React Native

A Beginner's Guide

J.P. Strydom

December 27, 2018

Preface

For this guide, I will go under the assumption that you have some experience with React and Redux. If that is not the case, I'd recommend that you familiarise yourself with these first, as it will shed a lot of light on some of the topics I'm about to discuss. If you are familiar with React and Redux, you can skip ahead to the next section.

Getting started with React

React and React Native are extremely similar. Once you fully understand the one, the other becomes effortless to learn. The best way to get to know pure React (no Redux just yet) is by going through this fantastic tutorial made by React themselves. Its an easy to follow tutorial that will teach you how to make a small multiplayer game using only React. It will only take you a few hours and it'll teach you all the React basics.

Getting started with Redux

Redux works exactly the same on React and React Native - there are literally no differences. If you'd like to learn more about Redux you can have a look at their introduction and at their basics. These will take you through all the Redux basics and will also briefly touch on how Redux integrates with React.

Integrating React & Redux

When it comes to using Redux with React or React Native there are a ton of resources! If you prefer reading you can have a look at this in-depth article - or if you're like me and prefer videos, I can HIGHLY recommend this Udemy course. This course is intended for people with no experience with React or Redux, and does a fantastic job at teaching both. It's by far the most popular, most complete, and most up-to-date resource available for learning React and Redux!

Resources

- React tutorial
- Redux introduction
- Redux basics
- In-depth React & Redux article
- Fantastic React & Redux Udemy course

Contents

Introduction

What is React Native?

React Native is a framework that enables you to build iOS and Android mobile apps using only JavaScript. It uses the same design as React, which allows you to quickly compose rich mobile UI's. You can find their documentation here.

Why use React Native?

- **Cross-platform development**: React Native allows you to simultaneously develop an Android and iOS application from one code-base. This saves a lot of development time as you don't have to learn and manage two separate front-end tech stacks.
- **Platform customisation**: React Native allows you to build platform-specific components, should you wish to. This can range from simple style differences between your platforms, to having entirely different UI's.
- **A lot of pre-built components**: React Native has a TON of pre-built components and API's! These range from visual components all the way to device API level components (such as device camera, local storage, and GPS functionality). These are all well-documented on their website here. You can also find an awesome list of external libraries here.
- **Familiarity**: As React Native closely follows the React design, it is extremely easy to use if you're familiar with React. This means that you can structure your React Native code-base the same way as your React code-base. You can think about it this way - React Native is the same as React, but instead of rendering JSX you'll be rendering React Native's JSX-like components.

```
// This React code for example:
render() {
    return (
        <div style={{ backgroundColor: '#98fb98' }}>
            <h2>{'Hello World!'}</h2>
        </div>
    );
}

// Will look like this in React Native:
import { View, Text } from 'react-native';
/* ... */
render() {
    return (
        <View style={{ backgroundColor: '#98fb98' }}>
            <Text>{'Hello world!'}</Text>
        </View>
    );
}
```

- **Popularity**: React Native has become extremely popular, with companies such as Facebook, Instagram, Uber, Tesla, Pinterest, Wix, and many more using it extensively. Due to this there are a lot of very useful and powerful packages out there to make your life easier. Plus if you do run into any errors, chances are, someone has already encountered and fixed it.
- **Fantastic documentation**: React Native has extremely thorough documentation - from getting started all the way to low level API descriptions. They've got it all! This should be your first stop if you have any questions or issues.

What to watch out for in React Native?

- **Platform inconsistencies**: Although React Native does a pretty amazing job at creating cross-platform applications, it's not perfect - and I don't blame the creators, the iOS and Android echo-systems are VERY different. This causes a few aspects and components to function differently on the two platforms, and some components are only supported on one of the platforms. But these inconsistencies are well-documented on their website, and there are usually external libraries that can handle some of those inconsistencies for you.

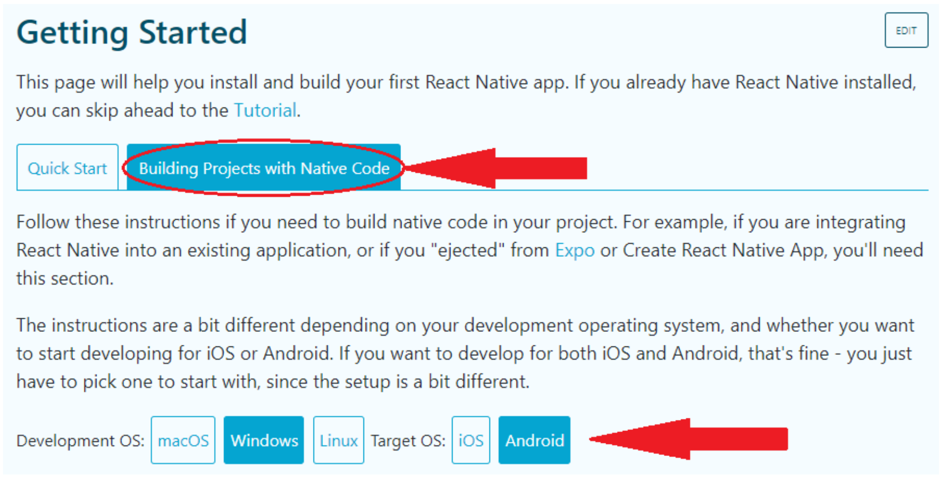
- **Still in the early phases**: React Native has still not reached v1 yet. Despite being very mature and so widely used, there might well be a few major changes to come in the near future. I doubt that the developers will introduce breaking changes, but it's something to keep in mind.
- **Visual inspectors are still immature**: Although React Native has some powerful debugging tools, its visual debugger is very finicky. In most browsers you have access to a detailed version on the DOM, which allows you to see very fine details about your site's layout and style. React Native unfortunately doesn't have such a nice tool (yet). React Native has a on-device inspector which lets you see some details about your current screen's DOM, but from my experience it's quite unreliable and will end up annoying you more than anything else.

Getting started

Setting up your environment & application

The best way to get started is to closely follow React Native's instructions. They have a *"Quick Start"* guide which shows you how to set up a small Expo app (which is similar to Create React App) and they have a *"Building Projects with Native Code"* guide which shows you how to set up a React Native app from scratch. Both of these guides will show you how to set up your environment and how to create and run a `"Hello World"` app. These guides can be found here.

- **Quick Start**: Following the *"Quick Start"* guide is useful for setting up quick prototypes, but it is quite limiting as it doesn't allow you to change the Android or iOS core files - which is necessary to integrate with a lot of packages and frameworks.
- **Building Projects with Native Code**: Following the *"Building Projects with Native Code"* guide will take slightly longer, but in the long run it'll give you much more control over your app. I would strongly recommend following it when setting up a production app. When following this guide, you need to specify your development OS and your target platform - the rest of the guide will vary depending on your selections. This guide also has useful tips and tricks on how to make the development process as easy as possible - I'd suggest reading through it a few times and going back to it once you've had some experience with React Native.


Getting Started

This page will help you install and build your first React Native app. If you already have React Native installed, you can skip ahead to the Tutorial.

Follow these instructions if you need to build native code in your project. For example, if you are integrating React Native into an existing application, or if you "ejected" from Expo or Create React Native App, you'll need this section.

The instructions are a bit different depending on your development operating system, and whether you want to start developing for iOS or Android. If you want to develop for both iOS and Android, that's fine - you just have to pick one to start with, since the setup is a bit different.

Development OS: macOS [✓ Windows] Linux Target OS: iOS [✓ Android]

*(**Note**: Something not mentioned in this guide is that you might have to accept the Android SDK license agreements before being able to build your app. This can easily be done by opening your project's android folder (`your-app/android`) in Android studio as an existing project - which should then automatically accept the agreements and install any missing dependencies for you.)*

Picking a navigation framework

Odds are your app will need more that one screen and some way to navigate between them (like a drawer/burger menu or tabs). Unfortunately React Native doesn't natively support navigation yet, so what are your options?

You could manage this yourself through state or Redux, but as your application grows this becomes increasingly more difficult to manage - and what about the back button on Android, or the mysterious iOS button?

Fortunately React Native recommends a few really good alternatives that manage most of the heavy lifting for you. These are `react-navigation` and Wix's `react-native-navigation`. So which one should you choose? Well... it depends. Essentially, `react-navigation` is more light weight and `react-native-navigation` is more powerful. But lets look at a more detailed breakdown:

Feature	`react-navigation`	`react-native-navigation`
Installation	Can be installed with NPM or Yarn	Can be installed with NPM or Yarn but requires additional native code alterations
Setup	Easy	Complicated
Customizability	Reasonable and consistent	Vast but inconsistent
Flexibility	Gives you the freedom to stick with any React structure	Forces you to use the framework's structure
Platform specifics	Supports each platform's native look and feel but it has to be implemented by the user	Does a great job at maintaining each platform's native look and feel for all navigation related aspects out of the box
Available components	Drawer/burger navigation, tab navigation, screen header bar, & navigation event listeners	Drawer/burger navigation, tab navigation, screen header bar, overlay modals, toast messages, notifications, & native navigation event listeners
Library support and documentation	Excellent	Reasonable

Personally, I have more experience with React Native Navigation - but I've been trying out React Navigation lately and I've been very impressed with it. It's simple to get going and offers a ton of flexibility. Their documentation is some of the best I've ever seen and it has a lot of useful examples. That being said, React Native Navigation is more powerful - but it's really painful to integrate into your application and it takes away a lot of your freedom. In exchange for this freedom however, it gives you the ability to customise OS specific aspects, such as the status bar and the Android on-screen buttons.

Both these navigation frameworks have their place and perform well when used correctly. I'd suggest going through their documentation and doing a bit of reading before picking one.

Guidelines & Suggestions

In this section I'll discuss a few useful libraries, packages, and architectures for React and React Native. These are by no means strict "rules", so take it with a pinch of salt - but I've found them to work really well, especially in very large applications.

Libraries & packages

The following libraries and packages are extremely useful to have in your application, especially during the development and CI process. They help improve the maintainability, extensibility, and easability of your application. You're probably already aware of most of them, but here's a refresher none the less.

Development (`devDependencies`)

- `eslint`: ESLint is a highly customizable tool for identifying and reporting on patterns found in ECMAScript/JavaScript code. It supports a multitude of plugins that work well with React, React Native, Prettier, Jest, and many more. Using ESLint, along with your desired plugins, will help ensure that your code is always optimised, up to standard, and follows best practices. This package is a MUST for any production JavaScript application.
- `prettier`: Prettier is a customizable and automated code formatter. It enforces a consistent style by parsing your code and re-printing it with its own rules that take the maximum line length into account, wrapping code when necessary. Prettier makes it extremely easy to keep your code consistent, which makes it very easy for developers to navigate your code-base. When using this package, you'll never have to worry about manual code formatting again.
- `jest`: What else is there to say about Jest that you haven't already heard? It's by far one of the best JavaScript testing frameworks out there and integrates seamlessly with React and React Native. It allows for fast and sandboxed testing, it comes with built-in code coverage reporting, it requires no configuration, and it comes with a very powerful built-in mocking library. This package is a MUST for any production JavaScript application.
- `enzyme`: Enzyme is a JavaScript testing utility for React and React Native that makes it easier to assert, manipulate, and traverse your React and React Native components' output. It works extremely well with Jest and allows you to fully test the entire React life-cycle of your components. This library makes visual component testing a breeze and is a MUST for any React and React Native production application.
- `lint-staged & husky`: These two packages work together to allow you to run any command or script against your stages git files. They essentially provide you with an easy means of setting up and customising pre-commit hooks. By automatically running your code-quality tools (such as ESLint, Prettier, Jest, etc.) it will ensure that no unwanted code or errors slip into your code-base.
- `generate-react-code`: This is a CLI scaffolding tool that allows you to generate state-full or state-less React or React Native components with just one command. It generates all the necessary code, and even some basic test code, for you. All the code it generates conforms to the Air BnB style guide naming and coding-style conventions - so if you use it to generate all your components, your code-base will be very clean and consistent. It can also generate all your Redux code for you (reducers, containers, actions, etc.), which all conform to the Ducks modular Redux pattern (we'll discuss this pattern in more detail later in this sub-section). This is a very useful tool that will help you development stay rapid and consistent, make sure to try it out.

Production (`dependencies`)

- `lodash`: Lodash, or "_" for short, is a modern JavaScript utility library. It comes with a ton of modular high-performing helper functions for arrays, strings, objects, and many more. Attached to this document is a presentation I gave on Lodash, please check it out for more details.
- `redux & react-redux`: Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time travelling debugger. It make it extremely easy to share and manage state across your React application.
- `redux-thunk`: Redux Thunk is a middleware for Redux. It allows you to do multiple asynchronous reads and writes to and from your application state. This enables you to do a lot of conditional checking and handling in your actions, which means you have much less logic to handle in your visual components. It also runs a promise-based architecture, which makes it easy to chain or wait for asynchronous methods and actions.
- `redux-promise`: Redux Promise is a middleware for Redux. If an action's payload is a promise, it will intercept that action and only dispatch it once the promise has been resolved. It will not dispatch anything if the promise is rejected. This makes it really easy to handle promise-based actions, such as network requests and device API calls.

Project file structure

While you're taking this sub-section with a pinch of salt, make sure to take an extra spoon for this sub-section. File structure is something everyone has an opinion about, and if you have something that works for you on your react projects, feel free to stick to it. This sub-section simply covers what I've found to work well in larger applications - perhaps you'll find it useful.

The general file structure will look as follows:

```
├── android
├── ios
├── node_modules
├── src
│   ├── assets
│   │   ├── fonts
│   │   ├── images
│   │   └── styles
│   ├── sub-components
│   │   ├── example-sub-component-one
│   │   └── example-sub-component-two
│   ├── components
│   │   ├── example-component-one
│   │   │   ├── example-local-sub-component-one
│   │   │   └── example-local-sub-component-two
│   │   └── example-component-two
│   ├── screens
│   │   ├── example-screen-one
│   │   └── example-screen-two
│   ├── services
│   │   ├── sub-services
│   │   │   ├── example-sub-services-one
│   │   │   └── example-sub-services-two
│   │   └── services.js
│   └── app.js
└── index.js
```

This may look daunting at first, but lets break it down into smaller chunks:

- `android` , `ios` , `node_modules` , & `index.js`: These are your standard directories and entrance file - nothing special here.
- `assets` : This is where you keep all your application's assets - fonts, images, logos, style-sheets, colour-sheets, animations, and whatever else you need.
- `sub-components` : This folder is for all your small, reusable, sub-components. These will be for small components such as buttons, dropdowns, toolbars, loaders, etc. One rule however is that a sub-component needs to be used in at least two other components or screens before it can live in this folder - if a sub-component is only used by one component, it should live inside that component as a local-sub-component.
- `components` : This folder is for your bigger components - some may be reusable and some may only be used once in your application. These will typically be constructed out of a bunch of smaller sub-components. Examples of such components are user-details-headers, video-players, profile-selectors, carousels, calendars, etc. As mentioned before, if a component uses a sub-component that only it needs (for example a carousel using a carousel-item sub-component), that sub-component should live inside the component and not inside the sub-component directory. I've found that this makes it easy for other developers to see which sub-components are reusable, and which are not.

- `screens`: This is where your application's screens will live. These screens will be added to your navigation framework in order to navigate amongst them in your application. These screens typically use components and sub-components to take care of visual rendering as their focus should be on handling the entire screen's state. Screens should for the most part be responsible for loading, editing, and storing the state related to it, which it will pass down to its components and sub-components.

- `services` & `services.js`: Utilities, helpers, jobs... This folder has gone by many names, but it's essentially for any shared functionality that's not directly related to visual components. Basically any and all services you would like to share across your applications - such as network-services, account-services, caching-services, and device API services. For React Native specifically, if you need to use a device service often (such as the location or local storage service), it makes a lot of sense to make a shared and standardised service to use across your application. These services can also be utilised by each other - for example, an account-service would use a network-service which might use a caching-service. These services can all be combined into one general `services.js` file like so:

```
// services.js
export { default as accountService } from './sub-services/account-service/account-service';
export { default as networkService } from './sub-services/network-service/network-service';
export { default as cachingService } from './sub-services/caching-service/caching-service';
```

  Which allows you to import all your services from the one file anywhere in your application like so:

```
// example-screen-one.js
import { accountService, networkService, cacheService } from '../services/services';
```

  This decouples your services from your application, which makes it easier to test and maintain them.

Ducks modular Redux pattern

To continue with the salt metaphor, this is the one section I'd recommend taking without any salt. The Ducks modular pattern is an exceptionally powerful state management pattern to use with React and Redux! It's extremely robust and can easily handle huge applications with a ton of state. The original proposal by Erik Rasmussen can be found here, but in this section I'll go into a lot more detail and explain exactly how the pattern can be used in your application. The Ducks modular Redux pattern, as its name suggests, is very modular - not only in its structure, but also in that it allows you to only use certain parts of it as you see fit. So whether you decide to use the pattern in its entirety, or just bits and pieces - this section will give you all the tools you'll need.

Redux refresher

Before we get started, lets make sure we have our React and Redux terminology in order. Redux is a global state management tool for React. Unlike React's component-level state, Redux's state is available throughout your entire application. This global state is referred to as a Redux **store**. This store can be accessed by connecting your plain React components to Redux - such a connected React component is typically called a Redux **container**. The Redux store can be edited by special methods called Redux **actions**, each of which has a unique type. Before such an action can edit the Redux store however, it first needs to be connected to Redux - this is usually also done inside your container component.

So when you hook your React component up to Redux (i.e. when you're creating your container), you give it access to certain store variables and you give it access to certain store-manipulating actions - and then you have a component that can read and write to and from the global store.

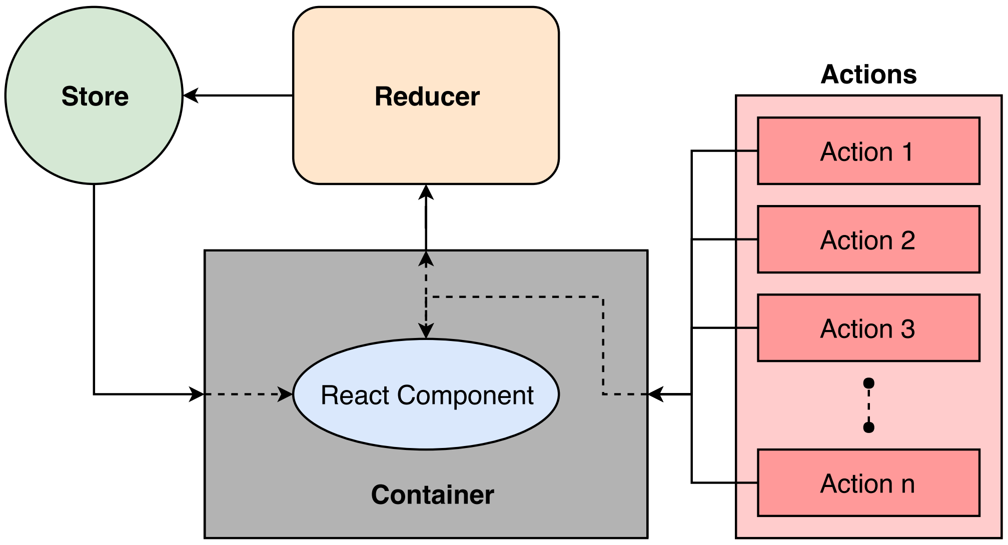
So how exactly do these store-manipulating actions work, and how do they know which particular state to manipulate? Well Redux uses something called a **reducer** which intercepts all actions and, based on their type, decides which bit of state they should update. Such a reducer will then, based on the action type and payload, update the appropriate state.

So let's look at what we have so far:

- **Store**: Your global application state
- **Action**: A special method that, when connected to Redux, can manipulate the store
- **Reducer**: A method that intercepts actions and updates the store accordingly
- **Container**: A React component that has been connected to the store (for reading, writing, or both)

Standard Redux pattern

Now that we're on the same page about Redux, let's have a look at how Redux is typically structured in an application. The standard Redux pattern can be described by the following diagram:



This structure will typically have the following file-structure:

- There is usually one reducer in an application
- Actions are scattered across an application. Some may live in a shared-actions file and some components will have a file for their own actions.
- React components are hooked up to Redux in one file. This means that a React component and its container lives in one file.

Although this all seems simple enough, it becomes increasingly difficult to manage as your application grows. Let's look at a few key issues in this pattern:

- As an application gets bigger, its reducer becomes massive and very difficult to maintain - not to mention it becomes hell to test!
- It becomes more and more difficult to find actions in your application, and often times they'll end up being unnecessarily rewritten.
- As React components and containers live as one entity, they become very big and extremely tedious to test thoroughly.
- It becomes very difficult to compartmentalise an application's state. This typically leads to applications with a complicated mixture of Redux state and React state.

Taking all this into consideration, we can see how this pattern leads to painful issues - just imagine the diagram above with hundreds of components! And here's where the Ducks modular pattern comes in...

Ducks modular Redux pattern

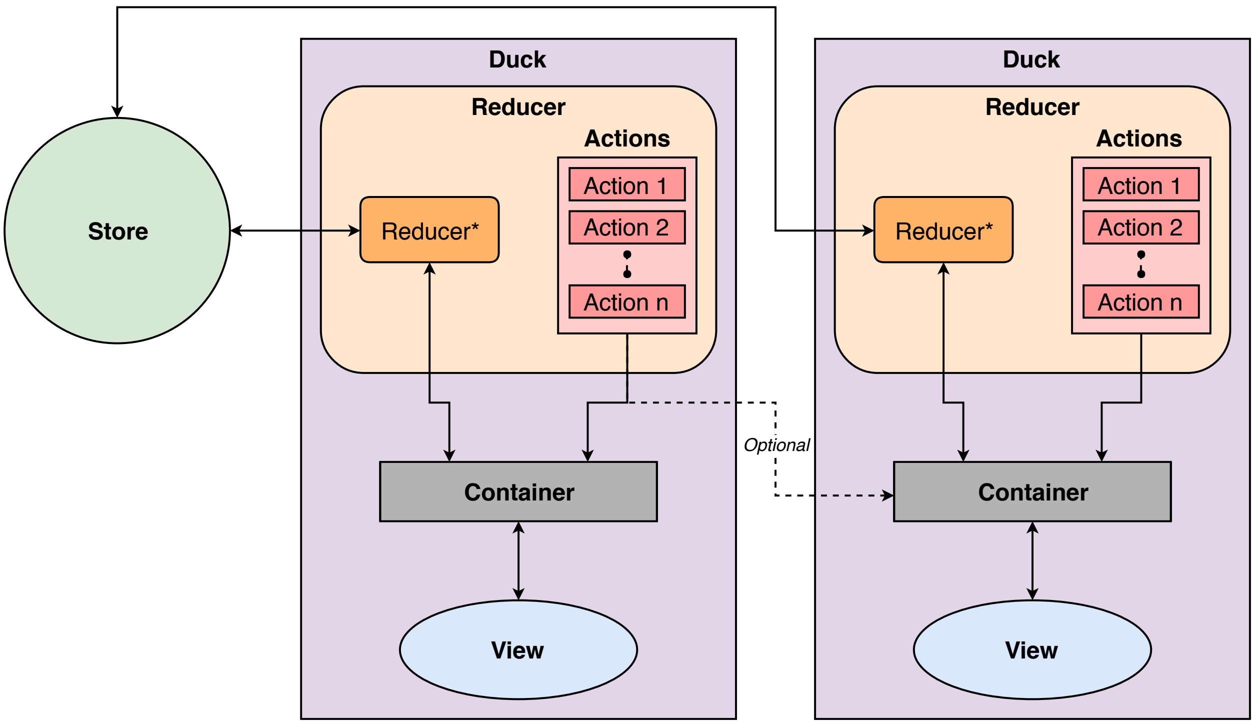
At its core, this pattern proposes splitting the whole Redux pattern into smaller, manageable, chunks - it calls these chunks **Ducks** (Ducks... Re***dux***... Get it? ☺). It does this in a few key ways:

- It focuses on segmenting your state and actions into small cohesive components. For example, all the state and actions related to user-account data should be kept together and separate from the rest of your application. The same can be said for settings-data, product-data, application-data, etc.
- It focuses on splitting Redux components (reducer, container, React component) up in such a way that they are easily manageable, reusable, and testable.
- It also makes sure that it is easy for these small Ducks to communicate with each other, should you need to.

From this we can derive some new and updated guidelines for the standard Redux components - a Duck should consist of some or all of the following components:

- **Reducer**: A reducer should contain all the state and actions related to a specific cohesive bit of state.
- **Container**: A container should only be responsible for connecting a React component up with the state and actions it needs (the state and actions may come from multiple reducers).
- **View**: A view is just a plain React component that knows nothing about Redux or state. It simply know it will receive certain props, it does not care where they come from.

These three components also describes the file-structure, as each Duck will consist of a reducer, a container, and a view file. This can all be illustrated by the following diagram:



*_**Note**_: Reducer here refers to the traditional Redux Reducer, but in the Ducks pattern the word Reducer refers to the collection of a standard Reducer and its related actions.

Let's look at some of the benefits of this pattern:

- Each Duck, and all of its components, have access to the entire application's state, but the state is now compartmentalised into a cohesive structure. For example, all the applications account-data and account-data actions will live inside the `accountReducer`, which can be accessed by any Duck.
- The various Duck components (reducer, container, and view) are modular which allow them to be easily maintained and tested.
- By having view components separate from containers allows you to use these view components as regular, state-less, React components.
- If you have multiple views in your application that need the same state but different visual presentations, you could hook all these views up to the same reducer.
- If you have a component that doesn't have its own unique state, but one that cares about the state of multiple Ducks - you can easily make a reducer-less Duck for that component that listens to all the state it needs to. An example of this would be a loading screen that listens to the loading state of each Duck and shows and hides itself accordingly.

Let's look at some very simple example code of such a Duck. Also pay close attention to the file naming pattern here - I like to give each file a sub-type (`xxx.reducer.js`, `xxx.container.js`, & `xxx.view.js`). I've found this helps to identify and navigate files quickly - and if you're using WebStorm, it'll even automatically display unique icons for each file sub-type. This also allows you to specify different test-coverage thresholds for each file sub-type. The respective files will look something like this:

```
// account.reducer.js
const reducerName = 'account';

const setUsernameActionType = reducerName + '/SET_USERNAME';
export const setUsernameAction = payload => ({
    type: setUsernameActionType,
    payload
});

const initialSate = {
    username: ''
};

export default function accountReducer(state = initialSate, action) {
    switch (action.type) {
        case setUsernameActionType:
            return { ...state, username: action.payload };
        default:
            return state;
    }
}
```

```
// account.container.js
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';

import AccountView from './account.view';
import { setUsernameAction } from './account.reducer';
// Here we could optionally include actions from other reducers

// Here we could also get state from other reducers ({ accountReducer, settingsReducer })
function mapStateToProps({ accountReducer }) {
    return { username: accountReducer.username };
}

function mapDispatchToProps(dispatch) {
    return bindActionCreators({ setUsernameAction }, dispatch);
}

export default connect(mapStateToProps, mapDispatchToProps)(AccountView);
```

```
// account.view.js
import React, { Component } from 'react';
import PropTypes from 'prop-types';

export default class AccountView extends Component {
    componentDidMount() {
        this.props.setUsernameAction('');
    }

    render() {
        const { username, setUsernameAction } = this.props;
        return (
            <input
                placeholder="Username"
                value={username}
                onChange={event => setUsernameAction(event.target.value)}
            />
        );
    }
}

AccountView.propTypes = {
    username: PropTypes.string,
    setUsernameAction: PropTypes.func
};
```

Now if you'd like to use a state-aware (i.e. Redux-connected) version on this Duck, you can import it from `account.container.js` - and if you'd like to use or test the plain old React component, you can import it from `account.view.js`. I'd also recommend giving the `xxx.view.js` sub-type to all your standard React components even if you don't plan on using them in a Duck, as it just makes it easy to identify them.

That's the Ducks pattern in a nutshell. I hope this section has been useful and that you've learnt something. I'd highly encourage you to try it out - and if you ever have any questions, don't hesitate to ask!

Pro-tips & Pitfalls

Installing libraries that require native code changes

Some libraries and packages require you to make changes to the Android and iOS native code. Usually this will be clearly stated in the library's/package's installation README along with a detailed guide on how to do so.

Most installation guides will suggest using the "`react-native link some-package`" command to do this for you, but from my experience this is a very unreliable method that hardly ever works for both Android and iOS. My suggestion would be to follow the installation guide's manual process, which should describe exactly which changes need to be made to which file (here is a good example).

Alternatively you could run the "`react-native link some-package`" command, but afterwards just make sure that the appropriate files have been changes by comparing the changes with that of the manual steps.

Another important thing to note here is that sometimes the order in which you compile native code modules might break your application. So if your ever run into cryptic errors after adding a new native code module, try moving the native module compile order around - I've wasted hours on exactly this issue, so hopefully it's something you'll be able to avoid.

Style sheets

React Native comes with a built in method for building very efficient style sheets. You can read more about these here. Making such a style sheet from a style object makes it possible to refer to it by ID instead of creating a new style object every time. You can also easily combine these style sheets much like you would with a CSS style sheet. Let's look at some examples of how you can use this:

```
// Creating a style sheet
const styles = StyleSheet.create({
    container: {
        borderRadius: 4,
        borderWidth: 0.5,
        borderColor: '#d6d7da',
    },
    title: {
        fontSize: 19,
        fontWeight: 'bold',
    },
    activeTitle: {
        color: 'red',
    }
});

// Mixing style sheets
<View style={styles.container}>
    <Text style={[styles.title, this.props.isActive && styles.activeTitle]} />
</View>
```

I would also recommend using a shared global style sheets for components such as for text, cards, buttons, images, etc. This will ensure that your application's look and feels stays consistent.

Responsive components

There is a huge range of resolutions available across the Android and iOS device spectrum - which makes it tricky to keep your application looking consistent on all devices. The best way to combat this is to utilise the flexbox system - which you should use as often as possible. The flexbox system is extremely powerful once you know how to use it - here is a very good tutorial that does a great job at explaining how the flexbox system works (even though the tutorial is for CSS, the concepts are exactly the same in React Native).

Unfortunately the flexbox system won't always work as certain components need physical dimensions to function properly (such as images, text, and borders). Fortunately, React Native has a built in device dimensions API. This enables you to scale components based on a device's width - as most devices have a 9:16 resolution, this gives you quite a good result across most devices. This dimensions API also allows you to add dimension event listeners, so you can listen for dimension or orientations changes should you wish to.

This dimensions API can be used as follows:

```
// Using the dimensions API to scale components
import { Dimensions, Image, Text } from 'react-native';

const { height, width } = Dimensions.get('window');

const styles = StyleSheet.create({
    wrapper: {
        height,
        width,
        display: 'flex',
        flexDirection: 'column',
        justifyContent: 'center',
        alignItems: 'center'
    },
    title: {
        fontSize: width * 0.05,
        fontWeight: 'bold'
    },
    image: {
        height: width * 0.5,
        width: width * 0.5
    }
});

<View style={styles.wrapper}>
    <Image style={styles.image} />
    <Text style={styles.title} />
</View>
```

Debugging

React Native gives you a bunch of useful debugging tools. You can read more about these here. Before I get into some of these tools, its really important to note that your application takes a BIG performance hit when running in debug/develop mode - so bear this in mind going forward.

Remote debugger

The remote debugger is by far the best tool at your disposal. It lets your application use the Chrome or Firefox developer tools, which means you can even use the React and Redux developer tools! It also lets you see your applications console, which is useful for picking up errors and warnings that the in-app tools might miss or suppress.

In-app developer tools

There are some useful developer tools available in-app. These can be accessed by pressing `cmd`+`M`/`ctr`+`M` on an Android emulator, `cmd`+`D` on an iOS emulator, or by shaking the device if you're running on a physical device... Yup you read that correctly!

All `console.error()` messages will be show to you as a red full-screen pop-ups, and all `console.warn()` messages will be show to you as as yellow toast message. This is why your Android app will ask you for overlay permissions when running in develop mode. This also means that `console.warn()` can be used as a means of in-app logging whilst debugging, but be careful as these "in-app logs" aren't always sequential.

You will also see an "Enable/Disable Live Reloading" and an "Enable/Disable Hot Reloading" option in the in-app developer tools. Enabling live reloading will cause your app to do a hard refresh (basically closing and restarting) every time you change some code, so that you don't have to manually do it. This is quite useful and quite reliable. Enabling hot reloading will cause your app to do a soft refresh (basically only updating the bit of code you're working on) every time you change some code. In theory this should allow you to see real-time changes as you code them, but sadly this feature isn't that reliable.

Physical device vs. emulator

React Native lets you develop on emulators and on physical devices. The ”Getting Started” section on React Natives site only shows you how to run on an emulated device, if you’d like to develop on a physical device you can find React Native’s guide on doing so here. So, which is better, physical device or emulator? Well, for the most part I’d highly recommend developing on a physical device if you can. There are tons of advantages to doing so lets look at some of those:

- **Better performance**: As mentioned before, your app will take a performance hit when running it in develop mode. This on top of the poor performance emulators typically have, will make your development experience very sluggish. You’ll feel like your app is performing very poorly even though it might very well not be the case at all.
- **Less resource intensive**: Emulators take up a lot of RAM and CPU capacity, so if you don’t have tons of those lying around your machine will suffer.
- **More accurate representation**: Emulators aren’t perfect, and as a result certain components won’t look and feel the same way they do on emulators as they would on physical devices. You could end up trying to fix certain visual bugs for hours only to realise that they don’t occur on actual devices at all (speaking from experience here ☹).
- **Better UX experience** We all use our hands and fingers to interact with our mobile devices, so when you’re using your mouse and keyboard on an emulator you won’t get a true feel for your applications UX. Navigating and interacting with your application might feel very strange when it’s only been tested with a mouse and keyboard.

There are, however, some advantages emulators offer - to mention a few:

- **Access to multiple OSs**: They allow you to test on older and newer operating systems that might not easily be available on physical devices
- **Access to multiple screen sizes**: They give you easy access to multiple screen sizes so you can see how responsive your application is
- **Access to multiple devices**: They allow you to test on devices you might not have available such as tablets and set top boxes

Application hosting

There are a few decent tools out there that handle automated deployments for you, but the best I’ve found is Bitrise. They offer automated deployments to both the iStore and the Google Play Store, they allow you to easily customise pipelines for various different builds, their user interface is extremely easy to use, and they can send you email and slack notifications for events you chose.

Conclusion

I hope the guide has been useful and that it’ll save you hours of painful debugging. If you have any questions or suggestions, feel free to email me at jp.strydom@entelect.co.za or find me on slack. Also make sure to check out the #react-native slack channel - if you have any questions, it’s a great place to start. Happy coding!