push({id: 'detail', title:"New", props\
27 :{ navEvents }}) }
28                style={styles.navBarRightButton}>
29                  <Icon style={[styles.navBarText, styles.navBarButtonText]}
30                        name="plus"
31                        size={18}/>
32              </TouchableOpacity>
33            )
34          case 'detail':
35             return (
36               <TouchableOpacity
37                 onPress={() => route.props.navEvents.emit("save") }
38                 style={styles.navBarRightButton}>
39                 <Text style={[styles.navBarText, styles.navBarButtonText]}>
40                   Save
41                 </Text>
42               </TouchableOpacity>
43             )
44        }
45    },
46    Title: function(route, navigator, index, navState) {
47      return <Text style={[styles.navBarText, styles.navBarTitleText]}>{route.t\
48 itle}</Text>
49    },
50 }
```

This looks a bit messy, since we're spitting out actual UI bits for the right button and title (that could easily be extracted out, of course). But if you squint you'll see that it really is simple; the left button typically just needs to pop an item from the navigation stack, and that is what the code describes, the title needs to pull the `title` key from our route (thankfully we have that ready for use right there!), and the right button need to do work as a function of the route `id`:

- `master` - if we're on the *Master* view, we want to navigate to the *Detail* view for making a new group. So this is a "New" type button and behavior - navigate to detail view.
- `detail` - if we're on the *Detail* view, we probably want to save it. So this means emit a "save" event onto the `navEvents` dispatch emitter. We know that the *Detail* view itself is listening on that event, so it in turn will trigger an actual save (we'll see that later when we dissect the *Detail* view)

This completes the picture of the boostrap, navigation, and *Master* view parts of the app. At this point our *Master* view is primed and our navigation stack will lead us to any other view that we want.

Now, all that is left is to take a look at the so called *Detail* view, or group edit screen, and the *Contacts* screen. This means we'll discuss UI, styling, events, and 3rd party component usage and integration.

Styling

Before we continue, let's take a look at how styling is done with React Native. I think you'll be pleasantly surprised.

React Native styling is done with Javascript, but it feels like CSS. If you noticed the React community at some point in time started moving towards the inline styles trend, and I remember this deck[46] to be an eye opener - I liked the tought process so much that it kept lingering on for a while, and I was happy to see the results in React Native when it finally went out.

In order to create a *style sheet*, or a usable style description, take a look at the following:

```
1  var S = StyleSheet.create
2  module.exports = S({
3    messageText: {
4      fontSize: 17,
5      fontWeight: '500',
6      padding: 15,
7      marginTop: 50,
8      marginLeft: 15,
9    },
10   button: {
11     backgroundColor: 'white',
12     padding: 15,
13     borderBottomWidth: StyleSheet.hairlineWidth,
14     borderBottomColor: '#CDCDCD',
15   },
16       :
17       :
18       .
```

I like to shorthand the `React.create` and `StyleSheet.create` functions as `R` and `S` respectively. But to our point, the create function is passed a dictionary that has the looks of an ordinary CSS stylesheet where dashes are replaced with camel casing. More over, React Native has a much smaller set of properties you can play with than with a regular CSS stack in a modern browser, and for a good reason - it doesn't have to support broken layout models or decades of mishaps due to browser wars.

To find out which properties you can use, check out the official style docs[47].

[46] https://speakerdeck.com/vjeux/react-css-in-js

[47] https://facebook.github.io/react-native/docs/style.html#supported-properties

After you've made your stylesheet, you can export it as with the above listing, and use it as a variable like so:

```
<View style={styles.myCustomStyle} />
```

Where `style` is a special prop React Native uses to apply styles, and it exists in every React Native component that supports styling - you should follow this convention as well in your own custom components.

In addition you can inline your styles like so:

```
<View style={{padding: 10}} />
```

And of course, you can pass a variable instead:

```
// somewhere around your code:
var viewStyle = { padding: 10 }
<View style={viewStyle} />
```

Here, you're providing a raw dictionary, so why even use `StyleSheet.create`? Well, because it will create an immutable and interned table that is more lightweight to pass on renders. For more on this, check out the official style docs[48].

The cascading property of CSS isn't gone either, you can use a limited cascading effect with React Native as well and it makes for a very useful construct. Simply supply an array of styles instead of an object, inline or as a variable:

```
<View style={[{padding:10}, styles.myCustomStyle, {color: "red"}]} />
```

The order of application of the styles (in case there are conflicting definitions) is left to right. So, keep your base styles at the leftmost corner of the array.

Another pleasant surprise is - your stylesheet is Javascript. This means you can do anything a CSS preprocessor such as Sass or Less can do, without all of that technology. You can define and use variables to indicate primary and secondary app colors, a theme for your entire app, compute height, width, and position on the fly, make a stylesheet generator factory and even compose stylesheets in more interesting ways, like plain Javascript objects, if you'd like.

The layout model that React Native implements is Flexbox, while we can chew through pages showing different layouts and how they turn out to be,

[48] https://facebook.github.io/react-native/docs/style.html

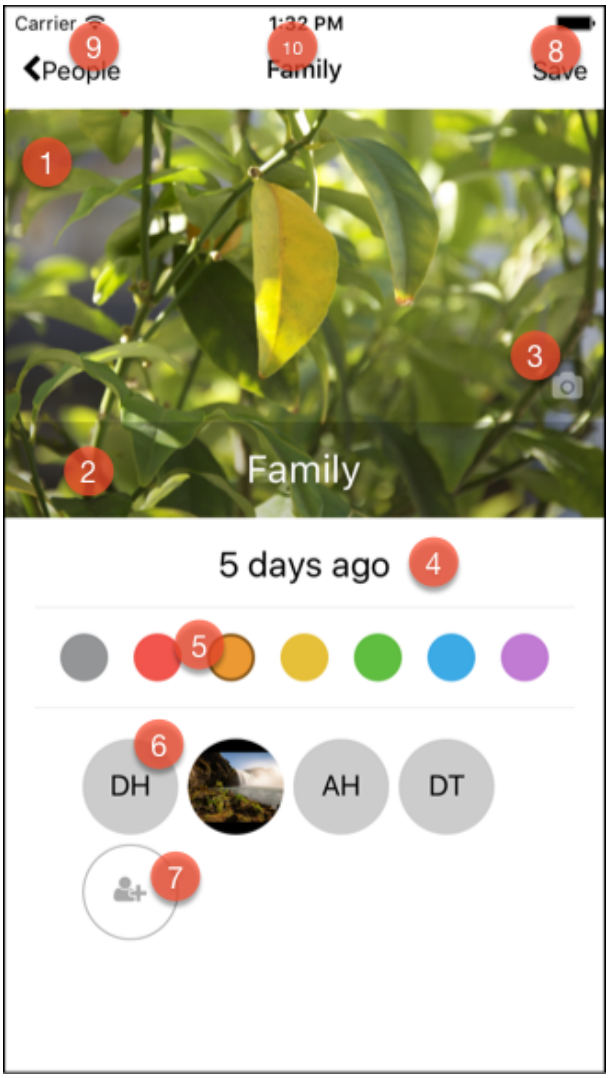
I think it's a waste of time since Flexbox is so well explained else where by now. To get the idea of it, check out this Web based Flexbox playground[49], it is a quick and interactive way to get started. Then you can check out the Flexbox specific properties for React Native on the docs page[50] instead of me just copying and pasting it here and upping the book's page count for no reason :-).

The Detail Screen

Or: the edit group screen. In this part we'll deep dive into more UI oriented topics such as how to compose views, the different React Native UI components, 3rd party components and styling.

Let's start with an annotation of a sample populated view:

[49] http://the-echoplex.net/flexyboxes/
[50] https://facebook.github.io/react-native/docs/flexbox.html



Detail Screen Annotated

1. A cover image holder. This is a regular React Native `Image` component.
2. An editable text box. This is a React Native `Text` component styled to look neat.
3. A `TouchableOpacity` component holds a view inside it and functions as a button. Here we're embedding an `Icon` component from the `react-native-vector-icons` package.
4. A live timestamp, translated to a "time ago" fashion with the familiar *moment.js* Javascript library. This library is commonly used on the Web, but works flawlessly on React Native, as would other plain Javascript libraries.
5. The tags section is a generated collection of views that we embed inside a holder. These are a series of views wrapped with `TouchableOpacity` for interactivity.
6. The contacts section is generated as well, from the description of contacts that are currently associated with this group. We use our own `Avatar` component, which is starring in other views as well.

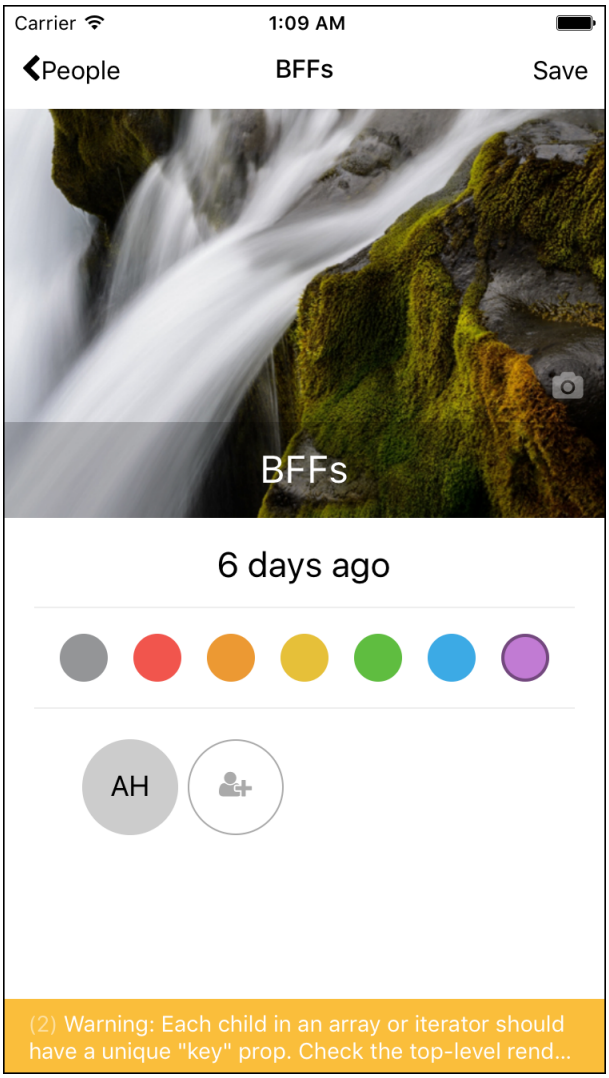
7. The last item in the contacts section looks like it is a contact, but it is really not. It's another view wrapped with `TouchableOpacity` styled to flow smoothly with the other real contact views.
8. The "Save" button is actually not part of this view. It is rendered on the navigation bar (which we reviewed earlier), but we do make sure to receive events from it within the body of the main detail view.
9. Same here, the "back" functionality of this view is really not on it. It is on the navbar.
10. Same again. The title is derived from our route mapper, that we've covered earlier. It's nice to see how it all fits!

Let's start dissecting the *Detail* view, this time we'll start directly with the `render` methods:

```
1  render: function() {
2    let item = this.state.item
3    let contactsViews = _.map(item.contacts, (v, k)=>{
4      return(
5        <Avatar key={k}
6                image={v.thumbnailPath}
7                style={styles.avatar}
8                textStyle={styles.avatarText}
9                firstText={v.firstName}
10               secondText={v.lastName} />
11     )
12   })
```

We're pulling the `item` directly from our state which we modified and kept there and initially got from our props when someone navigated into the *Detail* view. Next we're building the contacts section; this section is simply mapping `item.contacts` into a list of our custom `Avatar` components. It is important to provide a unique `key` as with any collection of views as this helps with React's reconciliation[51]. If you missed this, you'll get a friendly yellow warning from React Native directly in your app:

[51] https://facebook.github.io/react/docs/multiple-components.html



The Yello Box

Try running the sample project and removing the key properties. Then when you get the yellow box, click on it for more information.

P.S. we're also using the ES6 new `let`[52] variable declaration for better scoping semantics.

```
  let tagsViews = _.map(tags, (v, k)=>{
    let selectedStyle = (k == item.tag) ? styles.tagSelected : {}
    return(
      <TouchableOpacity key={k} style={[styles.tag, selectedStyle, {backgroun\
dColor: v.color}]} onPress={()=> this.onTagSelected(k)} />
    )
  })

  let placeHolder = null
```

[52] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/let

```
10    if(!item.image){
11      placeHolder = (
12        <Icon name="picture-o" size={84} style={styles.placeHolder}/>
13      )
14    }
```

Almost the same as with the avatars section, the tags section renders a collection of tag selectors and this is how we render them, keeping in mind that only the currently selected tag (`item.tag`) should be represented as selected. We simply pass down the selected *style* to our `TouchableOpacity` component which serves as the tag body *and* touch gesture responder. This is also a great chance to show off React Native's stylesheet composition; we have a general tag style `styles.tag`, a possible selected style `selectedStyle`, and a specific background color decided by the tag value `v.color`. As always, any interaction with such a generated component is simply deferred to the holding "smart" component.

Next we decide if there's no image provided with our `item`, if true, we build a place holder to cover for the lack of cover image. This would be a huge icon, covering the space of the cover image. Here, we're using `react-native-vector-icons` which you can get here[53]. The idea is to pack every vector icon that is freely available such as Fontawesome and Ionic Framework's Ionicons, as well as Foundation's icons and more. This creates an incredible playground and saves your resources should you want to invest in an icon set or hire a designer. To use these icons you simply provide a `name` and `size` prop. The usual styling holds as well:

```
1  <Icon name="picture-o" size={84} style={styles.placeHolder}/>
```

To install any 3rd party module, as with any node.js based project you would use `npm`:

```
1  $ npm install react-native-vector-icons --save
```

However, since this specific module involves some unusual manual work, such as including the actual font resource files in each of your iOS and Android platforms, it might be useful for you to be familiar with `rnpm`, which is a complementary tool for `npm`, just for React Native. Do this:

```
1  $ npm install rnpm -g
```

And then, after installing React Native module, you can run:

[53] https://www.npmjs.com/package/react-native-vector-icons

```
$ rnpm link
```

Within your project directory. `rnpm` will look for modules with a native requirement such as a binary to link against your app, or a resource to include within either your iOS or Android projects and do that work for you. You should be aware, though, that some times it misses and includes a stray example projects someone have put together with the main project, or files you don't really need, so be sure to take a look at your native project (iOS or Android) should your build suddenly malfunction. It is for this reason I always prefer doing the linking manually and reading the `README` before hand.

Next up, we need to compose everything together:

```
  let ImageComponent = item.image ? Image : View
  return (
    <ScrollView automaticallyAdjustContentInsets={false} contentContainerStyl\
e={styles.container}>
      <ImageComponent source={H.documentsImage(item.image)}  style={styles.im\
age}>
        {placeHolder}
        <View style={styles.toolBar}>
          <TouchableOpacity style={styles.pickImageIcon}  onPress={this.onPic\
kImage}>
            <Icon name='camera' style={styles.pickImageIconText} size={18} />
          </TouchableOpacity>
        </View>

        <TextInput
          style={styles.input}
          onChangeText={this.onTitleChanged}
          value={item.title}
        />
      </ImageComponent>

      <Text style={styles.date}>
        {H.capitalize(H.fromNowInWords(item.date))}
      </Text>

      <View style={styles.tagHolder}>
        {tagsViews}
      </View>

      <View style={styles.avatarHolder}>
        {contactsViews}
        <TouchableOpacity style={styles.avatarAdd} onPress={this.onPeoplePres\
```

```
33  sed}>
34            <Icon style={styles.avatarAddText} name="user-plus" size={18}/>
35          </TouchableOpacity>
36        </View>
37      </ScrollView>
38    )
39  }
```

First, we decide if the main cover component is a live `Image` when we have an image to display or a dud `View` when that image property is null; we call that `ImageComponent`.

Next, we wrap everything with `ScrollView`. The React Native `ScrollView` is quite flexible, and you should take a moment to look at the scroll view docs[54] before we continue. Here we basically compensate for our navigation bar, and specify the container style which is usually where you specify `flex:1` (note that this is not the usual `style` prop).

We compose the `ImageComponent` content. Here, it will be the cover image, the editable title and the camera button. We also do some Flex styling for this component, take a look at this stylesheet, pulled from `styles.js`:

```
1  image:{
2    flexDirection:'column',
3    justifyContent:'flex-end',
4    alignSelf:'stretch',
5    height: 320,
6  },
```

So we're saying content needs to flow as a column, and we also want to stick everything to the bottom with `flex-end`. We're stretching the cover image, and giving it a fixed height.

Next are the time and the tags section:

```
1  <Text style={styles.date}>
2    {H.capitalize(H.fromNowInWords(item.date))}
3  </Text>
4
5  <View style={styles.tagHolder}>
6    {tagsViews}
7  </View>
```

Where `H` is just a shorthand for our helpers, and the goal is to make the date into a "time ago" style text. Following is the tags layout, and here as well we're doing some Flex layouting:

[54] https://facebook.github.io/react-native/docs/scrollview.html

```
tagHolder:{
  marginTop:8,
  marginBottom:8,
  padding:8,
  flexDirection:'row',
  borderTopWidth: 1,
  borderBottomWidth: 1,
  borderColor: '#eee',
},
```

As you can see we're instructing the Flex layout engine to flow the content as a row. Finally, our contacts section is rendered as the following:

```
<View style={styles.avatarHolder}>
  {contactsViews}
  <TouchableOpacity style={styles.avatarAdd} onPress={this.onPeoplePressed}>
    <Icon style={styles.avatarAddText} name="user-plus" size={18}/>
  </TouchableOpacity>
</View>
```

We're dumping the `contactViews` variable we built earlier that contains a list of the contact `Avatar` components, and then rendering our special "add contact" button in the same style of the avatars that came before it. Here, too, you might have guessed, we're making use of Flex, letting the content flow as a row:

```
avatarHolder:{
  flexDirection:'row',
  flexWrap:'wrap',
  width: 300,
  padding: 8,
},
```

That's it for rendering the detail view. As expected this view, being a detail view it is heavier on UI. Next up, let's take a look at the mechanics behind the view, how the data flows and interactions; which is really everything except the `render` function.

This is how our component starts:

```
var Detail = R({
  mixins: [Subscribable.Mixin],

  componentDidMount: function() {
    this.addListenerOn(this.getStore().events, 'change', this.onStoreChanged)
    this.addListenerOn(this.props.navEvents, 'save', this.onSave)
  },

  onStoreChanged: function(event){
    if(event.action == "update"){
      if(event.subject.id == this.state.item.id){
        this.setState({item: event.subject })
      }
    }
  },

  getStore: function(){
    return this.props.store
  },
```

We mix the `Subscribable` mixin, so that we enjoy an automatic removal of listeners safely when the component is unmounting. See more here[55] about Subscribable.

Next we register subscribers on our store (conveniently pulled from our props), and our navigator for the "save" event. Again, these subscriptions will conveniently be removed for us when we unmount because we're going through the Subscribable `addListenerOn` function.

When the store changes, we simply do a sanity check to verify that the item that changed is the one we're holding. If it is, we simply set state and React takes care of the rest.

Let's see how events are handled. In the below listing, we need to handle *save*, *remove*, *title change*, *pick image*, and *tag selected*.

```
  onSave: function(){
    let item = this.state.item
    this.getStore().update(item)
    this.props.navigator.pop()
  },
```

This event comes from our `navEvents` event dispatcher that we saw during our discussion of the bootstrap and navigation code earlier. Here we simply get our store and tell it to update the item. More over, once saved we want to go back to the *Master* view or which ever view that called us - so we *pop* the navigation stack, as simple as that!

[55] https://github.com/facebook/react-native/blob/master/Libraries/Components/Subscribable.js

```
  onTitleChanged: function(text){
    let item = this.state.item
    item.title = text
    this.setState({item: item})
  },
```

This is a simple handler for when the title changed and it is bound to our `Text` component. We only modify the state, and when the use hits "Save" we'll commit everything to our store.

Next up, let's see how we're picking an image. For this we use a 3rd party module called `react-native-image-picker`. To install and link it we'll do:

```
$ npm install react-native-image-picker --save && rnpm link
```

Now for the handler:

```
  onPickImage: function(){
    UIImagePickerManager.showImagePicker(pickerOptions, (response) => {
      console.log(`DETAIL: picked image, response `, response)
      if (!response.didCancel) {
        this.state.item.image = {uri: response.uri}
        this.setState({
          item: this.state.item
        })
      }
    })
  },
```

Since this is a native module, it isn't really a UI component we include, but something that looks like a system service. `UIImagePickerManager` is that module and we tell it to display the picker interface using our `pickerOptions` (take a look here[56] for these options) and we get a suitable response only after the use have picked an image (or cancelled). If the user didn't cancel we plug the `response.uri` into our image property within `item`, which we'll later directly inject into a React Native `Image` component. We then force an update with `setState`.

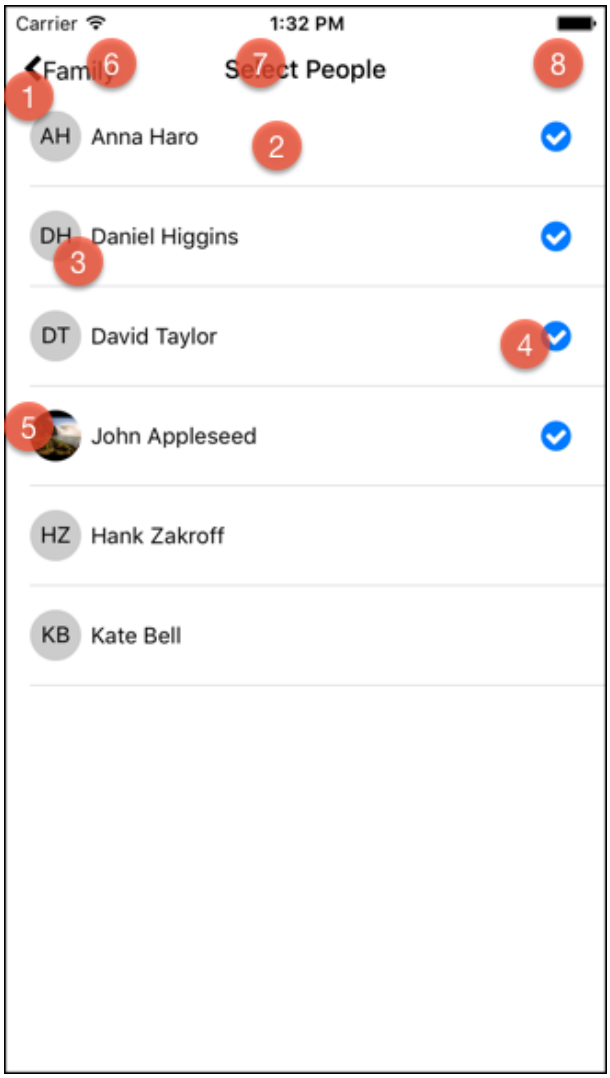
[56] https://github.com/marcshilling/react-native-image-picker

```
  onTagSelected:function(tag){
    let item = this.state.item
    item.tag = tag
    this.setState({item:item})
  },
  onPeoplePressed:function(){
    this.getStore().update(this.state.item)
    this.props.navigator.push({
      id: 'people',
      title: "Select People",
      props: {
        itemId: this.state.item.id,
      },
    })
  },
```

Both of the remaining handlers deal with the contacts or tags selection. However when a contact is selected (`onPeoplePressed`) we navigate to a completely different screen - the People screen, which will list out the contacts on the user's phone. If you squint, you can tell that the `push` function actually pushes a `route` onto the navigator, which gets fed back to our navigation and route mapping code. Correct or not, from a product perspective, we save the changes the user made so far before navigating out to the new screen.

The People (Contacts) Screen

The last "smart component" we're going to dissect is the contacts screen, where a user is able to pick the contacts that should belong to a given group. Let's take a look now.



Contacts Screen

1. As before with our *Master* view, this is a `ListView`
2. And again, each `ListView` holds a prototype cell, and this is our cell
3. Within each cell, we have our `Avatar` component again, playing a different role but still the same reusable component
4. We render a check mark with our `Icon` component for items that are selected, the entire list is refreshed and rebuilt so that the selected contacts are always on top.
5. Avatars can either hold an image, or if missing, display a first letter from each part of the contact name (first and last).
6. The navbar can go back to the group we came from, this is done automatically for us by our navigation mapping
7. Same for the title
8. And no right button here, we have no were else to go

Let's start going over the code for the contacts screen, this will again be a Master-Detail pattern, so we'll start with the *Master* part. Let's do this upside down as well, starting at the `render` function.

```
  render: function() {
    return (
      <ListView style={{paddingTop: 50, flex:1}}
        dataSource={this.state.contacts}
        renderRow={this.renderRow}
      />
    );
  }
```

Nothing new here, our data source is now the more properly named `contacts` (if you'd like a review of `ListView` and data sources, refer back to the *Master* view dissection). Let's continue on to our `renderRow` function.

```
  renderRow:function(row){
    return (
      <Cell key={this.state.item.id} item={this.state.item} contact={row} onP\
ress={()=> this.didSelectRow(row) }/>
    )
  },
```

Good news here, this also looks the same. A master-detail pattern would probably always look the same, like this. You're thinking what I'm thinking? Yes. You could make a `MaterComponent` that embeds this pattern, giving it just two things:

```
// Imaginary MasterComponent
class People extends MasterComponent{
  didMountListView(){
    return {
      data: contacts,
      row: (item)=> <Cell .../>
    }
  }
  // Now just handle regular smart components events.
}
```

This makes sense, because iOS has embedded a pattern like this with UITableViewController[57] as well, since this infrastructure doesn't really exist for React Native yet, I'll leave this as homework :-).

Let's see what happens when we mount this view, specifically, we need to fetch contacts.

[57] https://developer.apple.com/library/ios/documentation/UIKit/Reference/UITableViewController_Class/

```
  componentDidMount: function() {
    this.addListenerOn(this.getStore().events, 'change', this.onStoreChanged)

    Contacts.getAll((err, contacts) => {
      console.log("Contacts", contacts)
      _.each(contacts, (contact)=>{
        contact.firstName = emo(contact.givenName || contact.firstName || "")
        contact.lastName = emo(contact.familyName || contact.lastName || "")
      })
      if(err && err.type === 'permissionDenied'){
        console.log("PEOPLE: no contacts permissions")
      }
      else{
        contacts.sort(this.sortBySelectionAndName)
        this.setState({contactsArray: contacts, contacts: this.state.contacts\
.cloneWithRows(contacts)})
      }
    })

  },
```

First, we add a listener on the store. Remember the store have been passed to this component by our navigation stack. Next, we use the `Contacts` module, which is a 3rd party npm module we use to get a hold of each platform's contacts facility. You might imagine that not only iOS and Android don't share the same API for retrieving contacts, but the data model and the naming things might be different as well. This is where `react-native-contacts` fills the gap. To install it:

```
$ npm install react-native-contacts --save && rnpm link
```

Now you should get at an API which looks like this[58] in your regular Javascript code. If you're wondering how this magic is done, we cover how to make native modules, native UI components and more in this book (and at the time of writing, that might be the only place to read about it other than the official docs, which unfortunately provide a partial coverage).

Moving on, a `getAll` fetches all of the contacts for the device, and hands out nifty Javascript dictionaries with populated fields about every contact. We strip out *emoji* because we'd like to keep it business (you can leave as is and keep the emoji), using the `emoji-strip` npm library, and that's a completely plain Javascript library that works out of the box with React Native. After fixing up the contacts, we sort them and set them to our state.

Next up we handle `onStoreChanged` by refreshing our contacts in case the group itself changed, but this is just a safeguard since this scenario doesn't really exist. We also handle row selection like so:

[58] https://github.com/rt2zz/react-native-contacts

```
  didSelectRow:function(row){
    var item = this.state.item
    var rowId = row.recordID
    if(item.contacts[rowId]){
       delete item.contacts[rowId]
    }else {
      item.contacts[row.recordID] = row
    }
    this.getStore().update(item)
  },
```

This basically means we either insert that contact's ID into our group or remove it completely. Either way we push back our changes to our store.

Let's move on to our list view cell. It so happens that our cell is a functional component[59]. I love functional (or, "pure") components because they make a *pitfall of success*; they force you into the right way of doing things without side effects, leading to lighter weight and more composable components.

```
module.exports = (props) =>{
    let icon = null
    let contact = props.contact
    if(props.item.contacts[contact.recordID]){
      icon = <Icon style={styles.check} name='check-circle' size={22} />
    }

    let firstname = contact.firstName
    let lastname = contact.lastName
    let avatar = <Avatar
                    firstText={firstname}
                    secondText={lastname}
                    image={contact.thumbnailPath}
                    textStyle={styles.avatarText}
                    style={styles.avatar}
                    />

    return (
      <ListItem onPress={props.onPress}>
        <View style={styles.holder}>
          <View style={styles.avatarHolder}>
            {avatar}
          </View>
          <View style={styles.content}>
            <Text style={styles.text}>{firstname} {lastname}</Text>
          </View>
```

[59] https://facebook.github.io/react/blog/2015/10/07/react-v0.14.html#stateless-functional-components

```
27            {icon}
28          </View>
29        </ListItem>
30      )
31  }
```

In this view, we're composing a list item. We're starting out with the decision – do we or do we not draw a check sign? The check sign itself is rendered with our all time favorite `react-native-vector-icons` module, and the `Icon` component. Next we're composing the `Avatar` with everything that it needs from our contact, and finally the `ListItem` itself, and remember, being a "dumb view" we're simply firing off handlers that were passed to us through our props.

The completes our review of the entire app screens. Next, you might have noticed we're using components such as `Icon` and `Avatar` all over in many different situations, which is great, because we're reusing components. One of these components, `Avatar` is one we built for ourselves. Let's see how make sure a reusable component next.

Making Reusable Components

During our review, we bumped into `Avatar` several times. It's now times to see how it works, and what's the reasoning behind making such a reusable component.

First let's describe it. The avatar will:

- Render an image when it exists, and apply general styles to itself, and specific image styles if exists.
- Render a synthetic "image" which is composing the first letter and last letter of the contact's full name (which is quite common on mobile phones), and then apply specific textual styles to that as well as general avatar styles.

From this definition we arrive at three different sets of styles:

1. Component style
2. Text avatar style
3. Image avatar style

We'll take these as `style` (no surprise here), `textStyle` and `imageStyle`. we will have to wire each of the last two on top of `style`, but that's easy because we already saw React Native styles are easily composable.

Now that we're zeroed out on the logic, let's take a look at the code:

```
class Avatar extends React.Component{
  constructor(props){
    super(props)
    _.map(Object.getOwnPropertyNames(Object.getPrototypeOf(this)),
          (x)=>{
            if(x.match(/^__[a-z]/)){
              console.log(x)
              this[x] = this[x].bind(this)
            }
          })
  }
  __signature(first="",second=""){
    return `${(first[0] || "?").toUpperCase()}${second && second.length > 0 ?\
 second[0].toUpperCase() : ""}`
  }
  render(){
    var avatar = null
    if(this.props.image){
      avatar = <Image source={{ uri: this.props.image }}
                      style={[styles.image, this.props.style, this.props.imag\
eStyle]} />
    }else{
      avatar = (<View style={[styles.view, this.props.style]}>
                  <Text style={this.props.textStyle}>
                    {this.__signature(this.props.firstText, this.props.second\
Text)}
                  </Text>
                </View>)
    }
    return avatar
  }
}
```

First, there's a super strange block of code at the constructor isn't there? This is some generic code you can use to automatically bind class methods' `this`. With Javascript you'll find some semantics are sort of broken, like the treatment of `this`. While most of that is being fixed with ES6 and probably ES7, there is still some *impedence mismatch* when you want to use the new style of React components (class) and not the old style (React.createComponent); see more here[60]. So here, we're telling the component to bind any method that starts with __, by convention (our convention). This is just to show you how you can save up on that boilerplate code; you can adopt that approach if you like.

Next, we compose the view with toggling between our `Image` or `Text` based

[60] http://egorsmirnov.me/2015/08/16/react-and-es6-part3.html

view. Take note of the style composition, in case of `Image`:

```
1 style={[styles.image, this.props.style, this.props.imageStyle]}
```

And the `Text` based view:

```
1   avatar = (<View style={[styles.view, this.props.style]}>
2             <Text style={this.props.textStyle}>
```

We are also keeping the styles in-line in the same file so that our component is truly stand-alone and can be self-contained. It also makes sense to use as few external libraries as possible (unlike what we did here, by including `lodash` - but that's just a convenience and can be easily replaced with plain Javascript).

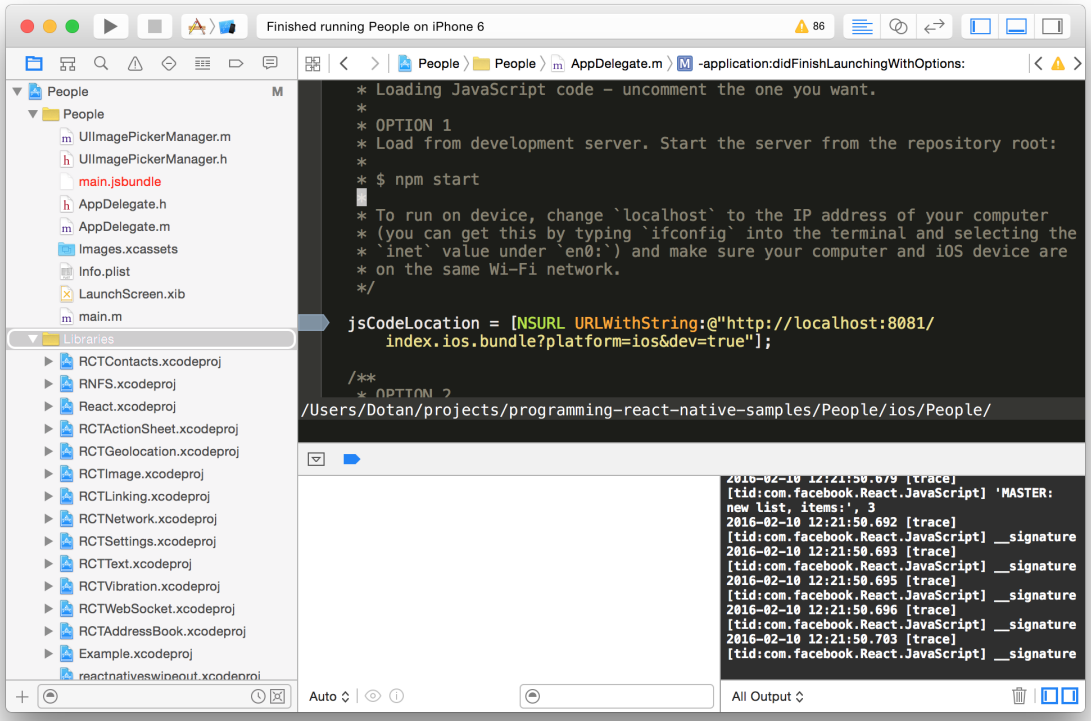
As homework, try converting the `Avatar` component into a pure functional component.

Using Community Components

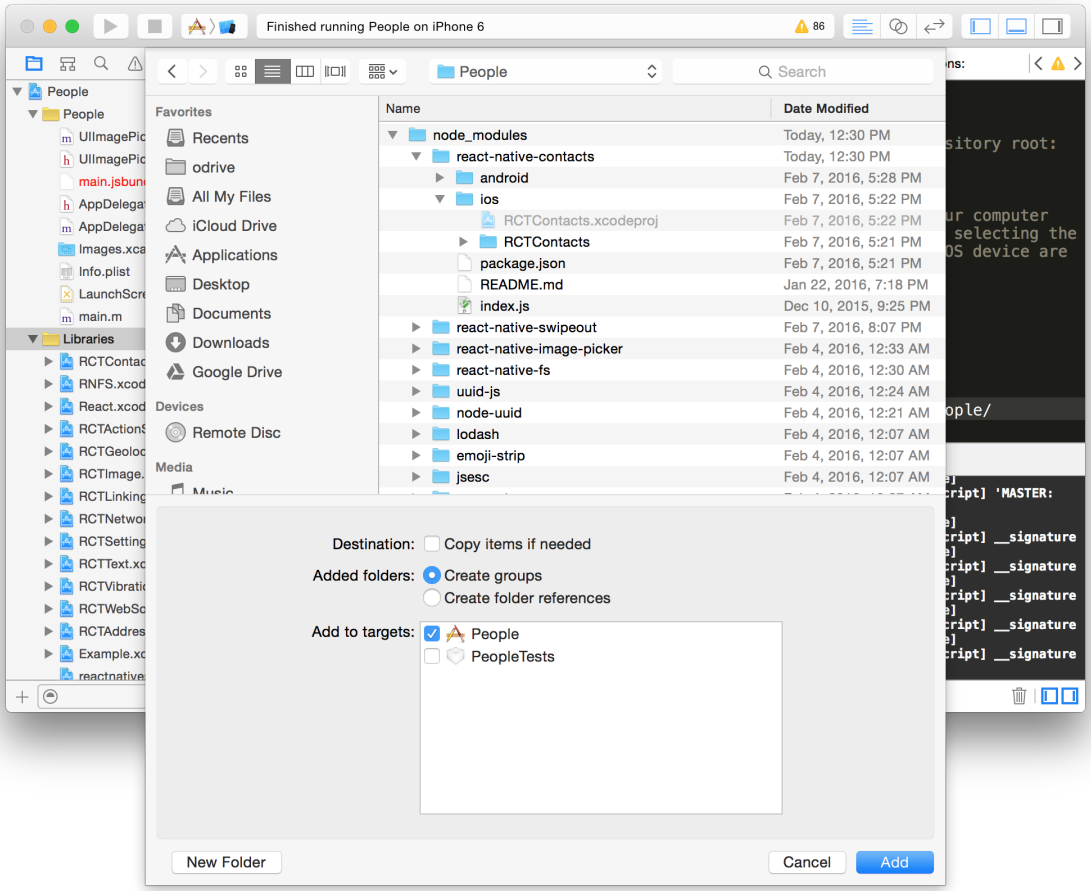
So far we've used community component pretty easily with `npm` to install these and `rnpm` to link them up. While `rnpm` is useful (and very new at the moment), it is important to see what you need to do in order to "link" components to your iOS and Android projects. Hopefully in the future, `rnpm` will be so bullet proof that you won't need that knowledge at all.

Linking iOS projects

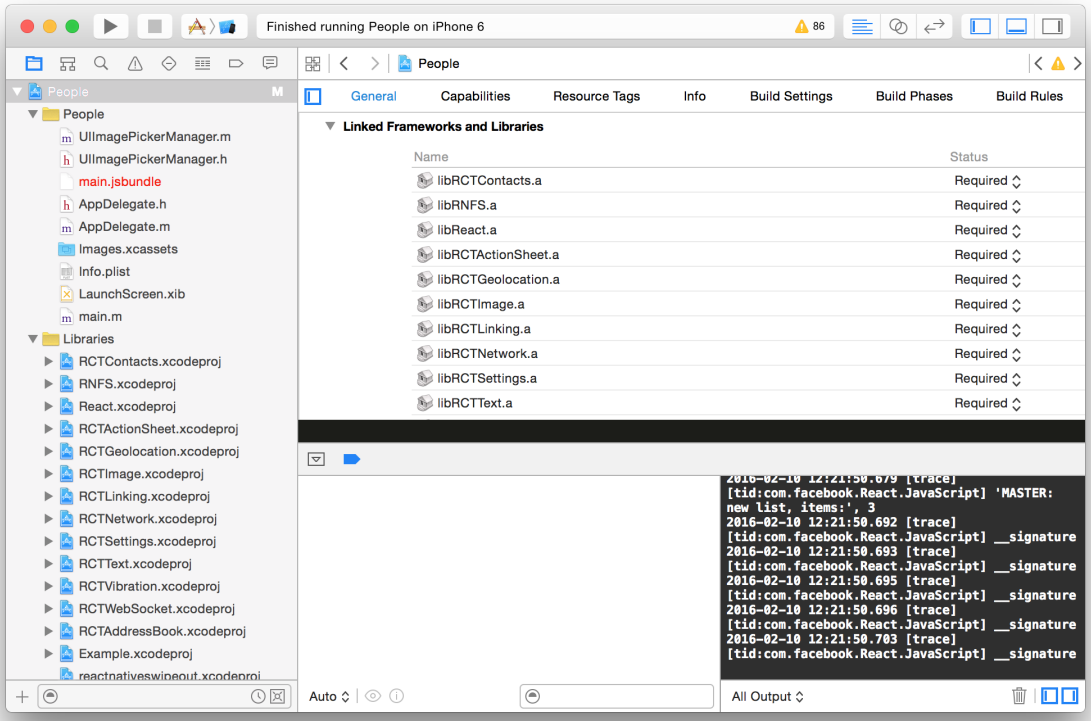
Some components have a native part to them, and that needs to be linked against your native project. For iOS these are the steps you need to follow after you install that component with `npm`:



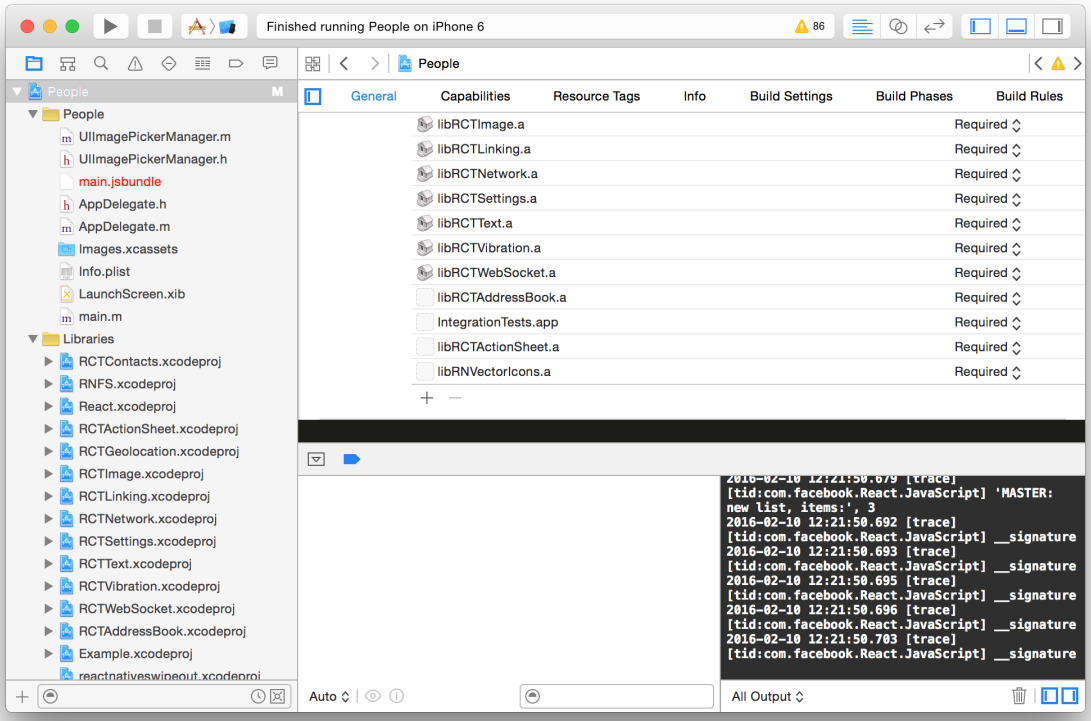
Locate your *Libraries* group within your project outline, and pop the context menu with right-clicking it. Then select “Add New File”.



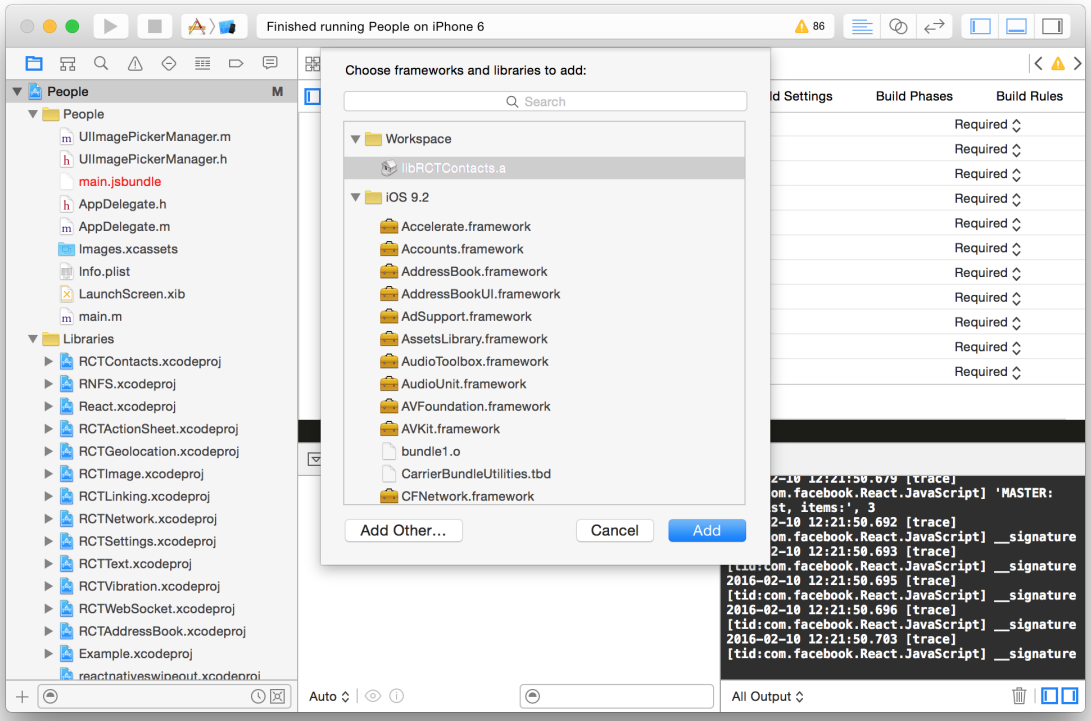
In your project source tree, find your main `node_modules` folder, and drill down to the library that you want to link. In our example we'll be linking the `react-native-contacts` library. You need to find the iOS Xcode project (ends with `.xcodeproj`). Once found, select it.



Next, go to your project build settings, in the “General” tab. Scroll downwards and find the “Linked Frameworks and Libraries” section.



At the bottom of this section you'll find a "+" button. Click that.

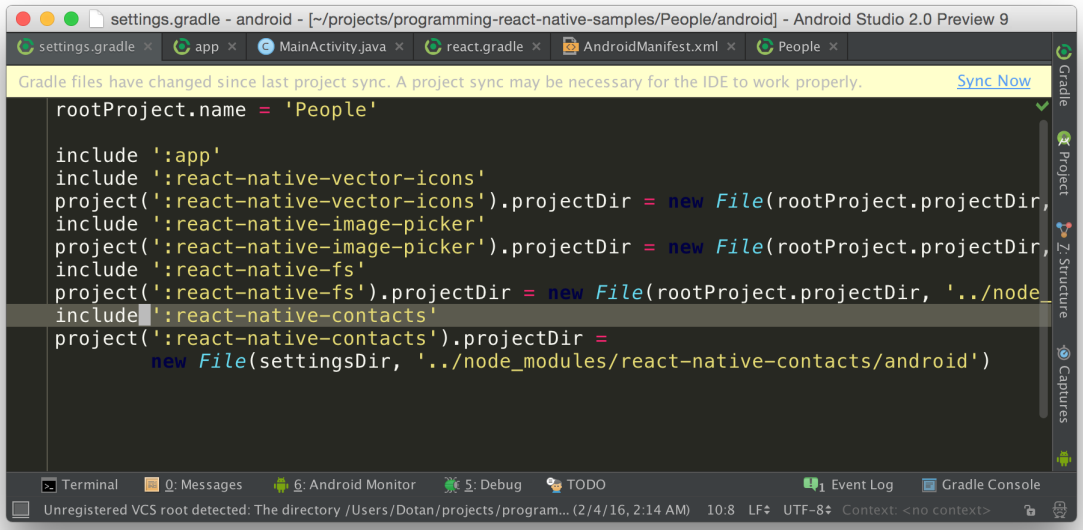


You'll need to find the `.a` library to link, it usually will appear first, and libraries that are not linked are listed there, so it should be very obvious. Here we're seeing the `libRCTContacts.a` binary that we need to add.

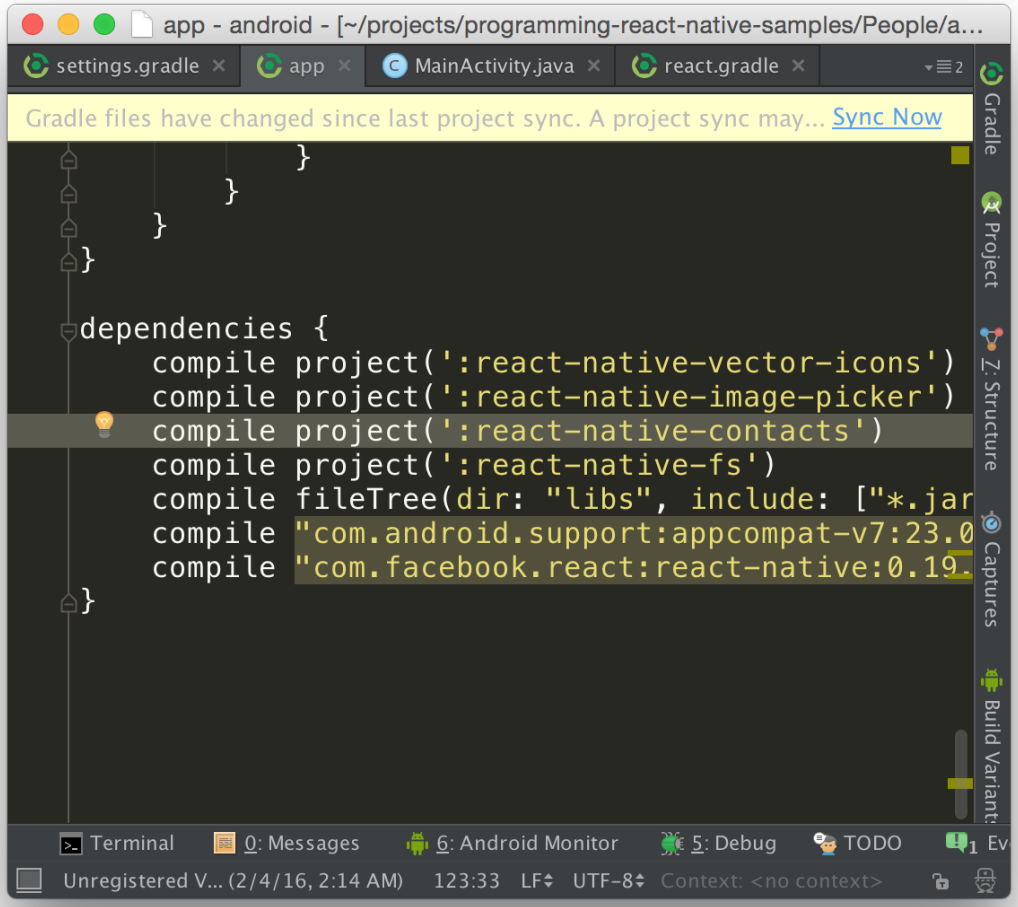
That's basically it. Build to make sure everything passes successfully.

Linking Android Projects

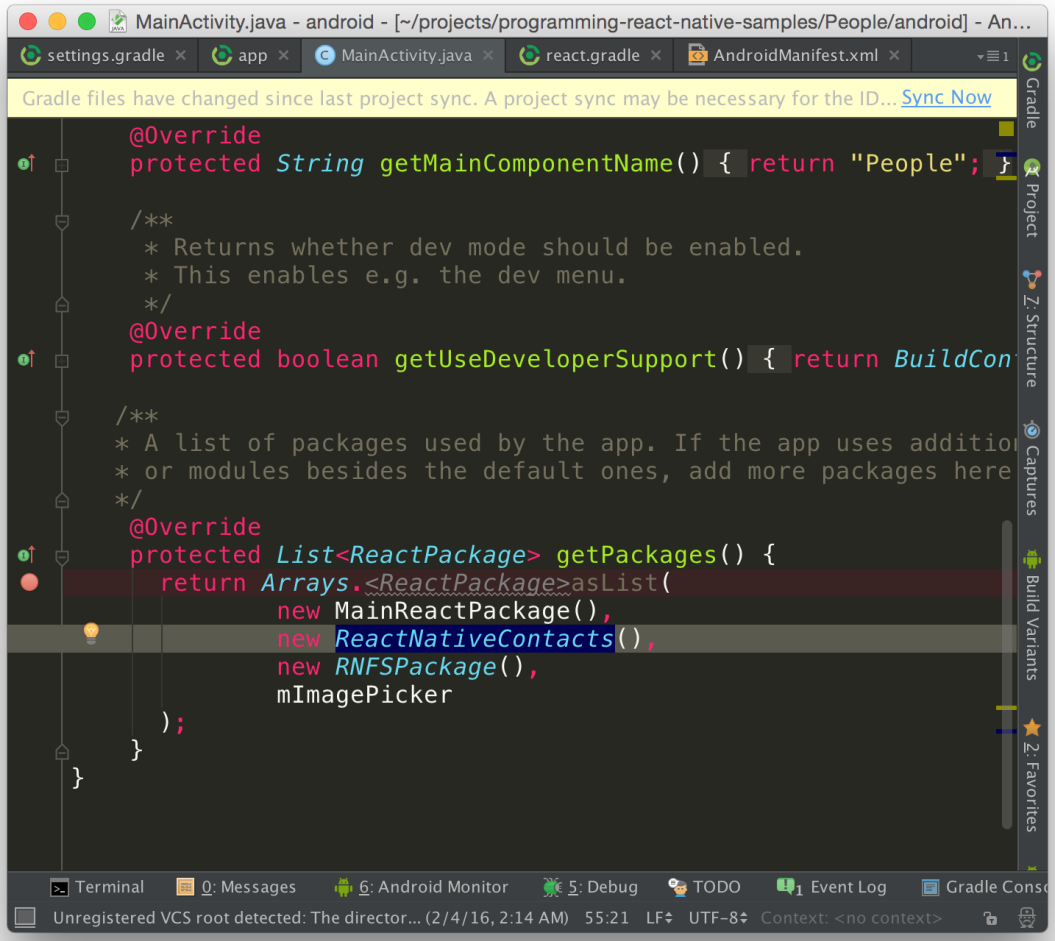
Luckily, Android adopted Gradle a couple years back. This makes life very simple for us.



Locate the `settings.gradle` file at the root of your project, and add the above lines to it. We're basically saying there's a new software project, and its location is somewhere within `node_modules`; you'll have to locate that manually. Here, again we're linking the `react-native-contacts` module.



Next find your `app/build.gradle` file, and declare that project as a compilation source. This is how Gradle takes dependencies and knows to include a subproject's source into the main compiled target. Now, the contacts native code can be visible within our own.



Register the package (you can read more on packages and native modules in the dedicate native modules chapter) within the main (and only) activity like the above. You'll need to include any missing Java imports (Intellij/Android Studio will happily request to do this for you).

That's it, build and make sure everything passes.

Javascript Components

There's nothing to link here as almost every project that's on `npm` will be available to you. It's a good idea to read the entire project's `README` to try to find out if there's any native part you need to know. At the time of writing, there is no standard way of declaring:

- If the project has native parts
- If it supports both iOS and Android, or just one of these

- If there are any additional steps to do (such as including resources, as in the case of `react-native-vector-icons`)

And again, I hope `rnpm` is destined to change that and make these concerns transparent to you, however without a standardized project layout - it is wise as me and you, or less, about any given project.

Another concern when evaluating Javascript components, is to take into account the risk that it uses some Node.js functionality that isn't available with the React Native engine (try using the popular `node-uuid` to generate UUIDs instead of `uuid-js` the we used). In this case just find the pure Javascript variant of the library.

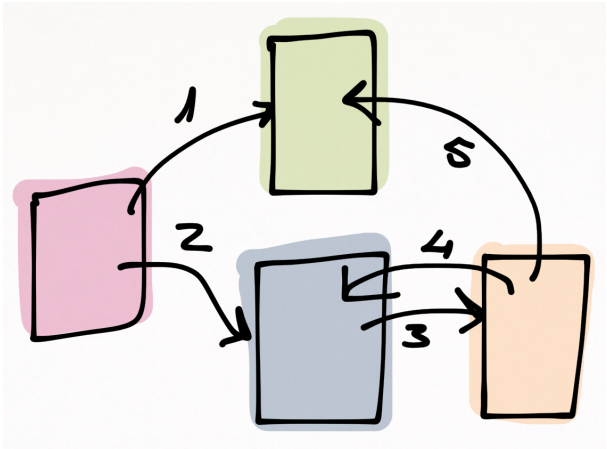
All in all I've briefed you on how to find those edge cases that will for sure be a time wasters, however you should know that most libraries work well without any special treatment. And of course there's the convention of a library that is prefixed with `react-native-`. You can find many of those and more in my awesome-react-native[61] repository on Github.

Summary

This has been our most intensive, action packed chapter yet. In it, you've learned about a powerhouse pattern in mobile applications (Master-Detail), and you've seen how to build your own smart components, dumb components. We've also learned about bootstrapping your app in your entry file and how central concerns like navigation and stores come into play. We've also seen how to make reusable components and use community components; this richness is directly driven by React being such a great framework for composition, and the ecosystem it enabled to create. We also got to know the tooling, the IDEs, `npm` and `rnpm`. We definitely covered a lot, and you may be surprised that the following chapters are not lowering the bar; we will deep dive into topics that are notoriously difficult for mobile app development.

[61] https://github.com/jondot/awesome-react-native

Navigation and Routing



Why navigation

You might ask yourself: why do you need navigation? What's a navigation stack? Well, navigation on mobile apps covers these concerns:

- Situational awareness: know where you're at, and possibly, where you came from
- Functional: be able to navigate. Go back, undo, skip, or "deep link" into some functionality of the app
- Infrastructural: consolidate ceremonies that need to be done before rendering a new screen.
- Maintainability: to realize the above concern, often, you'll need to build a state machine. A navigation stack is a state machine. Platforms may let you code this state machine declaratively or visually, and promote maintainability by keeping all of that logic in one place.

Why Navigation is Scary

In my opinion, one of the scariest topics in mobile application development is routing and navigation.

Navigation infrastructure typically holds a navigation stack, which holds screens, which hold components and data. Often, this state is not represented anywhere; it is a transient state that the user have created by merely navigating around, which is why it makes it hard to reason about.

Navigation flows are also hard to reproduce, should you bump into bugs in that area, and often these bugs carry memory leaks and the sorts (we did just mention a navigator stack holds reference to screens, which hold reference to components, which holds reference to data, and so on).

Navigation UX deals with how each operating system brought with it its own truth as to how to properly do navigation, as well as how to facilitate it: Android's implicit back button behavior, and iOS with its purely explicit text navigation bar.

And lastly, each platform has its own tooling and how to ease up all said pains. Android has the traditional approach, where you build screens and code transitions with the omnipotent `Intent`, and lately iOS offered a more intuitive approach with Storyboards and almost physically connecting screens.

Navigation in React Native

The bad news is that navigation doesn't get less scary to me, even with React Native. I guess that for me the saying "the more you know, the more you worry" painfully applies.

However, the good news, or maybe the great news is - that with the React, Javascript, and React Native combo everything becomes amazingly more predictable.

- React sports one-way binding, which helps us reason about our code. So to find memory leaks we travel a one-directional dependency tree (YMMV).
- Javascript brings about the Chrome Devtools, which by now are one of the most powerful toolsets for developers. If you're going to build something complex, you're lucky to be living in 2015 (at least, that's when this book was written, consider yourself luckier if you're from the future!).
- React Native gives us plenty of escape hatches. This is the case where we want to use `NavigatorIOS`, the native iOS navigation stack, in order to minimize the element of surprise by not emulating a navigation stack, or simply put - sticking to the host OS best practices.

In addition, if what you're building is really painfully complex (years of effort and complexity), I guess at this stage you probably don't really need this book and should build your app on each platform separately using dedicated tooling and approaches, just to be safe.

Let's take a look at the two React Native navigation solutions.

Navigator vs. NavigatorIOS

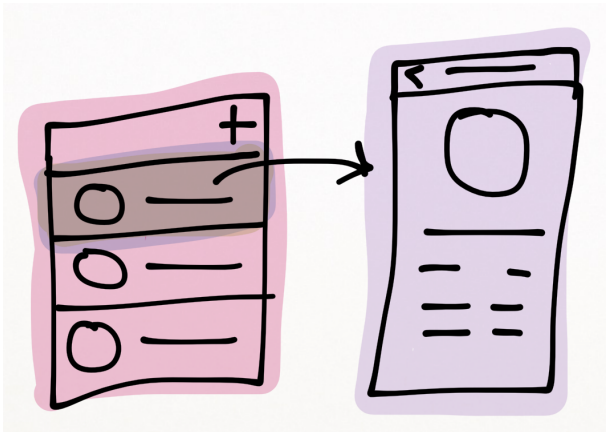
React Native started on iOS, and this is why the best-in-class or perhaps most used navigation stack is `NavigatorIOS`. If you're building an iOS only app and want to play it safe, use it. However, if a year or more passed from the writing of the book, it might be possible that the generic `Navigator` dwarfed `NavigatorIOS` in features and stability, so allow yourself to evaluate that again by Googling around.

`Navigator` is React Native's attempt to abstract the concept of navigation. At the time of writing, it is good-but-not-great a bet for a smooth ride if you want to keep a common navigation codebase for iOS and Android.

We'll use `NavigatorIOS` for iOS and `Navigator` for Android. Even if reality changes and `Navigator` becomes the be-all-end-all navigation solution for both platforms, I would want to keep it apart. It feels that navigation on iOS and Android will always want to be different, somehow, either UX or hardware-wise, so it makes sense to future-proof this and make room for code to evolve differently.

Navigator

Let's go through a brief `Navigator` example. For this, we'll assume the common master-detail pattern, where we have a master view containing a list of items, and then when tapping an item we want to navigate to a child (or detail) item.



Wiring Navigator

This is what you probably wanted to read before I filled your head with back button craze.

The Navigator is a React component that deals with two main concerns:

1. Returning a properly configured `Navigator` component, with icon, back icon, colors, initial route and more. In addition, it should wire a `renderScene` handler, which we talk about next.
2. A renderScene handler implementation, which is a function that takes care of mapping a route (which just a Javascript object) to a screen of our choosing. The goal is similar in concept to backend routing with frameworks like Express or Rails, which deals with separating routing from destination.

Here is how we set that up, for a simple two-screen ("first" to "second") navigation flow:

```
class Navigation extends React.Component{
  render() {
    return (
      <Navigator
        style={styles.container}
        initialRoute={{id: 'first'}}
        renderScene={this.navigatorRenderScene}/>
    );
  }

  navigatorRenderScene(route, navigator) {
    _navigator = navigator;
    switch (route.id) {
      case 'first':
        return (<First navigator={navigator} title="first"/>);
      case 'second':
        return (<Second navigator={navigator} title="second" />);
    }
  }
}
```

In the `render` function, we set up a `Navigator`, in which the most important properties to specify are the `initialRoute` and `renderScene`. Note that a *route* is simply any Javascript object that will be carried from screen to screen. Here, we make a convention to have our main routing concern exist behind an `id` property; if we have anything else to pass, we'll use a suitable payload, within the same object. However, let's agree that `id` will always be reserved for routing.

The good news is that we passed the clunky part of declaring a `Navigator` component and then wiring up screens. It *can* become less clunky, maybe, some day. It *can* have some sort of a DSLish feel to it such as:

```
// This doesn't really exist, but you can make it
var Navigation = routes({
  'first': (route, navigator)=>{
      <First title="foobar">
  }
})
```

But this kind of experience doesn't exist yet. It's quite a low-hanging fruit, so think of it as a nice weekend project to do :)

ToolbarAndroid and Navigator.NavigationBar

It is important to notice that `Navigator` comes with a pre-baked piece of UI that functions as your top Android navigation bar as well as the iOS navigation bar. If you'll dig into UIExplorer[62] within the `facebook/react-native` Github repo, you'll see these kinds of things:

```
var NavigationBarRouteMapper = {

  LeftButton: function(route, navigator, index, navState) {
    if (index === 0) {
      return null;
    }

    var previousRoute = navState.routeStack[index - 1];
    return (
      ...
    );
  },

  RightButton: function(route, navigator, index, navState) {
    return (
      ...
    );
  },

  Title: function(route, navigator, index, navState) {
    return (
      ...
    );
  },

};
```

[62] https://github.com/facebook/react-native/tree/master/Examples/UIExplorer/Navigator

That is a concept of a `NavigationBarRouteMapper` or in other words, you're telling your `Navigator` how to compose the `NavigationBar`'s UI on each and every route.

Then, you stick an actual `NavigationBar` into your `Navigator` this way:

```
<Navigator
      ...
      navigationBar={
        <Navigator.NavigationBar
          routeMapper={NavigationBarRouteMapper}
        />
      }
```

So what you've got here, is a `Navigator` which we learned how to configure and tell it how to render itself as a response to a given route, and a `NavigationBar` that pretty much does the same.

Immediately, you start thinking - why separate the two? Your `NavigationBar` and `Navigator` rendering is spread out through two different mappers, and also, why not make a route a first-class citizen and making itself render? something like `ShowCartRoute.render()`, and then `ShowCartRoute.renderBar()`? well, this is what `Exponentjs/ex-navigator` solves exactly, and more or less feels like. I recommend checking it out[63]. Also, you might imagine that this is not going to be the only flavor people would like to do their routing with, so keep watching out for new things.

Should you use a `ToolbarAndroid` or a `Navigator`'s own `NavigationBar`? The answer is again - tradeoff. The `NavigationBar` is strongly tied to routes, `ToolbarAndroid` is tied to your view. `ToolbarAndroid` is, well, Android, and you could probably implement such a thing yourself generically.

So bottom line, if you're implementing something simple, then go with iOS's `NavigatorIOS` for iOS (coming in the next section), and `ToolbarAndroid` for Android. Otherwise, use a `Navigator` for both, and either `NavigationBar`, or your own piece of bar that you construct manually. And, of course, do a quick browse on my awesome-react-native[64] list for any fancier bar that you'd like.

ToolbarAndroid

So `Navigator` and `NavigationBar` aside, let's give `ToolbarAndroid` a decent coverage in case you'd pick this option. In that case our screens will be normal views that take care of massaging a `ToolbarAndroid` subview to their needs.

[63] https://github.com/exponentjs/ex-navigator
[64] https://github.com/jondot/awesome-react-native

A `ToolbarAndroid` is our top navigation bar, and with React Native, it is quite flexible. So flexible, in fact, that we need to code every decision point such as when to show the back icon, what kind of title to display and so on, based on our current and past screens.

A `ToolbarAndroid` is also the real Android `Toolbar` widget (see here[65]) which, in accordance to our theme here (showing the platform-specific way first) is an Android-specific component.

Flexibility can be good and bad. Good since on each screen, we can specify exactly how we want things to be had on the navbar. Bad, because we need to be defensive here since it can become a maintenance overhead or a state machine of the worst kind - the one that is spread out through out our entire codebase.

Here is our `Second` screen:

```
1  class Second extends React.Component{
2    render() {
3      return (
4        <View style={styles.container}>
5          <ToolbarAndroid style={styles.toolbar}
6                          title={this.props.title}
7                          navIcon={require('image!ic_arrow_back_white_24dp')}
8                          onIconClicked={this.props.navigator.pop}
9                          titleColor={'#FFFFFF'}/>
10         <Text>
11           Second screen
12         </Text>
13       </View>
14     );
15   }
16 };
```

If you're trying this out, make sure that `ic_arrow_back_white_24dp` is an icon you've dropped in your resources folder - in this case Android. For the sake of the experiment you can use a single hi-res image for all size variants.

Note here, that we specify our navIcon explicitly. We want users to be able to tap a back icon right on the navbar.

Next up, our `First` screen:

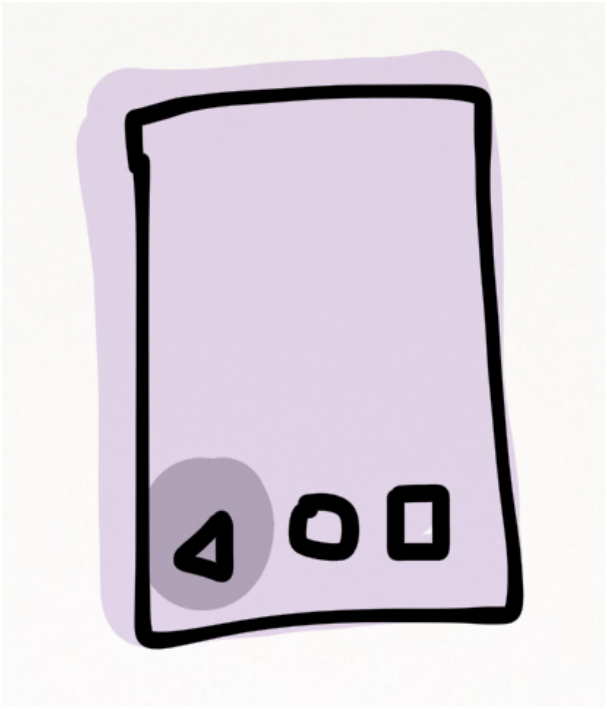
[65] https://developer.android.com/intl/zh-tw/reference/android/support/v7/widget/Toolbar.html

```
class First extends React.Component{
  navSecond(){
    this.props.navigator.push({
      id: 'second'
    })
  }
  render() {
    return (
      <View style={styles.container}>
        <ToolbarAndroid style={styles.toolbar}
                        title={this.props.title}
                        titleColor={'#FFFFFF'}/>
        <TouchableHighlight onPress={this.navSecond.bind(this)}>
          <Text>Navigate to second screen</Text>
        </TouchableHighlight>
      </View>
    );
  }
}
```

Here, we don't specify a back button, because we recognize it as the root screen. The most important (and fun) piece of code here is how we navigate to the `Second` screen. We just `push` a new object which happens to contain an `id` that we agreed is our routing property that we base our routing on. Note that we don't specify the screen type, object, or tag here - this is the essence of separating routing from destination or implementation, which is a Good Thing.

On Android, we'll have to take care of navigation, and also the back button.

Android's Back Button



Tapping into the back button is unique to Android devices, in that it is on occasion a "hardware" button on a dedicated touch area of the glass as in Samsung phones or HTC phones, or a software button that renders at the lower part of the screen (common on the Nexus family of devices).

The following boilerplate is more or less needed verbatim in our app to support responding to the back button. Note that the `_navigator` variable is scope-global, and it gets filled on first navigation. First read this snippet of code to understand what's going on, and then I'd recommend tracking the `_navigator` variable throughout as well.

```
var {
  ...,
  ...,
  ...,
  BackAndroid
} = React;

var _navigator; // we fill this up upon on first navigation.

BackAndroid.addEventListener('hardwareBackPress', () => {
  if (_navigator.getCurrentRoutes().length === 1  ) {
     return false;
  }
  _navigator.pop();
```

```
15    return true;
16  });
```

`BackAndroid` is a simple library that binds to the native events of the host device.

Deep dive alert!: here's how it works (notice the special exitApp case):

```
1   RCTDeviceEventEmitter.addListener(DEVICE_BACK_EVENT, function() {
2     var invokeDefault = true;
3     _backPressSubscriptions.forEach((subscription) => {
4       if (subscription()) {
5         invokeDefault = false;
6       }
7     });
8     if (invokeDefault) {
9       BackAndroid.exitApp();
10    }
11  });
```

On iOS you need not worry if you're mistakenly (or if you lazily want to use the exact same code) using the `BackAndroid` module - all of the functions it carries are no-ops.

As with any event listener, when you add a listener, you must immediately ask yourself how you are going to remove it - does the `subscribe` method return a special handle you need to provide when canceling?, or do you have to do the bookkeeping yourself and come up with the same reference to the handler you provided?

In the `BackAndroid` case, when we subscribe we must keep a reference to the handler function we give it. However, do we really see ourselves disabling the back button in real life?

NavigatorIOS

If you're not making an Android version of your app, or you didn't choose to implement `Navigator` and `NavigationBar` then this NavigatorIOS is a lot simpler, and relies on the native iOS navigation stack. The iOS navigation stack is a powerful beast, it is only encouraging that it is hidden under such a simple React Native component, however be sure that if we wanted to do more involved things (custom Segues and such) it might have become trickier.

For now, let's implement the same `Navigator` example with `NavigatorIOS`.

```
class Navigation extends React.Component{
  render() {
    return (
      <NavigatorIOS
        style={styles.container}
        initialRoute={{
          title: 'first',
          component: First
        }} />
    );
  }
}
```

Our `Navigation` component is simple and explicit, and the initial route specifies the verbatim component (here `First`) that we want to run. As a side note, you might also like to call this component `Router` or `Handler` or anything that represents a concept of a routing shell component.

Next, we take a look at both our screens, `First` and `Second`.

```
class First extends React.Component{
  navSecond(){
    this.props.navigator.push({
        title: 'second',
        component: Second
    })
  }
  render() {
    return (
      <View style={styles.content}>
        <TouchableHighlight onPress={this.navSecond.bind(this)}>
          <Text>Navigate to second screen</Text>
        </TouchableHighlight>
      </View>
    );
  }
```

Somewhat similar in structure to `Navigator`, however again we see explicit mention of the `Second` screen.

Note that when we use a `NavigatorIOS` and a plain child `View` as in this case, we will need to handle the height of a typical iOS navigation bar. In other words, we need to add a `paddingTop` property to our `View` such that the content will be offset from under the navigation bar. On more advanced components, such as the `ScrollView`, we might want to look for the `automaticallyAdjustContentInsets` property that allows the component to handle this for us automatically.

The `Second` screen looks at least as simple:

```
class Second extends React.Component{
  render() {
    return (
      <View style={styles.container}>
        <Text>
          Second screen
        </Text>
      </View>
    );
  }
};
```

As mentioned during the `Navigator` overview, `NavigatorIOS` is simpler - we don't have any navigation bar to tweak here, and in this case the view is completely vanilla - reusable and clean as-is.

Passing Data

Often when doing navigation, in addition to the route, we want to pass an additional data, for the new screen-to-be to root on.

In the generic `Navigator` component, data is passed as part of the plain Javascript object (in this example, the `data` property), so we can do something like this:

```
    switch (route.id) {
      case 'first':
        return (<First navigator={navigator} data={route.data} title="first"/\
>);
```

With `NavigationBar` it will mostly be the same, if you take care to pass your data within your `routeMapper` route object, and picking it apart in your `routeMapper` callbacks.

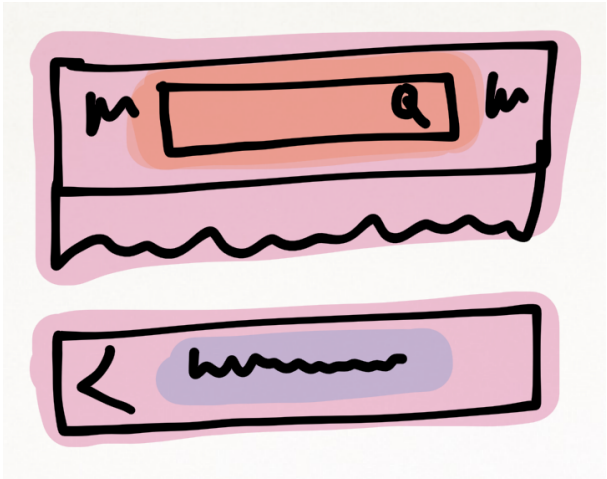
With `NavigatorIOS` we need to use the special `passProps` property:

```
  <NavigatorIOS
      initialRoute={{
        component: First,
        title: 'first',
        passProps: { data: 'some-data' },
      }}
    />
```

And the receiving component will get both the `data` property and a special `route` and `navigator` props it can use to make decisions and to navigate further.

There is a fascinating question about when a new screen is born due to a new navigation. Is the data that the screen just got from the route a pointer to data it needs to fetch?, or the actual data it needed to fetch verbatim?. No answer is wrong: it is a trade-off between being implicit and explicit, or defensive rather than naive, respectively.

Patterns



Search in Navbar

- `NavigatorIOS` - Can't do this.
- `Navigator`, with `Navigator.NavigationBar` - make sure that your Navigation-Bar mapper will render your variant of a title like this:

```
  Title: function(route, navigator, index, navState) {
    return (
      <AwesomeSearchbar .../>
    )
```

- `Navigator` and `ToolbarAndroid` - make sure that the screen you are routing to, which is supposed to contain the search bar inlined within the navbar, should now render a *single child* which is your searchbar, prefer not providing a title in this case (styling and others are cut for brevity):

```
<ToolbarAndroid>
  <AwesomeSearchbar .../>
</ToolbarAndroid>
```

A searchbar is interactive - it is completely OK to make the `AwesomeSearchbar` component interactive by supplying callbacks via `props`. You can pull these callbacks in two ways:

1. `NavigatorBar` renderer - from your route, or by making the renderer take parameters.
2. `ToolbarAndroid` - since it is rendered within a container view, simply the callbacks of the neareset smart component will do.

Custom Content in Title

As in the previous pattern, the idea is similar sans the callbacks.

- `NavigatorIOS` - content is string only
- `NavigatorBar` and `ToolbarAndroid` - as with the previous pattern, simply hand over the component you want to render, this time no callbacks or interactivity needed.

Routed Navbar Content

We saw this while handling `NavigatorBar` mapper - the content can change as a function of the route you are not traveling into.

- `NavigatorIOS` - each time you `push` a navigation, you can define how the coming-to-be navbar will look like, so this is a simple matter of using a new title:

```
this.props.navigator.push({
      title: "some new title",
      ...
      })
```

- `Navigator` and `NavigationBar` - use the route mapper, as we've seen before.
- `ToolbarAndroid` - this is trivial since you're rendering it as part of the view, so you can couple the rendering to the view itself. Meaning, don't fuss with the route, but with the actual view component content.

Reactive Navbar Content

There comes a time where your navbar changes, but not as a reaction to route changes, but to some kind of an external event, timed event, or anything reactive in nature.

The way to solve it right now, which is gaining consensus, is to inject an event emitter to your app flows, and make sure that components on that navbar know to use it.

- `NavigatorIOS` - again, not possible
- `Navigator.NavigationBar` and `ToolbarAndroid` - make sure that the content you give each, will be able to use your global event emitter:

```
<ToolbarAndroid>
  <AwesomeSearchbar emitter={this.emitter} .../>
</ToolbarAndroid>
```

You can also not use an explicit emitter but a Flux dispatcher, or do use an explicit emitter and inject it via something similar to React's context[66], an so on. Since this is an advanced/thought material topic (under the "lightbulb" category) I'll leave it to you for exploration.

Getting Input

If you want to just go to a screen to collect input (i.e. modals), you can either use the actual `Modal` component (see here[67]), or send off a view with a callback within the `renderScene` logic block. At extreme cases you can use event emitter, or if you're using Flux than a Flux action trivially solves it.

[66] https://facebook.github.io/react/docs/context.html
[67] https://facebook.github.io/react-native/docs/modal.html

Spreading Props

If we scratch our head for a moment, we remember that routing in `Navigator` is done with a plain Javascript object. We can actually be opinionated and define this object as such:

```
{
  id: 'route-id'
  props: {
    some: 'props'
  }
}
```

And this way we can then do something like this, within our `renderScene`:

```
  navigatorRenderScene(route, navigator) {
    _navigator = navigator;
    return (<First navigator={navigator}
                   {...route.props}/>);
  }
```

Using the new spread operator `...`, we easily inline the entire `props` bag of properties into our component. Remember React will go left-to-right on various props so you can enjoy a cascading effect (provide defaults and then override them with specific data)

Summary

This concludes our discussion about routing. In summary, we've learned the following:

- Routing is hard, however at least on mobile, we have less rope to hang ourselves with by adopting the best practices of each mobile platform.
- React Native offers the generic and flexible `Navigator` for a general case use, and the older `NavigatorIOS` for iOS specific work. Choose the latter if you don't need flexibility and want to run fast by getting the default iOS navigation stack behavior.
- Routing is either defined implicitly with `Navigator` and routes, or explicitly with `NavigatorIOS` and the specific components we route to.
- Remember to compensate for various UI glitches when using a navbar. Some content may vanish under the navbar because it is not padded.
- Passing data is easy through both `Navigator` and `NavigatorIOS`.
- `NavigationBar` is a common to both concept, it has its own tradeoffs.

Going Native: Native UI

Why go Native?

React Native allows you to build a custom, fully native UI, and mix it into your existing app. It does this by introducing a clever series of abstractions and building blocks we'll be using throughout this chapter.

But first, if you're reading this book from start to finish, there is a chance you'd like to explore the reasons for using pure native controls with React Native. After all, the primary reason for using React Native is to improve dealing with each and every mobile platform, its tooling, and developer experience.

Otherwise, if you're landing on this chapter with the intent of figuring out how to do this kind of voodoo, then you have probably bumped into one of the motivations I list out below.

Performance

There are two kinds of people trying out performance optimizations - those that actually have a performance problem and are spending each day trying to alleviate the pain by identifying performance hogs, memory leaks, and bad UI trees, and – those who heard about such problems.

First, don't be the latter; if you think you have a performance problem, try to create an isolated example that demonstrates it. Rip out that part from your app, isolate it, and try to reproduce. A lot of times during this process alone you find out the root causer of your problem; and even if you didn't, you now have an example that is shareable with others. It's been said that premature optimization is the root of all evil[68], and if you solve invisible problems thinking you are really fixing a performance problem, you probably are creating a bigger performance problem instead.

Lastly, yes. If you have a performance problem, using and mixing in a native UI component will be great for you. I'm willing to make a bet that you're having this problem on Android.

Hence, as a general rule of thumb - you'll be probably mixing in custom native controls on Android, involving lists and animations, which are the two most notorious issues on that platform.

[68] http://c2.com/cgi/wiki?PrematureOptimization

Making use of Existing Work

If you already built native iOS apps, and trying to make a new app that will be cross-platform with React Native, then you probably want to use any of that existing iOS infrastructure and UI components that you've built already. Of course, the same is true if you already have Android apps and want to expand to the iOS platform. This makes it a very trivial decision, use those existing components and ship things faster.

Better Tooling

It's not a secret that Xcode is pretty darn good. In some cases, you might find yourself building your UIs a lot faster in Interface Builder. For Android, it is less true since Android Studio is fairly young - but you may have your reasons to appreciate Android Studio better. Mixing in native controls is quite smart actually since React Native doesn't need a ViewController (iOS) or an Activity/Fragment (Android) – it just needs a simple View. So once you've designed it, you'll be implementing all logic in Javascript - this is how React Native takes away the bad choices and leaves you with one good way to do things.

Custom UI and Complex UI Work

This would probably be the best reason to use native controls. While React Native does aim to provide an all-encompassing solution for all of your mobile challenges, you should strive to be practical. Always think about that square peg into round hole[69] – if you are trying to do something no one had done before with React Native for too long, it might make sense to do away with that and build it in the relevant native platform as a custom view.

Wrapping Existing Components

You might want to pick out a control you've been using in your native work for a while, but that doesn't exist for React Native. If you'd like, you can put the time and effort to "wrap" it so that it will be available to you in Javascript land. If you do that, it would probably be very kind of you to share it with the community[70] :).

A General Escape Hatch

Every closed architectural solution should have escape hatches. Some may use the fancy "extension points" term, but the core reasoning is the same:

[69] http://history.nasa.gov/SP-350/ch-13-4.html
[70] https://github.com/jondot/awesome-react-native

you need to be sure you *always* have a solution should you need it. Having a native escape hatch is powerful because you obviously can do everything that is possible to be done there. This contributes to the feeling of safety and commitment that you carry when entering a new project, and it is definitely not always true for other cross-platform mobile SDKs. Also, this is also how Facebook did it.

When Not to Go Native?

Well, basically the answer to that lies with what you're making. If you're making a standard looking app, some variation of a master-detail or a document oriented app, with regular or standard (by means of the relevant mobile platform) UX, then by all means try to avoid making custom native controls. Take a look at the React Native Showcase[71] to find out if someone "had done it before".

Remember that when you look at the showcase, certain apps are edge cases - Chats, VR, Cameras and Effects, and so on are demanding apps that probably have performance or edge case concerns already, and there is a chance that they themselves use native controls. So when you try to compare the app you want to build with those and judge how "standard" it is, know which ones to avoid when comparing.

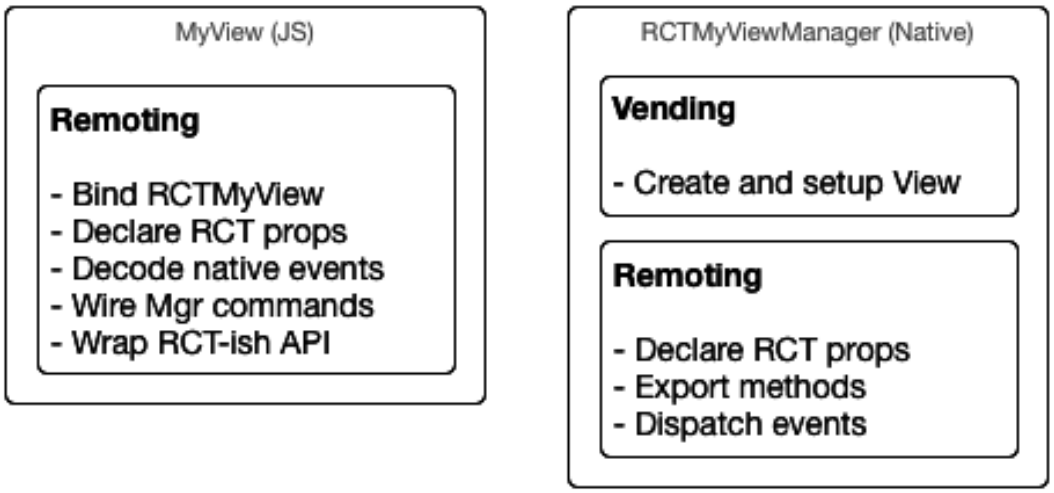
The Building Blocks of a Custom View

If we think about it, we can define what makes a custom view in a simple way:

- UI - The native UI hierarchy that is handed over to React Native. In the end, this can be anything that conforms to a `UIView` (iOS) or `View` (Android); be it a custom-drawing view of your own, or a composite view with many subviews.
- Props - The way React always strives to hand over state. Even with native UI, there is proper way to define React-ish props on your native view, and let React Native find it.
- Events - Your native view may emit events. `OnChange`, or any other event you can think of.
- Commands - Less React-ful, a command is an API that sits on top of your view, that orders it to do things. An example would be `ScrollView`'s `scrollTo`. It is less a "React way", because obviously the position of the scrollbar is also a sort of a state, and that specific detail isn't modeled with a prop. There will be situations, in reality, when you'll need these.

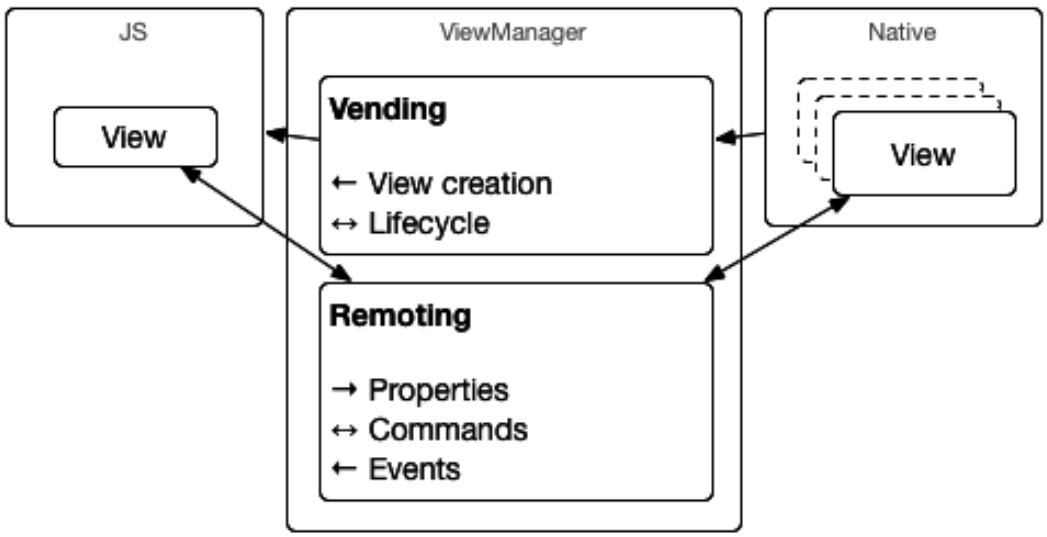
[71] https://facebook.github.io/react-native/showcase.html

Also, it may help to keep this mental image in mind (or at your desk), where it shows each “world” - native and Javascript, and its responsibilities.



The ViewManager

A custom native view will receive props, emit events, and accept commands. The way React Native glues these all up is by creating the concept of a `ViewManager`, which is responsible for creating our custom view, maintaining instances and the lifecycle of such views, and making sure messages are being passed back and forth from our React Native Javascript based view and our purely native iOS or Android view.

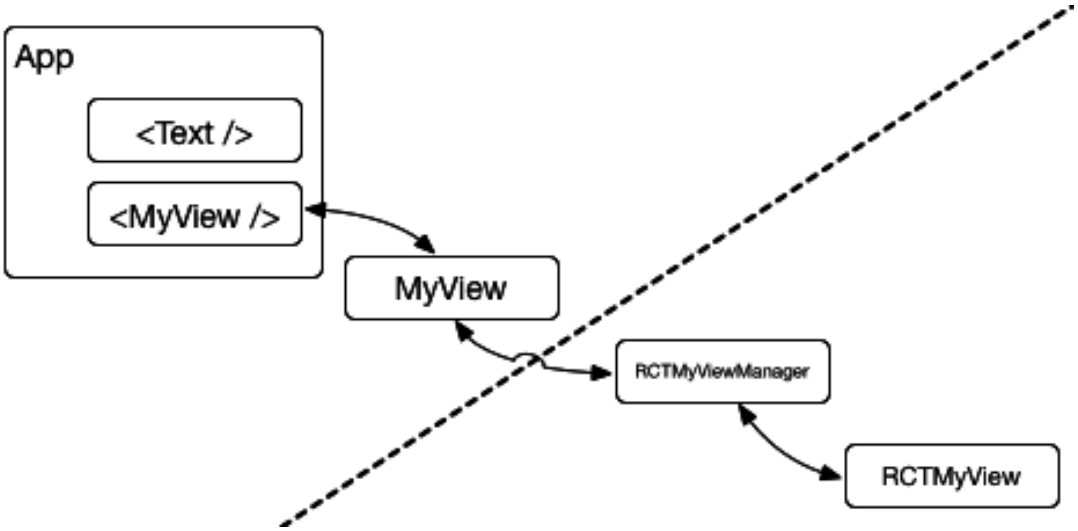


ViewManager’s Bridging Boundaries

With this diagram in mind, let’s go over what the ViewManager does:

- Create and set up our native view when it is requested by React Native, maintain an instance of it such that we don't pay the heavy price of recreating such a view on each render. This is referred to as "vending" the view.
- Implement the property setting bridge for the managed view. Often this will be just a simple delegation, but sometimes we'll want to create custom types from the simple types that are being passed across the remoting boundaries. Then reflect the React Native Javascript view prop changes through these back to the native view that's being managed.
- On iOS, implement an event bridge from managed view to React Native's dispatcher, and on Android this kind of logic can also belong to the managed view itself if you'd prefer. Then, receive events from the native view and move these along to the React Native Javascript view through the dispatcher.
- On iOS, declare commands as an exported method (more on this later), and on Android, define the available commands and implement a command dispatcher that will bridge a given command to the managed view. Then, accept command from React Native's Javascript view wrapper and move these along to the native view.

So basically, every time your native view needs to communicate, we're crossing the chasm with the help of the view manager:



Now that you know how this "voodoo" works, it will be fun, rather than strange, to see it working.

Our Example: MessagesView

The example native component we'll implement is a messages view. A composite view that has a `ListView`, and a `Label`. Specifically on iOS it is a

`UITableView` and a `UILabel` and on Android it is the performance optimized `RecyclerView` and a simple `TextView`.

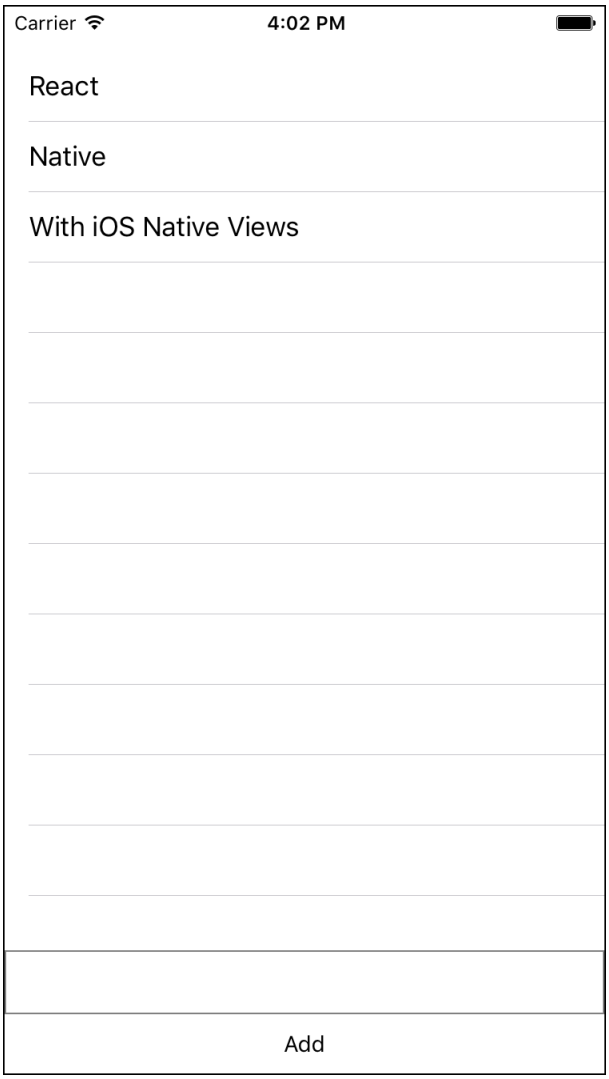
Our messages view will:

- List messages in a performant way
- Scroll to last message when a new one is added (Android)
- Bind the label to a last message prop
- Accept a command to add a message (we're modeling state out of React which is a Bad Thing, but this is only for demonstration)

This messages view can be a chat conversation view, a todo list item view, or what ever that fits the list-and-infinite-items pattern which often presents performance problems for rendering the list and a non-trivial manipulation of it that only works well when you break the abstraction and have a reach at the native implementation.

Next, let's take a look at the `nativecontrols` sample and break it down to its parts together.

Breakdown: iOS



Our hybrid iOS + React Native UI

Lacking a proper design, these examples are made to serve a single purpose: so that we figure out how to rinse-and-repeat wrapping or creating our own custom native components.

MessagesView

There are several ways to build composite or custom views on iOS. Programmatically, with Interface Builder, or a combination. We'll use a combination: use Interface Builder to sketch out your UI, and tweak it up through code. It is best to simply use the samples provided, rather build it from scratch if you didn't have that much experience with iOS so far, for more about this rather advanced topic, see this great guide[72].

[72] https://guides.codepath.com/ios/Custom-Views-Quickstart

Some times, you'll already have a view that you'd simply want to wrap up. In this case, the procedure is quite the same and we'll focus on that part as well.

First, let's take a look at such a custom view's inner workings. We'll try to point out the strange stuff that would be unfamiliar to anyone that has been working with iOS for a while.

As a side note - we'll be using Swift to build our view, and Objective-C for the ViewManager.

```
1  import Foundation
2  import UIKit
3
4  @objc protocol MessagesViewDelegate: class{
5    func messagesView(messagesView: MessagesView, didSelectIndex index: Int)
6  }
7
8  @objc class MessagesView : UIView, UITableViewDelegate, UITableViewDataSource{
9    @IBOutlet weak var tblView: UITableView!
10   @IBOutlet weak var lastMessageLbl: UILabel!
11   @IBOutlet var contentView: UIView!
12
13   var lastMessage:String {
14     get {
15       return lastMessageLbl.text!
16     }
17     set(msg){
18       lastMessageLbl.text=msg
19     }
20   }
21
22   var items: [String] = ["React", "Native", "With iOS Native Views"]
23
24   weak var delegate: MessagesViewDelegate?
25
26   func addItem(item: String){
27     self.items.append(item)
28     tblView.reloadData()
29   }
30
31   func initViews(){
32     // rig up our nib/xib with the current view
33   }
34
35   func tableView(tableView: UITableView, numberOfRowsInSection section: Int) \
36 -> Int {
```

```
37      return self.items.count;
38    }
39
40    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSI\
41  ndexPath) -> UITableViewCell {
42      let cell:UITableViewCell = self.tblView.dequeueReusableCellWithIdentifier\
43  ("cell")! as UITableViewCell
44      cell.textLabel?.text = self.items[indexPath.row]
45      return cell
46    }
47
48    func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: N\
49  SIndexPath) {
50      delegate?.messagesView(self as MessagesView, didSelectIndex: indexPath.ro\
51  w)
52    }
53  }
```

I've removed the view rigging code in `initViews` because it's not relevant (you can take a look at the sample code) and the standard constructors for brevity. So, take a good look at this view. See anything React Native specific? The answer is that there is none. To us, this is a plain iOS view, and this is good news since there are no leaky abstractions that litter our view, and if we had views from our existing codebase - this is a guarantee that we'd probably wouldn't need to modify them in order to repurpose them for React Native.

RCTMessagesViewManager

This is where most of our React Native specific code lives. And even then, we'll use React Native's infrastructure that it uses itself for everything that you've used so far (i.e. `Text`, `View` and so on), and we'll see that it feels quite baked and doesn't require that much coding.

Let's break down `RCTMessagesViewManager` into pieces and discuss each piece separately.

```
1  @interface RCTMessagesViewManager<MessagesViewDelegate> : RCTViewManager
2  @end
```

Here we obviously want to inherit from `RCTViewManager` which is the base view manager that React Native supplies, and conform to `MessagesViewDelegate` which is by convention our own way of saying that we want the view manager to take responsibility of for things happening inside MessagesView.

```
@implementation RCTMessagesViewManager

RCT_EXPORT_MODULE()
```

Using `RCT_EXPORT_MODULE` we're declaring our implementation of `RCTMessagesViewManager` as a React Native Native module. This is something you'll do for any component you want to expose to React Native, be it a UI component or a service module.

```
- (UIView *)view
{
  MessagesView *v =  [MessagesView new];
  v.delegate = self;
  return v;
}
```

This is the "vending" part. Here, in addition to creating a view, we're also setting it up with a delegate (which is the view manager itself).

Next up we go over the *Events*, *Props* and *Actions* building blocks of a native component.

```
- (void) messagesView:(MessagesView *)messagesView didSelectIndex:(int)indexP\
ath
{
  NSDictionary *event = @{
                          @"target": messagesView.reactTag,
                          @"index": @(indexPath)
                          };
  [self.bridge.eventDispatcher sendInputEventWithName:@"topChange" body: even\
t];
}
```

Starting out with `Events`. When we declared our `RCTMessagesView` as a delegate for `MessagesView`, we were conforming to `didSelectIndex` as one of the delegated methods. Here, we are implementing it, and in the end we want our Javascript wrapper view to be able to receive such an event from our native `MessagesView` component.

We use `topChange` as a name of an event which is internally mapped to `onChange` within the Javascript part of our view. Next up, we define the payload that would be passed to it. It is highly important to shape this payload properly, and not just throw a bunch of key/values onto it. It has a `target` key which you should leave as is, with the matching `messagesView.reactTag` value, or `anyOfYouViews.reactTag` as will happen with any view (remember to import the correct header files to get a hold of that - advise the samples).

In this example, for brevity and simplicity of the example our payload is an int (`indexPath`). For the general case I would recommend to shape this as a structured object, for example:

```
  "type": "messagesview#didselectindex/1"
  "payload" : {
    "key": "value"
      :
      :
  }
}
```

Where `type` is a directive that points to which method sent this event. The way I chose to depict it here is much like how Ruby notes a method (#), and Erlang notes methods with arity (number of parameters) in the name (/), to allow for overloads, and to help with tracing back to the right native method in a moment of need.

If you're using Flux, this event should be shaped just like your events generated from Flux actions[73], and the `type` field would be your `SPECIFIC_REACT_ACTION`; that would make a very elegant architecture. Incidentally, this is how the example data above is shaped, excluding the sneaky naming of the action as a way of making a better debugging experience.

```
RCT_CUSTOM_VIEW_PROPERTY(lastMessage, NSString, MessagesView )
{
  [view setLastMessage: json];
}
```

Moving on with `Props`. This is where we define our prop: it is called `lastMessage` and this is how it will appear to you in Javascript world, it is typed as a string (`NSString`) and the target view is a `MessagesView` which is exactly what we want.

This is also where we implement setting the prop from the outside world into our native component. Note that something magical is going on here as we have two magic variables: `view` and `json`, and obviously, we're tasked with merging the two together. So in our case, `json` would be an `NSString` but in other cases it can be a more complex object which you'll need to parse out and build a real structured object[74] from exactly here.

If you know the prop type is simple: String, Boolean, Integer, and so on, you are allowed to not provide an implementation part. Just use `RCT_EXPORT_VIEW_PROPERTY` with the same definition parts. Obviously, you can define all of your properties this way and get very far.

[73] https://github.com/acdlite/flux-standard-action

[74] https://github.com/facebook/react-native/blob/master/React/Views/RCTMapManager.m#L106

```
RCT_EXPORT_METHOD(addItem:(nonnull NSNumber *)reactTag
                  item: (NSString *)item)
{
  [self.bridge.uiManager addUIBlock:
   ^(__unused RCTUIManager *uiManager, NSDictionary<NSNumber *, MessagesView \
*> *viewRegistry) {

     MessagesView *view = viewRegistry[reactTag];
     if (!view || ![view isKindOfClass:[MessagesView class]]) {
       RCTLogError(@"Cannot find RCTMessagesView with tag #%@", reactTag);
       return;
     }

     [view addItem: item];
   }];
}
@end
```

Ending with `Commands`. This is where we define a command that we'd like to expose out to the Javascript world on an instance of the view it holds (use a `ref` to get a hold of it). While somewhat complex, if we squint, we'll be able to identify a very simple structure:

- `reactTag` is passed and being used later to fetch our instance of the managed view
- Running code is preceded by a little ceremony: The code should be supplied to the `UIManager` which will probably decided on which thread queue to run it.
- When we want to run code against our view (the command), we need to find it first, and we need to do that sort of "manually."

So if we walk through this piece of code, we see we are accessing the `bridge.uiManager` instance to `addUIBlock`:

```
  [self.bridge.uiManager addUIBlock:
   ^(__unused RCTUIManager *uiManager, NSDictionary<NSNumber *, MessagesView \
*> *viewRegistry) {
```

Keep a snippet of that block signature to yourself and just apply it each time when needed. Then within the block we are fetching our instance from the special `viewRegistry` variable, with our `reactTag` (which is just an interned number):

```
1       MessagesView *view = viewRegistry[reactTag];
```

Make the required hoola-hooping by validating the view not to be nil and of the correct class, and we're in.

```
1       if (!view || ![view isKindOfClass:[MessagesView class]]) {
2         RCTLogError(@"Cannot find RCTMessagesView with tag #%@", reactTag);
3         return;
4       }
```

Finally, call anything you want on the `view` and use the parameters passed through `RCT_EXPORT_METHOD`, in this case `item`.

```
1       [view addItem: item];
```

This completes our in-depth analysis of the `ViewManager`!. Next up, let's see what is required to glue a Javascript view, or view wrapper onto it.

messagesview.ios.js

What's left to do now, is make sure we create a Javascript view, that knows how to bind to our `MessagesViewManager`, and do our custom mapping between "raw" manager events and more beautiful Javascript events.

Note that I intentionally named this `messagesview.ios.js`, with the `ios` infix. This means that this file will only be visible on iOS builds, but that is not to say you must build a different Javascript view for iOS and Android - on the contrary - in the end we'll see that it is virtually the same, which completely drives home the fact that React Native is so great at promoting code sharing between platforms.

The reason I start out "pessimistic" is because intuition dictates that when you're building native components, there are going to be differences that are a bit immaterial - UX, custom props, events, or commands. You want to leave that hatch open, and not prematurely optimize the codebase by forcing it to be the same for both platforms. In the end, when I finish, I can easily gauge the similarity between the iOS and Android Javascript views, and I can then make a deliberate decision to merge the codebase, as well easily identify and separate the platform specific parts perhaps to a different Javascript module.

Now, let's go over the solution bit by bit.

```
import React,
      { requireNativeComponent, View }
      from 'react-native'
var MessagesViewManager = require('NativeModules').MessagesViewManager
```

Here we're pulling out the `MessagesViewManager`. This is our contact into the native `RCTMessagesViewManager` we've just implemented. React Native took care of the names. We'll use `requireNativeComponent` to do the final binding and `View` to drop a set of required props directly into our custom view.

```
class MessagesView extends React.Component {
  _onChange(event){
    this.props.onChange(event)
  }
```

Remeber the native `topChange` part? This is where we're reaching into the event that we've supplied. This so called *internal* `_onChange` is responsible for making the mapping into perhaps various many different kinds of change events. So, typically instead of directly delegating like in this example, we'll implement some switching logic over the `event.type` field.

```
  addItem(item){
    MessagesViewManager.addItem(
        React.findNodeHandle(this),
        item
      )
  }
```

Our command implementation. Remember that we are applying methods from Javascript land and hope they get beamed up to the native land by way of commands. This means we have everything we've exposed in our native `MessagesViewManager`, and you can simply access it here. Note that a special `findNodeHandle(this)` is needed in order find and beam up that `reactTag` we've been discussing before.

```
  render() {
    return <RCTMessagesView {...this.props} onChange={this._onChange.bind(thi\
s)} />
  }
}
```

This should be trivial, we're using an `RCTMessagesView` that we're going to create in a few more lines, and rigging up props and events. So far, so good.

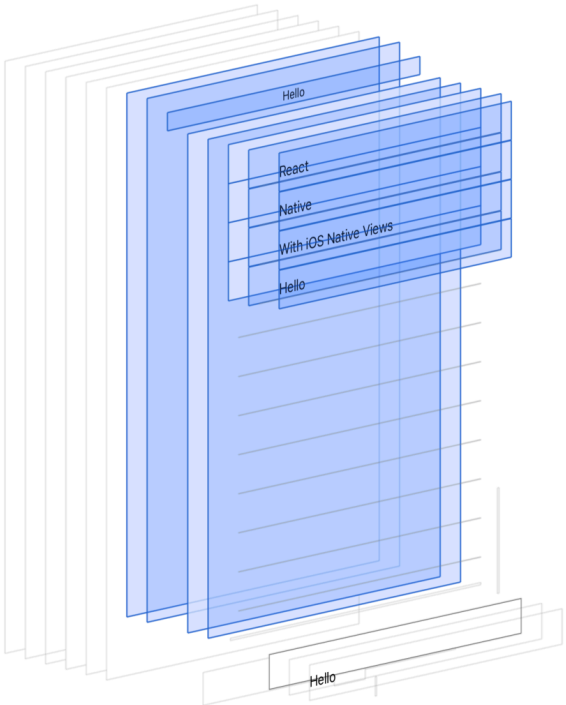
```
MessagesView.propTypes = {
  ...View.propTypes,
  lastMessage: React.PropTypes.string,
};
```

Like with normal React code, we are defining the required props and their types. However please note that his is *important* and not optional so that React Native can discover these and bind them.

```
var RCTMessagesView = requireNativeComponent('RCTMessagesView', MessagesView);
module.exports = MessagesView;
```

Finally, we're calling `requireNativeComponent` to make sure React Native binds `MessagesView` and `RCTMessagesView` together. The object that is created is the actual view we can use in our `render` method. Note that it feels a bit strange to define a class, which uses a dependency that is only defined in the future, with that class itself. This is a *circular dependency* which in software is always seen as bad, however the semantics here dictate that this is actually what we're doing - these two components are symbiotic.

To refresh your memory, below is the call chain sketch from before, and you can pretty much see how the native component relies on the Javascript component and the other way around. This image shows you how wonderful that synergy turns out to be:

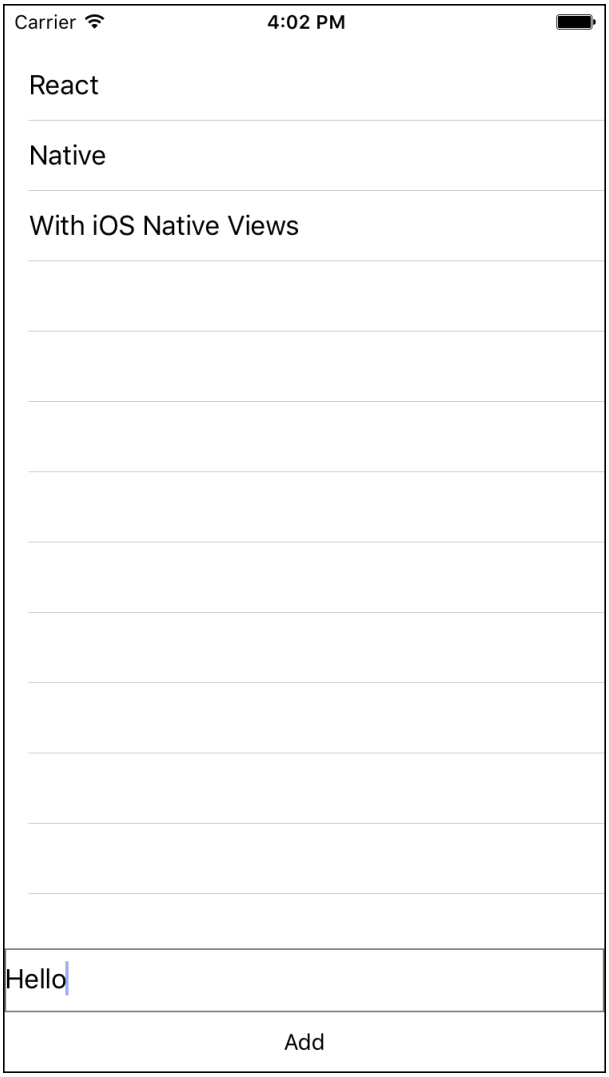


Breakdown of iOS UI: Native highlighted, React Native in white

This concludes our entire voyage into the iOS native UI component. It may have felt “heavy” to learn all of the low-level stuff, but remember that React Native is only at 0.19 (at the time of writing), and in my opinion, when you decide to adopt a technology at such an early (relatively) stage, you should be able to reason about how it works behind the scenes. And more importantly, you should have that native component escape hatch when you need it.

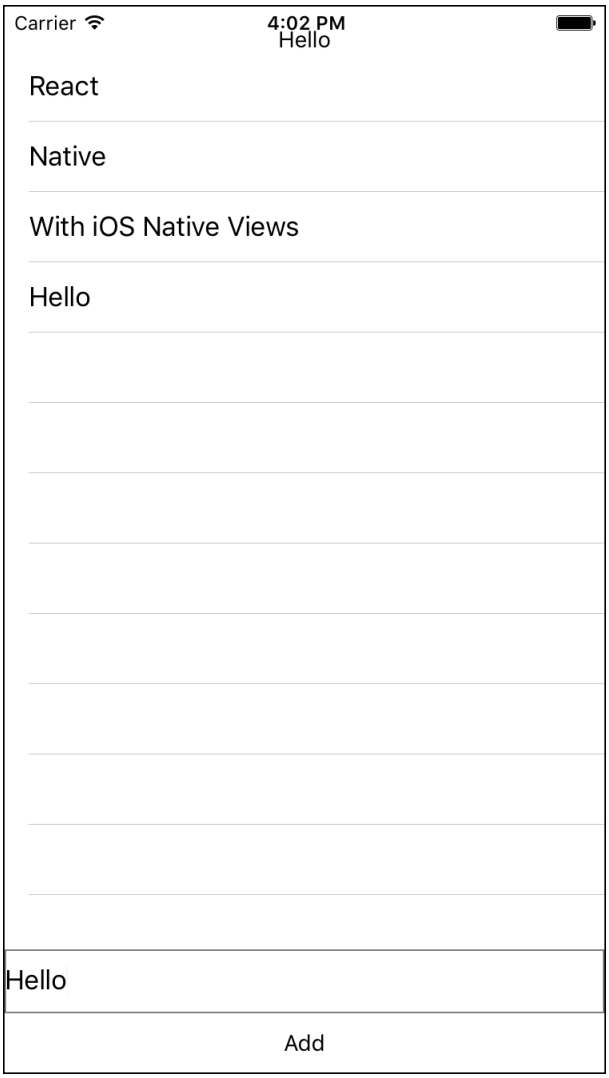
All in all, once you walk through this one time and work with the sample code, you’ll feel confident that wrapping native UI components which sounds and feels advanced, is actually easy and almost a mechanical thing to do.

Eventually, we get this:



Input is React Native, saying “Hello”

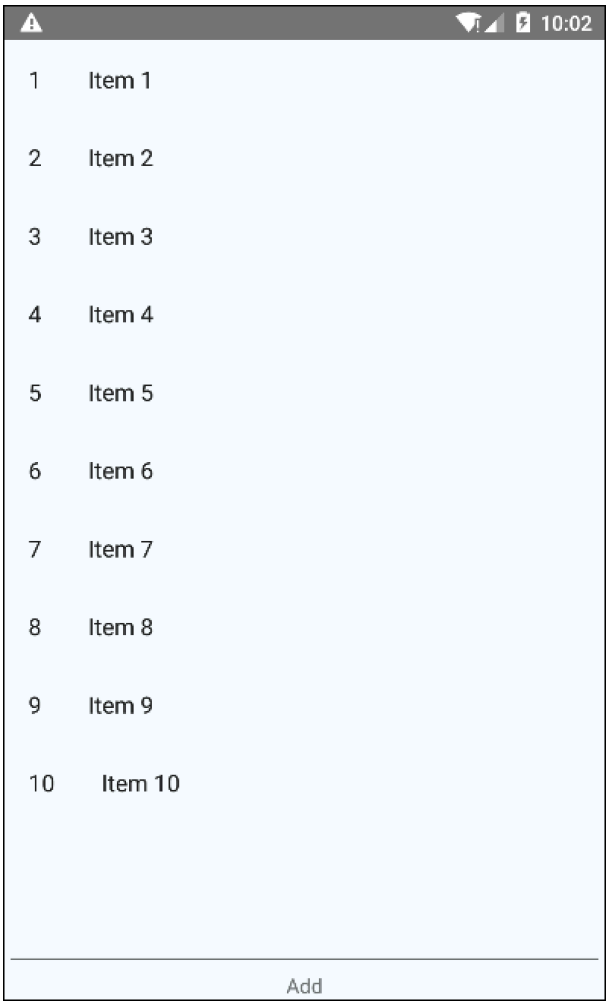
And adding an item adds the item to our item list, and label:



Clicking “Add” adds an item

Fun!

Breakdown: Android



Our hybrid Android + React Native UI

Again, note we're cheap on the UI style and design part. Remember: we only care about figuring out how to make our own custom native components here.

Let's take a look on our `MessagesView` fully blown, and as before, try to identify anything React Native specific that has leaked into our view.

MessagesView

```
public class MessagesView extends LinearLayout implements MessageViewDelegate\
 {
    private RecyclerView listView;
    private TextView lastMessage;

    public MessagesView(Context context) {
        super(context);
        init();
    }

    private void init() {
        inflate(getContext(), R.layout.messages_view,this);
        this.listView = (RecyclerView)findViewById(R.id.listView);
        this.lastMessage = (TextView)findViewById(R.id.lastMessage);

        LinearLayoutManager llm = new LinearLayoutManager(getContext());
        llm.setOrientation(LinearLayoutManager.VERTICAL);
        listView.setLayoutManager(llm);
        listView.setAdapter(new MessagesViewRecyclerViewAdapter(DummyContent.\
ITEMS, this));
    }

    @Override
    public void onItemTap(DummyContent.DummyItem item) {
        WritableMap event = Arguments.createMap();
        event.putString("item", item.toString());
        ReactContext reactContext = (ReactContext)getContext();
        reactContext.getJSModule(RCTEventEmitter.class).receiveEvent(
                getId(),
                "topChange",
                event);
    }

    public void addItem(final String content) {
        DummyContent.ITEMS.add(new DummyContent.DummyItem(content, content, c\
ontent));
        listView.getAdapter().notifyDataSetChanged();
        final int position = DummyContent.ITEMS.size() - 1;
        RecyclerView.LayoutManager layoutManager = listView.getLayoutManager(\
);
        //layoutManager.smoothScrollToPosition(listView, null, position);
        layoutManager.scrollToPosition(position);
    }

    public void setLastMessage(String msg){
        lastMessage.setText(msg);
```

```
47      }
48  }
```

See anything? Well, this time there is something: `onItemTap` reaches out and pulls the React Native event emitter in order to push out events. That detail was made intentionally, to show you that in Android you can do both. We can leave this code for convenience here, and we can also pull it out to our view manager (we'll take a look at that in a bit).

I've taken the liberty to use the same idioms as the iOS codebase. We're using a `delegate` and our `MessagesView` hands over that delegate to our adapter. Some Android developers would frown upon this selection of concepts, but it does not come from the fact that I'm originally an iOS programmer - on the contrary, I've started doing Android when it first went out and only a few years later I've done native iOS programming. So really the idea here is that React Native lets you adopt even the same ideas as means to an end on different platforms, as long as it works for you and your team, because the crux of your productivity will ideally happen within React Native itself.

Back to our code, what we're doing here is creating a compound view[75], which is Android's composite view - a view that holds one or more other subviews. For convenience (or laziness) we build a root view, that inflates a layout that has been designed visually in Android Studio.

[75] http://developer.android.com/guide/topics/ui/custom-components.html

Android Studio UI Editor showing MessagesView

And this is the view initialization code:

```
    private void init() {
        inflate(getContext(), R.layout.messages_view,this);
        this.listView = (RecyclerView)findViewById(R.id.listView);
        this.lastMessage = (TextView)findViewById(R.id.lastMessage);

        LinearLayoutManager llm = new LinearLayoutManager(getContext());
        llm.setOrientation(LinearLayoutManager.VERTICAL);
        listView.setLayoutManager(llm);
        listView.setAdapter(new MessagesViewRecyclerViewAdapter(DummyContent.\
ITEMS, this));
    }
```

Feel free to build this again as we walk through the code (if you're an

Android developer originally), or simply use the samples provided. Since we're using the fancy `RecyclerView` in our layout, we need to provide a layout manager, in this case a `LinearLayoutManager` - and in other words, we're making a list. The Android `RecyclerView` is able to optimize a list of *things* in various layouts (list, grid, etc.) in a way that it can move away items that are not displayed in order to improve performance.

What's left is as with the iOS part, walk through our *Events*, *Props* and *Commands*. In that order, let's continue with *Events*:

```
1     @Override
2     public void onItemTap(DummyContent.DummyItem item) {
3         WritableMap event = Arguments.createMap();
4         event.putString("item", item.toString());
5         ReactContext reactContext = (ReactContext)getContext();
6         reactContext.getJSModule(RCTEventEmitter.class).receiveEvent(
7                 getId(),
8                 "topChange",
9                 event);
10    }
```

Implementing the `MessagesViewDelegate` interface, we're bumping into `onItemTap` each time a user taps an item in our list. In it, we're building our payload that needs to be transmitted back to the Javascript view, and again here the same advice is correct - it is wise make sure that payload looks standard or effectively is a Flux action if you're into that. For the sake of the example, the payload here is nothing but a simple key and value for the item itself.

The context which you're given within the activity or fragment that eventually holds this native view, will be a special `ReactContext`, so any time you want to reach out and do something React Native specific, be sure to case to that context instead of the general Android `Context`. Here we're pulling the `RCTEventEmitter` so that we can instruct it to accept an event, and we're supplying a React tag ID with `getId`, and again using `topChange` with our events - this should be very similar but a bit different as with the iOS example.

Moving on to *Props*, let's observe the following:

```
1     public void setLastMessage(String msg){
2         lastMessage.setText(msg);
3     }
```

This is something we'll use later when we introduce the view manager, so for now keep in mind that it is just a simple setter.

And finally `Commands`, take a look at this:

```
    public void addItem(final String content) {
        DummyContent.ITEMS.add(new DummyContent.DummyItem(content, content, c\
ontent));
        listView.getAdapter().notifyDataSetChanged();
        final int position = DummyContent.ITEMS.size() - 1;
        RecyclerView.LayoutManager layoutManager = listView.getLayoutManager(\
);
        //layoutManager.smoothScrollToPosition(listView, null, position);
        layoutManager.scrollToPosition(position);
    }
```

This command is exposed out as `public` and - you guessed it - will later be used in our view manager. Note that we're reaching our to our own view, infrastructure (adapter) and our layout manager in order to perform scrolling. Commented out is a `smoothScroll` variant of scrolling - think how hard would it be to implement that with React Native in a performant way? If you want a hint, take a look at this discussion[76] in the excellent `gifted-messenger` project.

MessagesViewManager

Let's break down our MessagesViewManager, bit by bit (you can take a look at the complete class in the provided samples):

```
public class MessagesViewManager extends SimpleViewManager<MessagesView> {
    public static final String REACT_CLASS = "RCTMessagesView";
    private static final int CMD_ADDITEM = 1;

    @Override
    public String getName() {
        return REACT_CLASS;
    }
```

This is how we start. Inherit from `SimpleViewManager`, that takes our `MessagesView` as a generic parameter, and define various constants for later use. `REACT_CLASS` to supply through `getName`, and `CMD_ADDITEM` for our command map (more on this later). This is just a ceremony to comply with the bridging that React Native will want to do on our behalf.

[76] https://github.com/FaridSafi/react-native-gifted-messenger/issues/3

```
    @Override
    protected MessagesView createViewInstance(ThemedReactContext reactContext\
) {
        return new MessagesView(reactContext);
    }
```

Here we vend our view, just like with the iOS example. Nothing too exciting except that we're looking at a `ThemedReactContext` that is special (I encourage you to take a look and see how different it is from a regular `Context`).

```
    @ReactProp(name="lastMessage")
    public void setLastMessage(MessagesView view, @Nullable String msg){
        view.setLastMessage(msg);
    }
```

Moving on to `Props`, this is how we expose a prop in Android's version of React Native. While a personal opinion - it does look cleaner. Obviously, `lastMessage` is the name of the prop we're exposing here.

```
    @Override
    public void receiveCommand(MessagesView root, int commandId, @Nullable Re\
adableArray args) {
        switch(commandId){
            case CMD_ADDITEM:
                root.addItem(args.getString(0));
        }
        super.receiveCommand(root, commandId, args);
    }

    @Nullable
    @Override
    public Map<String, Integer> getCommandsMap() {
        return MapBuilder.of(
                "addItem", CMD_ADDITEM
        );
    }
}
```

Lastly, *Commands*. We're saying two things here. First, we're implementing a generic looking `receiveCommand` which will accept a view, a command ID, and zero or more arguments. Our job is to make sure we map each `commandId` to its actual command (did someone say flux actions? :-). Second, we need to tell React Native what commands are we allowing and what are their names and values, and this is done via `getCommandsMap`. This is how you can make sure that no one gives you an unfamiliar `commandId` in `receiveCommand`.

NativeControlsPackage

In the Android world of React Native, we need to explicitly tell it how to find our `MessagesViewManager`. This is best done by supplying a ReactPackage, which you can look at as a bundle of custom modules you want React Native to know about. It is also wise to invest and make your own `NativeControlsPackage` which will specify this `MessagesViewManager` as well as any future managers you'd want to build.

```
1  public class NativeControlsPackage implements ReactPackage{
2      @Override
3      public List<NativeModule> createNativeModules(ReactApplicationContext rea\
4  ctContext) {
5          return Collections.emptyList();
6      }
7
8      @Override
9      public List<Class<? extends JavaScriptModule>> createJSModules() {
10         return Collections.emptyList();
11     }
12
13     @Override
14     public List<ViewManager> createViewManagers(ReactApplicationContext react\
15 Context) {
16         return Arrays.<ViewManager>asList(new MessagesViewManager());
17     }
18 }
```

The only place we care about so far, is `createViewManagers` where we return a list with a single item - our `MessagesViewManager` instantiated. If you want to see a full blown `ReactPackage`, take a look at the `MainReactPackage` class within React Native itself:

```
1  public class MainReactPackage implements ReactPackage {
2
3    @Override
4    public List<NativeModule> createNativeModules(ReactApplicationContext react\
5  Context) {
6      return Arrays.<NativeModule>asList(
7        new AsyncStorageModule(reactContext),
8        new ClipboardModule(reactContext),
9        new DialogModule(reactContext),
10       new FrescoModule(reactContext),
11       new IntentModule(reactContext),
12       new LocationModule(reactContext),
```

```
13        new NetworkingModule(reactContext),
14        new NetInfoModule(reactContext),
15        new WebSocketModule(reactContext),
16        new ToastModule(reactContext));
17    }
18
19    @Override
20    public List<Class<? extends JavaScriptModule>> createJSModules() {
21      return Collections.emptyList();
22    }
23
24    @Override
25    public List<ViewManager> createViewManagers(ReactApplicationContext reactCo\
26  ntext) {
27      return Arrays.<ViewManager>asList(
28        ARTRenderableViewManager.createARTGroupViewManager(),
29        ARTRenderableViewManager.createARTShapeViewManager(),
30        ARTRenderableViewManager.createARTTextViewManager(),
31        new ARTSurfaceViewManager(),
32        new ReactDrawerLayoutManager(),
33        new ReactHorizontalScrollViewManager(),
34        new ReactImageManager(),
35        new ReactProgressBarViewManager(),
36        new ReactRawTextManager(),
37        new RecyclerViewBackedScrollViewManager(),
38        new ReactScrollViewManager(),
39        new ReactSwitchManager(),
40        new ReactTextInputManager(),
41        new ReactTextViewManager(),
42        new ReactToolbarManager(),
43        new ReactViewManager(),
44        new ReactViewPagerManager(),
45        new ReactTextInlineImageViewManager(),
46        new ReactVirtualTextViewManager(),
47        new SwipeRefreshLayoutManager(),
48        new ReactWebViewManager());
49    }
50  }
```

Obviously there's a lot more going on there, but you can also verify to yourself that you are using the very same technology React Native is using behind the scenes in order to extend it. The React Native team is showing us dogfooding at its best!

With this, we conclude our discussion of the Android view manager.

messagesview.android.js

Let's move on to the Javascript world, and see how to bind that view to our native components view.

I'm boldly stating this: both the `.ios` and `.android` view versions of messagesview can be (quite amazingly) converged to the same class! (or file, or codebase). I'm keeping an `.android` version here so that we can prove this to ourselves, now, let's see how this unfolds by breaking down the `MessagesView` class (again, you can get the full listing from the samples provided):

```
import React,
  { requireNativeComponent,
    PropTypes }
from 'react-native'
import UIManager from 'UIManager'
import View from 'View'
```

Nothing quite new here, so moving along.

```
class MessagesView extends React.Component {
  constructor(props){
    super(props)
    this._onChange = this._onChange.bind(this)
  }
```

Still, nothing new. Let's take a look at `_onChange`:

```
  _onChange(event){
    if(!this.props.onItemTapped){
      return
    }
    this.props.onItemTapped(event.item)
  }
```

Still no change, we're only showing a bit of a different flavor. We're naming the target prop differently and making sure the prop exists.

```
  addItem(item){
    UIManager.dispatchViewManagerCommand(
       React.findNodeHandle(this),
       UIManager.RCTMessagesView.Commands.addItem,
       [ item ])
  }
```

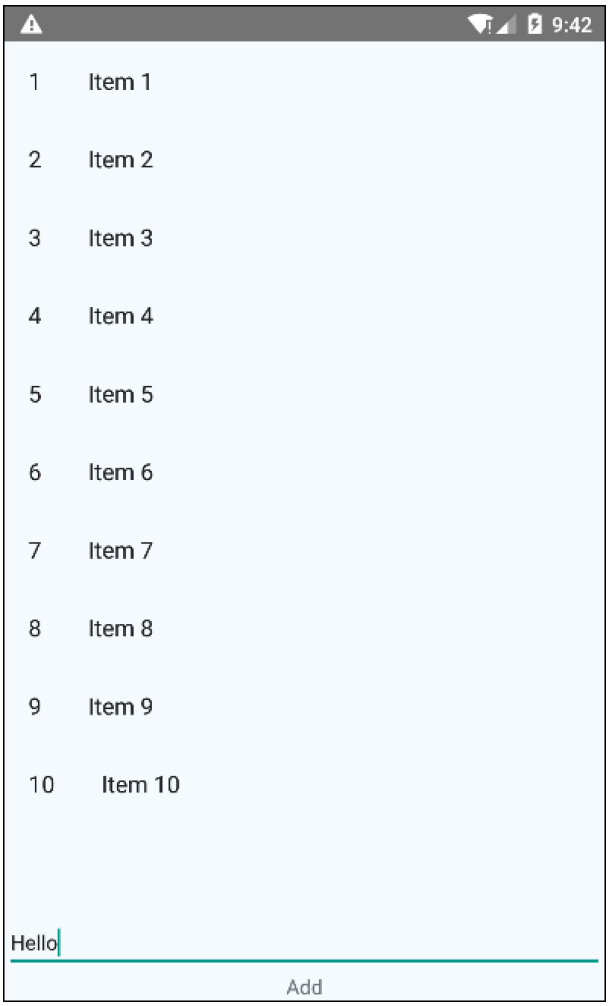
Here is our first practical difference between the `.android` and `.ios` versions. We're using a `UIManager.dispatchViewManagerCommand` in order to relay our command (in iOS, we used the `MessagesViewManager` native module that we required from the `native-modules` package).

If we wanted to converge the codebase, both versions could be easily abstracted away with a service class that might have been called `Commands`.

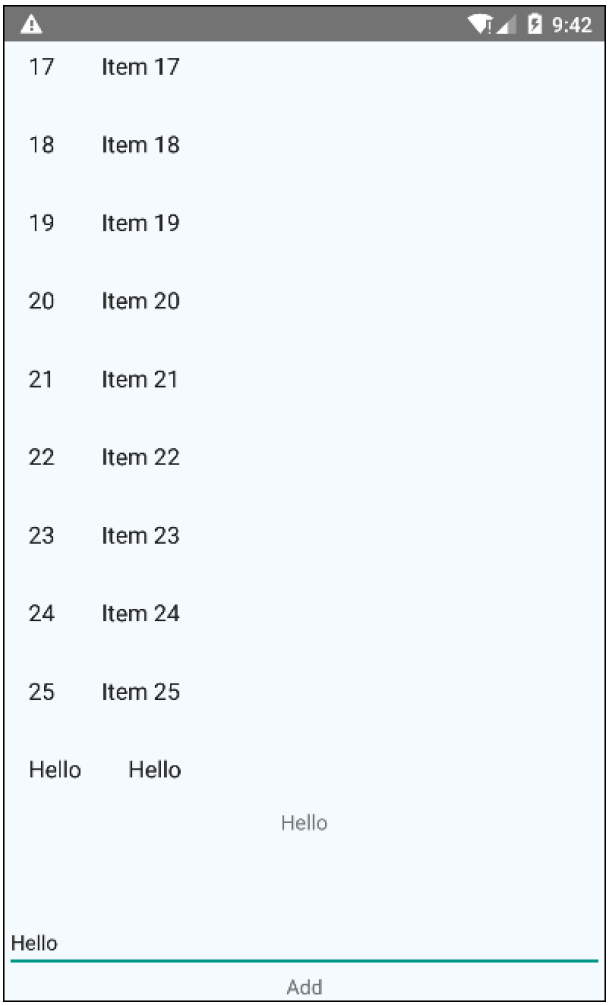
```
  render(){
    return(
      <RCTMessagesView {...this.props} onChange={this._onChange} />
    )
  }

}
MessagesView.propTypes = {
  ...View.propTypes,
  onItemAdded: PropTypes.func,
  lastMessage: PropTypes.string,
}

let RCTMessagesView = requireNativeComponent('RCTMessagesView',
                                             MessagesView,
                                             {nativeOnly:{onChange:true}});
module.exports = MessagesView
```

The last part shows that again, a minimal-to-no difference between both the `.ios` and `.android` versions exists. Just for the sake of the example, here `lastMessage` is bound through props. Also, `propTypes` is set differently. None of these are a deal breaker for converging the code, and some of them are here just to show that you have more than one way to do things.

And here is how it all behaves, on Android:



Input is React Native, saying “Hello”



Clicking “Add” adds an item

Using MessagesView

And now, to enjoy the cherry on the top. To use `MessagesView` in our container views, we can treat it like any other view, and even better, we get to keep a single codebase for both Android and iOS, using a different native implementation at once!

Here’s how it looks:

```
<MessagesView
  ref="messages"
  lastMessage={this.state.lastMessage}
  style={{flex:1}}
  onChange={(ev)=>console.log(ev)} />
```

And, anywhere you’d like, you can send commands to this view like so:

```
addMessage(){
  this.refs.messages.addItem(this.state.text)
  this.setState({lastMessage:this.state.text})
}
```

Summary

In this chapter, we covered one of the most hard-core topics of React Native, this is a topic that touches on how React Native itself is built and allows you to make a good use of the same building blocks in order to make your own escape hatches, and make sure your app carries on by implementing UI natively per platform, if you suddenly find that React Native doesn't have all of the answers.

We've learnt that to do this kind of bridging, React Native expects:

- A custom view of your own, per platform
- A view manager that takes that view, and takes care of vending it
- Remoting and the concepts of: *Props*, *Events*, and *Commands* and making sure the view manager knows how to glue these up.

And that's basically it. If you like to dial things up to eleven, make sure to read the code for ScrollView and WebView[77] for iOS, and the same on Android[78].

[77] https://github.com/facebook/react-native/tree/master/React/Views

[78] https://github.com/facebook/react-native/tree/master/ReactAndroid/src/main/java/com/facebook/react/views

Going Native: Native Modules

In this chapter we'll see how to expose native platform capabilities on iOS and Android, and that you can make available to your Javascript by building and exposing a React Native module.

If you didn't read the previous *Going Native: Native UI* chapter, please take the time to do it now. You don't *have* to, but even if you're not building a native UI component, much of the material in that chapter is relevant since by way of building the native UI component we had to also build a native module.

If you really only want to make a module I'll explain everything as well - so no worries; when you feel that you want to know *how* everything works behind the scenes, check out the previous chapter.

Our Example: Cryptboard

Since we've already gone through quite a lengthy road in the previous sample, every thing you want to do should already be clear to you in terms of React Native plumbing and remoting. All you should be missing is to validate that building a module is quite the same, and to see how to rig up such a module into the application infrastructure.

Since modules are simple to make, a lot, if not all of the common native facilities such as contacts, geolocation, notifications, storage, and so on are already covered either by React Native or by the community and you probably wouldn't find yourself making one from scratch. You might, however, find yourself improving or tweaking one and for that, the knowledge you've gained in the previous chapter should be enough.

So given that, let's expose the native clipboard as a module with a twist - it will store the contents encrypted using AES. We want to reach out and grab the clipboard's content from iOS or Android, and then to be able to encrypt and set a content of our own. As with any module, we'll be accessing this as a pure Javascript API with no UI.

The code will be very identical to the existing `RCTClipboard` module already provided with React; should you ever want to compare and see how a clear-text clipboard module would behave :-).

In this example, we'll insist on having a *single javascript codebase*, which means both iOS and Android will load the very same files, with no `.ios` or `.android` qualifiers!

iOS Breakdown

We'll show a few things on iOS:

- Building a module
- Configuring the project for external libraries with Cocoapods
- Pointing the project at a single `index` bundle, no `.ios` qualifier

Let's start with adding an `AES` capable library. I've chosen AESCrypt[79] to start with for its simplicity. Using Cocoapods let's do this:

```
[nativemodules] $ cd ios
[nativemodules/ios] $ pod init
```

Then edit your `Podfile` to be this one:

```
target 'nativemodules' do
  pod 'AESCrypt'
end

target 'nativemodulesTests' do

end
```

Finally run `pod install` and open the newly generated `nativemodules.xcworkspace` rather than the existing `nativemodules.xcproject`. You should now have access to the `AESCrypt` library from your Objective-C code.

Let's proceed with our native `Cryptboard` module:

```
// RCTCryptboard.m

#import "RCTBridgeModule.h"
#import "RCTUtils.h"
#import <UIKit/UIKit.h>
#import "AESCrypt.h"


#define CRYPT_PASS @"jondot/awesome-react-native"

@interface RCTCryptboard : NSObject <RCTBridgeModule>
@end
```

So far, if you've read the previous chapter this should be familiar. If not, then this is just a ceremony we have to perform: inherit from `NSObject<RCTBridgeModule>`. We also import the bridge, utils, and our `AESCrypt` headers

[79] https://github.com/Gurpartap/AESCrypt-ObjC

```
@implementation RCTCryptboard

RCT_EXPORT_MODULE()

- (dispatch_queue_t)methodQueue
{
  return dispatch_get_main_queue();
}
```

With this, we're registering our module and telling which thread queue React Native should use. Not all tasks are made equal and we have the liberty of telling React Native what priority to run our tasks at by supplying the matching queue. This should make sense to you if you're originally an iOS developer, and if not, I recommend reading about the various dispatch queues iOS offers and the threading model and reasoning behind these here[80].

```
RCT_EXPORT_METHOD(set:(NSString *)content)
{
  UIPasteboard *clipboard = [UIPasteboard generalPasteboard];
  clipboard.string = [AESCrypt encrypt:content password:CRYPT_PASS];
}

RCT_EXPORT_METHOD(get:(RCTPromiseResolveBlock)resolve
                  rejecter:(__unused RCTPromiseRejectBlock)reject)
{
  UIPasteboard *clipboard = [UIPasteboard generalPasteboard];
  resolve(@[RCTNullIfNil([AESCrypt decrypt:clipboard.string password:CRYPT_PA\
SS])]);
}
@end
```

This should also be very familiar, we're using the same `EXPORT_METHOD` macros that we've used before, and these tell React Native to expose our methods on the bridge. Note that the `get` method makes use of `RCTPromiseResolveBlock` which will appear as a regular Javascript promise on the other end.

We're also making use of the `AESCrypt` library to encrypt the clipboard data but that's not especially related to React Native itself.

That's it. From my experience making a module is really easy, and feels more enjoyable than wrapping a full-blown native component with UI *and* logic.

Since we're using a single Javascript codebase for both platforms this time, we'll leave it for the end as a dessert. Let's move on to the Android part.

[80] https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html

Android Breakdown

Let's go over the Java part. In my opinion it looks cleaner and more structured than our iOS counterpart.

```
// CryptboardModule.java
public class CryptboardModule extends ReactContextBaseJavaModule {
    final private String CRYPT_PASS = "awesome-react-native/jondot";

    public CryptboardModule(ReactApplicationContext reactContext) {
        super(reactContext);
    }

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

    private ClipboardManager getClipboardService() {
        return (ClipboardManager) getReactApplicationContext().getSystemServi\
ce(getReactApplicationContext().CLIPBOARD_SERVICE);
    }
```

For the initialization, the most important part is to extend `ReactContextBaseJavaModule` and to provide a module name, here we choose `Cryptboard` so that it will deliberately match the iOS part. The rest is the typical Android pattern of grabbing a system service such as the clipboard.

```
    @ReactMethod
    public void get(Promise promise){
        try {
            ClipboardManager clipboard = getClipboardService();
            ClipData clipData = clipboard.getPrimaryClip();
            if (clipData == null) {
                promise.resolve("");
            }
            if (clipData.getItemCount() >= 1) {
                ClipData.Item firstItem = clipboard.getPrimaryClip().getItemA\
t(0);
                promise.resolve("" + firstItem.getText());
            } else {
                promise.resolve("");
            }
        } catch(Exception e) {
            promise.reject(e);
```

```
18          }
19      }
20
21      @SuppressLint("DeprecatedMethod")
22      @ReactMethod
23      public void set(String text) {
24          ClipData clipdata = ClipData.newPlainText(null, text);
25          ClipboardManager clipboard = getClipboardService();
26          clipboard.setPrimaryClip(clipdata);
27      }
28  }
```

Still everything here looks not surprising. We're marking methods that will be exposed to the Javascript world with the `@ReactMethod` annotation. Other than that, the only thing I can think of that is worth mentioning is the use of `Promise` in the `get` method; keep that in mind as we will have to discuss it separately further down below.

Next up let's make React Native for Android know about this module. This done by creating and registering a `ReactPackage`.

```
1   public class NativeModulesPackage implements ReactPackage{
2       @Override
3       public List<NativeModule> createNativeModules(ReactApplicationContext rea\
4   ctContext) {
5           return Arrays.<NativeModule>asList(
6               new CryptboardModule(reactContext)
7           );
8       }
9
10      @Override
11      public List<Class<? extends JavaScriptModule>> createJSModules() {
12          return Collections.emptyList();
13      }
14
15      @Override
16      public List<ViewManager> createViewManagers(ReactApplicationContext react\
17  Context) {
18          return Collections.emptyList();
19      }
20  }
```

Here, we're plainly giving out a list of one item, our `CryptboardModule` in the `createNativeModules` override, while the rest of the overrides hand out empty lists - we don't have anything that React Native needs to know about there.

Let's rig that package up with our main activity (take a look at `getPackages` in your activity):

```
    protected List<ReactPackage> getPackages() {
      return Arrays.<ReactPackage>asList(
              new MainReactPackage(),
              new NativeModulesPackage()
      );
    }
```

That completes the cycle! Now, we'll be able to use our native module from both iOS and Android. The great thing is that we're going to use the very same Javascript codebase with no change for both.

Cryptboard.js

This part is going to be a surprise for you. Ready? Here we go:

```
import { Cryptboard } from 'NativeModules'

class CryptboardModule{
  get(){
    return Cryptboard.get()
  }

  set(str){
    // optional: do fancy stuff on str?
    Cryptboard.set(str)
  }
}

module.exports = new CryptboardModule
```

That's it. And this works both for Android and iOS. If you ever wondered about what is the X-Factor with React Native as opposed to other cross-platform SDKs; well, this is it.

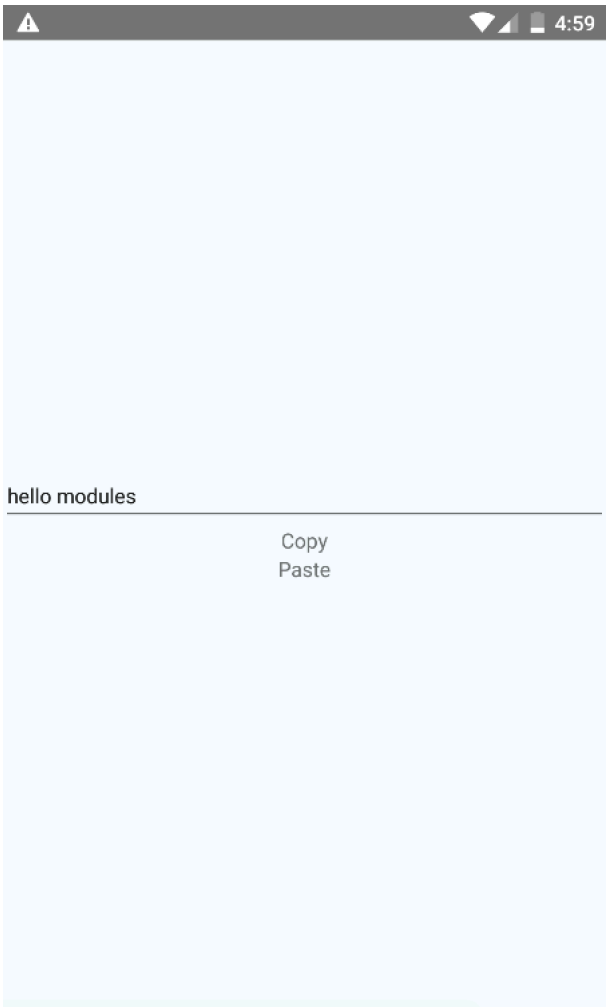
Now, let's see how we use this module from actual application code:

```
import Cryptboard from './cryptboard'

class nativemodules extends Component {
  constructor(props){
    super(props)
    this.state = { text: "hello modules" }
    this.copy = this.copy.bind(this)
    this.paste = this.paste.bind(this)
  }
```

```
10    copy(){
11      Cryptboard.set(this.state.text)
12    }
13    paste(){
14      Cryptboard.get().then((args)=>{
15        // unfortunately, an iOS promise is different from an Android promise.
16        // The iOS one returns a single variable which is an array of objects
17        // while the Android one returns a single object.
18        var text = args
19        if(args instanceof Array){
20          text = args[0]
21        }
22        this.setState({text})
23      })
24    }
25    render() {
26      return (
27        <View style={styles.container}>
28          <TextInput
29            style={{height: 40, borderWidth: 1}}
30            value={this.state.text}
31            onChangeText={(text)=>this.setState({ text })} />
32          <TouchableOpacity onPress={this.copy}>
33            <Text>Copy</Text>
34          </TouchableOpacity>
35          <TouchableOpacity onPress={this.paste}>
36            <Text>Paste</Text>
37          </TouchableOpacity>
38        </View>
39      );
40    }
41  }
```

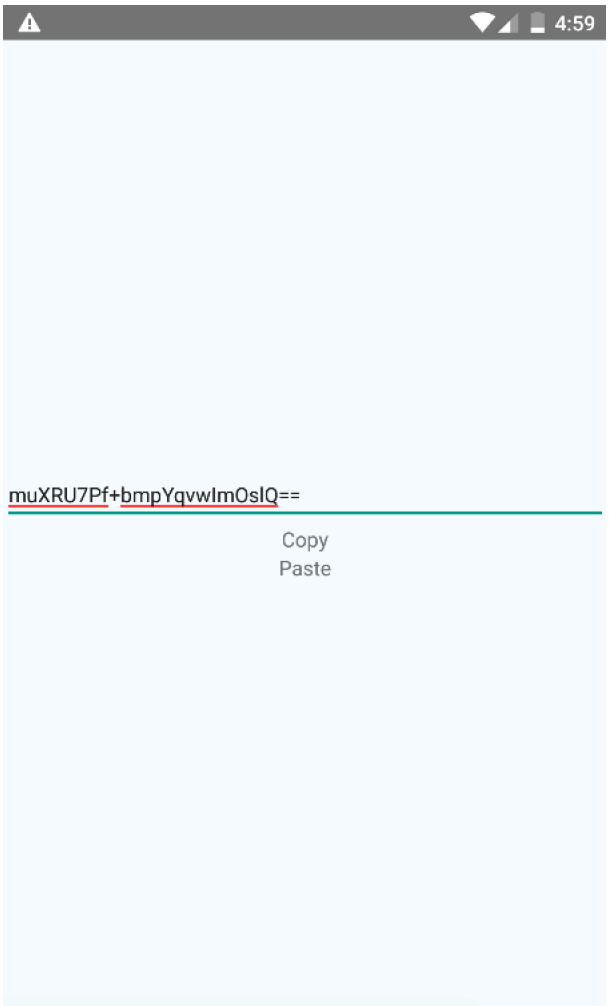
I've omitted redundant imports and style definitions, but this should be it. Again this is the same codebase for iOS and Android. This is where React Native shines as it makes a hard thing - remember we're getting our hands dirty with native code here, not necessarily React Native code - trivially easy.

And finally, this is how it should behave for both iOS and Android:



Our Simplistic Clipboard UI

Copy and Paste works, they copy and paste out the very same text, on both platforms. Let’s paste directly from the clipboard *without* hitting `Paste`:



Our Encrypted Text (Base64)

So encryption works. This means no one besides us can understand anything that is being copied or pasted from this app. Now all we need is a tinfoil hat!

Bridging Promises

But hold on a second. There's one thing left to talk about, which is probably just a sign that React Native is at 0.19, and is still very young. The way it bridges the concept of a promise is different between iOS and Android, unfortunately. On iOS React Native will pass an array with objects on a promise callback, while on Android it will pass a single object.

For now, such friction is dealt with explicitly as you can see in our code. Hopefully these differences will be ironed out - the encouraging thing is that it should be quite easy to iron out as well (as you can see from our small fix there).

Using a Single Codebase

In this example, we're using a single React Native codebase, with just two files:

- `index.js`
- `cryptboard.js`

Normally, your React Native projects will load an `index.[platform].js` and expect a Javascript bundle that is also infixed with the `platform` qualifier. Let's see how to disable that and make it load the very same codebase both on iOS and Android.

On iOS the solution is quite simple. Within `AppDelegate.m` locate and change `jsCodeLocation` to this (we've removed the `.ios` infix):

```
  jsCodeLocation = [NSURL URLWithString:@"http://localhost:8081/index.bundle?\
platform=ios&dev=true"];
```

On Android, we have to change few places. First, in your `MainActivity`, override `getBundleAssetName` and `getJSMainModuleName`:

```
    @Override
    protected String getBundleAssetName() {
        return "index.bundle";
    }

    @Override
    protected String getJSMainModuleName() {
        return "index";
    }
```

This allow you to conveniently instruct React Native which file and bundle to fetch. Next, since the Android part includes Gradle, you can also tweak the build tasks for your convenience. Locate your `app/build.gradle` file and within it uncomment and set this part as the following:

```
// app/build.gradle
project.ext.react = [
    bundleAssetName: "index.bundle",
    entryFile: "index.js"
]
```

Now, both your iOS and Android builds will use the same codebase configuration.

Summary

In this chapter, we've breezed through implementing native modules. These are the kinds of modules that you need when you want to expose part of the native specific capabilities to your React Native Javascript code.

This is a recap of what we did:

- We've implemented a fun example - a Cryptboard. A clipboard that saves its data encrypted with the symmetric AES encryption algorithm
- By that way, we've seen how to take in dependencies for iOS with Cocoapods, and for Android with Gradle
- We've learnt that it is much simpler than building a full-blown native UI component
- It boils down to registering the module, exposing methods and optionally using promises
- When using native modules, there's a greater chance the way you use these from both iOS and Android Javascript modules will be exactly the same. We've seen how to force React Native to use the same codebase on each platform from the get-go.
- implement the clipboard service in both plats.
- go over them.
- the difference between promise blocks - iOS and Android
- how to make iOS and Android use a single index.js (ios - appmodule, android - overriding activity and fixing the gradle build tasks)
- screenshots - copy, paste, paste encrypted