OCTOBER 2, 2018 / #REACT NATIVE

How to structure your project and manage static resources in React Native

by Khoa Pham

React and React Native are just frameworks, and they do not dictate how we should structure our projects. It all depends on your personal taste and the project you're working on.

you are free to apply only the pieces that suit you. Hope you learn something.

For a project bootstrapped with `react-native init`, we get only the basic structure.

There is the `ios` folder for Xcode projects, the `android` folder for Android projects, and an `index.js` and an `App.js` file for the React Native starting point.

```
ios/
android/
index.js
App.js
```

As someone who has worked with native on both Windows Phone, iOS and Android, I find that structuring a project all comes down to separating files by **type** or **feature**

type vs feature

Separating by type means that we organise files by their type. If it is a component, there are container and presentational files. If it is Redux, there are action, reducer, and store files. If it is view, there are JavaScript, HTML, and CSS files.

Group by type

```
    reducers
  components
    container
    presentational
  view
    javascript
    html
    css
```

This way, we can see the type of each file, and easily run a script toward a certain file type. This is general for all projects, but it does not answer the question “what is this project about?” Is it news application? Is it a loyalty app? Is it about nutrition tracking?

Organising files by type is for a machine, not for a human. Many times we work on a feature, and finding files to fix in multiple directories is a hassle. It’s also a pain if we plan to make a framework out of our project, as files are spread across many places.

Group by feature

A more reasonable solution is to organise files by feature. Files related to a feature should be placed together. And test files should stay close to the source files. Check out this article to learn more.

A feature can be related to login, sign up, onboarding, or a user’s profile. A feature can contain sub-features as long as they belong to the same flow. If we wanted to move the sub feature around, it would be easy, as all related files are already grouped together.

```
index.js
App.js
ios/
android/
src
  screens
    login
      LoginScreen.js
      LoginNavigator.js
    onboarding
      OnboardingNavigator
      welcome
        WelcomeScreen.js
      term
        TermScreen.js
      notification
        NotificationScreen.js
    main
      MainNavigator.js
      news
        NewsScreen.js
      profile
        ProfileScreen.js
      search
        SearchScreen.js
  library
    package.json
    components
      ImageButton.js
      RoundImage.js
    utils
      moveToBottom.js
      safeArea.js
    networking
      API.js
      Auth.js
  res
    package.json
    strings.js
    colors.js
```

```
        logo@2x.png
        logo@3x.png
        button@2x.png
        button@3x.png
  scripts
    images.js
    clear.js
```

Besides the traditional files `App.js` and `index.js` and the `ios1` and `android` folders, I put all the source files inside the `src` folder. Inside `src` I have `res` for resources, `library` for common files used across features, and `screens` for a screen of content.

As few dependencies as possible

Since React Native is heavily dependent on tons of dependencies, I try to be pretty aware when adding more. In my project I use just `react-navigation` for navigation. And I'm not a fan of `redux` as it adds unneeded complexity. Only add a dependency when you really need it, otherwise you are just setting yourself up for more trouble than value.

The thing I like about React is the components. A component is where we define view, style and behavior. React has inline style — it's like using JavaScript to define script, HTML and CSS. This fits the feature approach we are aiming for. That's why I don't use styled-components. Since styles are just JavaScript objects, we can just share comment styles in `library` .

src

`react-native init` sets up babel for us. But for a typical JavaScript project, it's good to organise files in the `src` folder. In my `electron.js` application IconGenerator, I put the source files inside the `src` folder. This not only helps in terms of organising, but also helps babel transpile the entire folder at once. Just a command and I have the files in `src` transpiled to `dist` in a blink.

```
babel ./src --out-dir ./dist --copy-files
```

Screen

React is based around components. Yup. There are container and presentational components, but we can compose components to build more complex components. They usually end in showing in the full screen. It is called `Page` in Windows Phone, `ViewController` in iOS and `Activity` in Android. The React Native guide mentions screen very often as something that covers the entire space:

> Mobile apps are rarely made up of a single screen. Managing the presentation of, and transition between, multiple screens is typically handled by what is known as a navigator.

index.js or not?

Each screen is considered the entry point for each feature. You can rename the `LoginScreen.js` to `index.js` by leveraging the Node module feature:

> there. The same `require('find-me')` line will use that folder's `index.js` file

So instead of `import LoginScreen from './screens/LoginScreen'` , we can just do `import LoginScreen from './screens'`.

Using `index.js` results in encapsulation and provides a public interface for the feature. This is all personal taste. I myself prefer explicit naming for a file, hence the name `LoginScreen.js`.

Navigator

react-navigation seems to be the most popular choice for handling navigation in a React Native app. For a feature like onboarding, there are probably many screens managed by a stack navigation, so there is `OnboardingNavigator` .

You can think of Navigator as something that groups sub screens or features. Since we group by feature, it's reasonable to place Navigator inside the feature folder. It basically looks like this:

```
import { createStackNavigator } from 'react-navigation'
import Welcome from './Welcome'
import Term from './Term'

const routeConfig = {
  Welcome: {
    screen: Welcome
  },
  Term: {
    screen: Term
    navigationOptions: {
      header: null
    }
  }

  export default OnboardingNavigator = createStackNavigator(routeCo
```

library

This is the most controversial part of structuring a project. If you don't like the name `library`, you can name it `utilities`, `common`, `citadel`, `whatever`...

This is not meant for homeless files, but it is where we place common utilities and components that are used by many features. Things like atomic components, wrappers, quick fixes function, networking stuff, and login info are used a lot, and it's hard to move them to a specific feature folder. Sometimes we just need to be practical and get the work done.

In React Native, we often need to implement a button with an image background in many screens. Here is a simple one that stays inside `library/components/ImageButton.js`. The `components` folder is for reusable components, sometimes called atomic components. According to React naming conventions, the first letter should be uppercase.

```
import React from 'react'
import { TouchableOpacity, View, Image, Text, StyleSheet } from '
import images from 'res/images'
```

```
      return (
        <TouchableOpacity style={styles.touchable} onPress={this.pr
          <View style={styles.view}>
            <Text style={styles.text}>{this.props.title}</Text>
          </View>
          <Image
            source={images.button}
            style={styles.image} />
        </TouchableOpacity>
      )
    }
  }

  const styles = StyleSheet.create({
    view: {
      position: 'absolute',
      backgroundColor: 'transparent'
    },
    image: {

  },
    touchable: {
      alignItems: 'center',
      justifyContent: 'center'
    },
    text: {
      color: colors.button,
      fontSize: 18,
      textAlign: 'center'
    }
  })
```

And if we want to place the button at the bottom, we use a utility function to prevent code duplication. Here is `library/utils/moveToBottom.js`:

```
function moveToBottom(component) {
  return (
    <View style={styles.container}>
      {component}
    </View>
  )
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'flex-end',
    marginBottom: 36
  }
})

export default moveToBottom
```

Use package.json to avoid relative path

Then somewhere in the `src/screens/onboarding/term/Term.js` , we can import by using relative paths:

```
import moveToBottom from '../../../../library/utils/move'
import ImageButton from '../../../../library/components/ImageButt
```

This is a big red flag in my eyes. It's error prone, as we need to calculate how many `..` we need to perform. And if we move feature around, all of the paths need to be recalculated.

Since `library` is meant to be used many places, it's good to dependency as this is extremely easy to fix.

The solution is to turn `library` into a `module` so `node` can find it. Adding `package.json` to any folder makes it into a Node `module` . Add `package.json` inside the `library` folder with this simple content:

```
{
  "name": "library",
  "version": "0.0.1"
}
```

Now in `Term.js` we can easily import things from `library` because it is now a `module`:

```
import React from 'react'
import { View, StyleSheet, Image, Text, Button } from 'react-nat
import strings from 'res/strings'
import palette from 'res/palette'
import images from 'res/images'
import ImageButton from 'library/components/ImageButton'
import moveToBottom from 'library/utils/moveToBottom'

export default class Term extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.heading}>{strings.onboarding.term.hea
        {
          moveToBottom(
            <ImageButton style={styles.button} title={strings.onb
          )
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center'
  },
  heading: {...palette.heading, ...{
    marginTop: 72
  }}
})
```

res

You may wonder what `res/colors`, `res/strings`, `res/images` and `res/fonts` are in the above examples. Well, for front end projects, we usually have components and style them using fonts, localised strings, colors, images and styles. JavaScript is a very dynamic language, and it's easy to use stringly types everywhere. We could have a bunch of `#00B75D` `color` across many files, or `Fira` as a `fontFamily` in many `Text` components. This is error-prone and hard to refactor.

Let's encapsulate resource usage inside the `res` folder with safer objects. They look like the examples below:

res/colors

```
const colors = {
  title: '#00B75D',
  text: '#0C222B',
```

res/strings

```
const strings = {
  onboarding: {
    welcome: {
      heading: 'Welcome',
      text1: "What you don't know is what you haven't learn",
      text2: 'Visit my GitHub at https://github.com/onmyway133',
      button: 'Log in'
    },
    term: {
      heading: 'Terms and conditions',
      button: 'Read'
    }
  }
}

export default strings
```

res/fonts

```
const fonts = {
  title: 'Arial',
  text: 'SanFrancisco',
  code: 'Fira'
}

export default fonts
```

```
const images = {
  button: require('./images/button.png'),
  logo: require('./images/logo.png'),
  placeholder: require('./images/placeholder.png')
}

export default images
```

Like `library`, `res` files can be access from anywhere, so let's make it a `module`. Add `package.json` to the `res` folder:

```
{
  "name": "res",
  "version": "0.0.1"
}
```

so we can access resource files like normal modules:

```
import strings from 'res/strings'
import palette from 'res/palette'
import images from 'res/images'
```

Group colors, images, fonts with palette

The design of the app should be consistent. Certain elements should have the same look and feel so they don't confuse the user. For example, the heading `Text` should use one color, font, and font size.

Below is my simple palette. It defines common styles for heading and `Text`:

res/palette

```
import colors from './colors'

const palette = {
  heading: {
    color: colors.title,
    fontSize: 20,
    textAlign: 'center'
  },
  text: {
    color: colors.text,
    fontSize: 17,
    textAlign: 'center'
  }
}

export default palette
```

And then we can use them in our screen:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center'
  },
  heading: {...palette.heading, ...{
    marginTop: 72
  }}
```

Here we use the object spread operator to merge `palette.heading` and our custom style object. This means that we use the styles from `palette.heading` but also specify more properties.

If we were to reskin the app for multiple brands, we could have multiple palettes. This is a really powerful pattern.

Generate images

You can see that in `/src/res/images.js` we have properties for each image in the `src/res/images` folder:

```
const images = {
  button: require('./images/button.png'),
  logo: require('./images/logo.png'),
  placeholder: require('./images/placeholder.png')
}

export default images
```

This is tedious to do manually, and we have to update if there's changes in image naming convention. Instead, we can add a script to generate the `images.js` based on the images we have. Add a file at the root of the project `/scripts/images.js`:

```
const fs = require('fs')

const imageFileNames = () => {
  const array = fs
```

```
    .map((file) => {
      return file.replace('@2x.png', '').replace('@3x.png', '')
    })

return Array.from(new Set(array))
}

const generate = () => {
  let properties = imageFileNames()
    .map((name) => {
      return `${name}: require('./images/${name}.png')`
    })
    .join(',\n  ')

const string = `const images = {
  ${properties}
}

export default images
`

fs.writeFileSync('src/res/images.js', string, 'utf8')
}

generate()
```

The cool thing about Node is that we have access to the `fs` module, which is really good at file processing. Here we simply traverse through images, and update `/src/res/images.js` accordingly.

Whenever we add or change images, we can run:

```
node scripts/images.js
```

```
"scripts": {
  "start": "node node_modules/react-native/local-cli/cli.js start
  "test": "jest",
  "lint": "eslint *.js **/*.js",
  "images": "node scripts/images.js"
}
```

Now we can just run `npm run images` and we get an up-to-date `images.js` resource file.

How about custom fonts

React Native has some custom fonts that may be good enough for your projects. You can also use custom fonts.

One thing to note is that Android uses the name of the font file, but iOS uses the full name. You can see the full name in Font Book app, or by inspecting in running app

```
for (NSString* family in [UIFont familyNames]) {
  NSLog(@"%@", family);

for (NSString* name in [UIFont fontNamesForFamilyName: family]) {
    NSLog(@"Family name:  %@", name);
  }
}
```

For custom fonts to be registered in iOS, we need to declare `UIAppFonts` in `Info.plist` using the file name of the fonts, and for

It is good practice to name the font file the same as full name. React Native is said to dynamically load custom fonts, but in case you get “Unrecognized font family”, then simply add those fonts to target within Xcode.

Doing this by hand takes time, luckily we have rnpm that can help. First add all the fonts inside `res/fonts` folder. Then simply declare `rnpm` in `package.json` and run `react-native link` . This should declare `UIAppFonts` in iOS and move all the fonts into `app/src/main/assets/fonts` for Android.

```
"rnpm": {
  "assets": [
    "./src/res/fonts/"
  ]
}
```

Accessing fonts by name is error prone, we can create a script similar to what we have done with images to generate a safer accession. Add `fonts.js` to our `scripts` folder

```
const fs = require('fs')

const fontFileNames = () => {
  const array = fs
    .readdirSync('src/res/fonts')
    .map((file) => {
      return file.replace('.ttf', '')
    })
```

```
const generate = () => {
  const properties = fontFileNames()
    .map((name) => {
      const key = name.replace(/\s/g, '')
      return `${key}: '${name}'`
    })
    .join(',\n  ')

const string = `const fonts = {
  ${properties}
}

export default fonts
`

fs.writeFileSync('src/res/fonts.js', string, 'utf8')
}

generate()
```

Now you can use custom font via `R` namespace.

```
import R from 'res/R'

const styles = StyleSheet.create({
  text: {
    fontFamily: R.fonts.FireCodeNormal
  }
})
```

The R namespace

This step depends on personal taste, but I find it more organised if we introduce the R namespace, just like how Android does for assets

> them using resource IDs that are generated in your project's `R` class. This document shows you how to group your resources in your Android project and provide alternative resources for specific device configurations, and then access them from your app code or other XML files.

This way, let's make a file called `R.js` in `src/res`:

```
import strings from './strings'
import images from './images'
import colors from './colors'
import palette from './palette'

const R = {
  strings,
  images,
  colors,
  palette
}

export default R
```

And access it in the screen:

```
import R from 'res/R'

render() {
  return (
    <SafeAreaView style={styles.container}>
      <Image
        style={styles.logo}
      <Text style={styles.title}>{R.strings.onboarding.welcome.ti
  )
}
```

Replace `strings` with `R.strings`, `colors` with `R.colors`, and `images` with `R.images`. With the R annotation, it is clear that we are accessing static assets from the app bundle.

This also matches the Airbnb convention for singleton, as our R is now like a global constant.

> 23.8 Use PascalCase when you export a constructor / class / singleton / function library / bare object.

```
const AirbnbStyleGuide = {
  es6: {
  },
}

export default AirbnbStyleGuide
```

Where to go from here

In this post, I've shown you how I think you should structure folders and files in a React Native project. We've also learned how to manage resources and access them in a safer manner. I hope you've found it useful. Here are some more resources to explore further:

- Organizing a React Native Project

Since you are here, you may enjoy my other articles

- Deploying React Native to Bitrise, Fabric, CircleCI
- Position element at the bottom of the screen using Flexbox in React Native
- Setting up ESLint and EditorConfig in React Native projects
- Firebase SDK with Firestore for React Native apps in 2018

If you like this post, consider visiting my other articles and apps ?
