A Multi-platform Mobile App using Flux-like Architecture

Abstract

Flux is a design-pattern for building client-side software. Whereas many applications built with Model-View-Controller (MVC) architecture propagate data to the user view from multiple sources and directions (e.g. Controller to View, Model to Controller, etc.), Flux enforces unidirectional data flow. This strict data flow pattern allows for a declarative coding style, and can ease development and debugging.

The motivation for this project was for me to learn how to develop a cross-platform mobile app using a Flux-like architecture. The Flux-like framework I used is called Redux, developed by Dan Abramov in 2015. The app, a mobile client for StoryTime, was written in a framework called React Native. I chose React Native because it can run a single codebase on both Android and iOS hosts.

StoryTime is an app for Android and iOS that delivers high-quality, relevant, and free illustrated children's stories to families that don't have books at home. The user receives stories several times per week. The stories contain helpful reading tips. The app has full networking capabilities, including a chat interface, user account management, and dynamic content updates. I found thatRedux helped to abstract the handling of network data away from layout code. This is helpful because it isolates logical coding errors from the layout code. However, without prior knowledge of the Flux paradigm, it may be hard for other developers to understand data flow in the app. I believe the learning curve of Flux is justified because of how it separates data from views, which eases the development of large, complex apps.
App Description

The app is a iOS/Android client for StoryTime. Many key features of the client rely on a remote server, which manages user data. This server was developed in a separate CPSC 490 project. I will describe the app with screenshots of it running on both iOS and Android.

Fig 1. Login Screen

The app has full login/logout and user-creation flows.
Fig 2. Chat View
This is view where the user receives new stories from their teacher. If the user taps the ‘Tap Here!’ text, they go to the reader view (see Fig. 4).

Fig 3. Library View
This view displays all of the stories that the user has received from their teacher in the Chat View. The user can tap the icons to read the stories.
**Fig 4. Reader View**

The user can swipe through pages in the stories. If the user taps on the ‘?’ bubbles, they are given helpful reading tips.

**Fig 5. Account Management**

The user can manage their account. The app has full Spanish localization. (I speak fluent Spanish, so I did this myself.)
Implementation Details

1. React Native

The App was implemented in JavaScript (JS) using the React Native framework. Neither Android nor iOS apps are natively written in JS, but React Native transpiles the JS code so that it can run on the Android and iOS hosts natively.

Source `app/scenes/homePage/scenes/Messages/index.js`

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

import * as Animatable from 'react-native-animatable'

import STChat from './chat'

export default class MessagesContainer extends Component {
    render() {
        return {
            <View style={{ flex: 1 }}>
                <View style={styles.floatingContainer}>
                    <Text style={styles.loadingText}>
                        LOADING
                    </Text>
                </View>
                <Animatable.View animation="fadeIn" delay={350}
                    style={{ flex: 1 }}>
                    <STChat {...this.props} />
                </Animatable.View>
            </View>
        }
    }
}
```

*Fig. 6, a minimal component in the app. This is the layout for the ‘Chat View’ (see Fig. 2).*

Fig. 6 is a simple presentation component with an animated transition. Note how line 27 declares the STChat component, which is implemented and laid out in a separate module.
React Native has knowledge of all such declared modules, and is able to compose them on runtime.

2. Redux

![Diagram of MVC vs Flux](image)

Fig. 7 Comparison of data flow in MVC and Flux (Shen).

To understand how Redux works, it helps to understand data flow in traditional MVC projects. In MVC, application logic on both the client-side and backend are responsible for directly changing the state of the application. For example, clicking a button on the client-side may cause the view to render a loading indicator (a visual change). However, the click may also signal the backend to do computations/retrieve data, which when returned to the client will cause more visual (and state) changes, depending on the data returned.

By contrast, Redux enforces unidirectional data-flow, propagating all state in the app from a single source, called a Store. The store is essentially a single key-value pair object that sits in
memory. In Redux, the application logic never directly mutates the Store. Instead, all possible state-mutating interactions must be specified by dispatching 'actions', which can modify the Store. These state changes will then propagate down to the view layer, which then cause the visual change in the app.

Source ‘app/appRoot.js’

```javascript
import React, { Component } from 'react'
import { connect } from 'react-redux'
import PropTypes from 'prop-types'
import { NavigationProvider, NavigationContext } from '@expo/ex-navigation'
import SceneRoot from './scenes'
import PushController from './data/pushController'
import KeyboardMonitor from './components/keyboardWatcher'
import ConnectionMonitor from './components/connectionWatcher'
import Store from './createStore'
import Router from './router'
import { setGlobalHeight, setInternetAvailable } from './reducer'

const navigationContext = new NavigationContext({
  router: Router,
  store: Store,
});

@connect(
  state => ({
    appBooted: state.global.appBooted,
  }),
  dispatch => ({
    setWindowHeight: (h, keyboardUp) => {
      dispatch(setGlobalHeight(h, keyboardUp))
    },
  })),
export default class AppRoot extends Component {
  render() {
    if (this.props.appBooted) {
      return <NavigationProvider context={navigationContext}>
        <SceneRoot />
        <PushController />
        <KeyboardMonitor
          onChangeKeyboardStatus={this.props.setWindowHeight}
        />
        <NavigationProvider>
          <SceneRoot />
        </NavigationProvider>
    } else {
      return null
    }
  }
}
```

Fig. 8 A component in the app that is connected to the Redux Store.

Fig. 8 shows a component called AppRoot that is connected to the Redux Store. AppRoot
listens to changes in the state of the Redux store, but can also dispatch an action called `setGlobalHeight` that can mutate the state of the Store.

`AppRoot` is connected to the Redux store using the helper decorator `@connect`. Line 25 declares `AppRoot`’s dependency on the value of the Redux Store at `global.appBooted`. `AppRoot` displays the visual component `<SceneRoot>` on line 40, but only when the value of the Store at `global.appBooted` evaluates to true (see line 37).

`AppRoot` also contains a component call `<KeyboardMonitor>` on line 42. This is a non-visual component that watches for when the user summons the on-screen keyboard (usually by tapping an input-field in the app). When the Monitor detects that the Keyboard has appeared or disappeared, it dispatches the `setGlobalHeight` action with the dimensions of the screen minus the height of the keyboard. This action then updates the value of the Store, though the behavior of the update is abstracted away into a separate module (see line 15).

This example is minimally complex compared to other Redux-enabled components in the app, but it does demonstrate how data in the Redux Store is listened for, how it can be mutated, and how the details of the Redux Store state mutation are abstracted away from the layout.

**Analysis**

I was able to build a functional app using Redux and React Native. It is currently live on [Google Play](https://play.google.com) and the [AppStore](https://appstore.com). I built the app iteratively, starting with a basic feature set. I released the minimal version publicly, waiting for feedback, and then made adjustments. The original release of the app had a faux chat interface—just some default messages—and had minimal reading
interactivity. There also wasn’t any form of user account management. I was able to add in those features over time.

In the past, I’ve written large programs with many modules. However, sometimes distinct parts of the program began to depend on each other—these interdependencies are hard to debug. Following the Flux paradigm helped me to think about how to modularize my app in a way to avoid this problem. For example, the Story Reader needs to know what story the user has most recently selected. This information lives in the Redux Store. Any part of the app that wants to change the current selected story (e.g. from the Library View, Chat View, etc) needs only to dispatch the ‘SELECT_STORY’ action, a function that is shared by all callers (minimizes repeated code). The Story Reader has no knowledge of who dispatched the action or why. The Story Reader simply listens for the information about the currently selected story, and adjusts itself accordingly. Callers that dispatch the action also have no knowledge of the Story Reader, so we could swap the Story Reader component out for a fancier one. This modularity helps to isolate bugs since we don’t have to reason about direct dependencies between components. Each component either listens for data from the Store or sends data to the Store.

One major drawback of Redux in React Native apps is that the developer needs to know how Redux works to understand the source code. While it is convenient for the purpose of debugging to offload data-manipulation logic from the presentational code to the Store, this offloading is predicated on the developer understanding the pattern of data flow in Redux, which isn’t inherently obvious. Additionally, React Native has a binding library for Redux called ‘react-redux’, which is an additional layer of abstraction on top of Redux that adds to the complexity of integrating Redux. However, I have found Redux to be helpful in enforcing proper
modularity and to separate data from presentational views, so I believe that learning Redux is worth the investment.

Future Work

The biggest problem with the app is that it does not have any unit tests. In the future, I would like to add many unit tests, especially for code that mutates the state of the Redux Store. Unit tests will help to verify the integrity of any manipulation of data in the app, and also to make changes to the code with confidence.

I would also like to refactor the whole codebase to follow a strict, declarative style. When I first started writing the app, I didn’t know React Native coding conventions very well, so there are certain parts of the implementation at the over-modularized, under modularized, and others that don’t use good variable naming conventions. I wanted to practice building many different kinds of features into the app, so I didn’t spend enough time pruning old code to reflect more recent learnings.

In the future, I will also add more robust account-management features. I plan to build speech-recognition into the app to help struggling readers practice their pronunciation and fluency.

Acknowledgements

Thanks to Prof. Abadi for advising this project.
Some of the visual components used in the app, as well as many helper functions used for networking, computations, usage analytics, etc. were imported components from open source libraries. Every single such dependency is listed in the package.json file in the root directory of the source code.
Works Cited