What is Redux?

Redux is a standalone state management library, which can be used with any library or frameworkThe primary use of Redux is that we can use one application state as a global state and interact with the state from any react component is very easy whether they are siblings or parent-child.

Basic usage of Redux comes into picture when the app gets large and complex. In such apps, simple data management as parent-child becomes difficult using props. There are multiple components trying to communicate with multiple other components. In such cases, Redux comes in handy.

Redux can be broken down into few sections while building the application which is listed below.

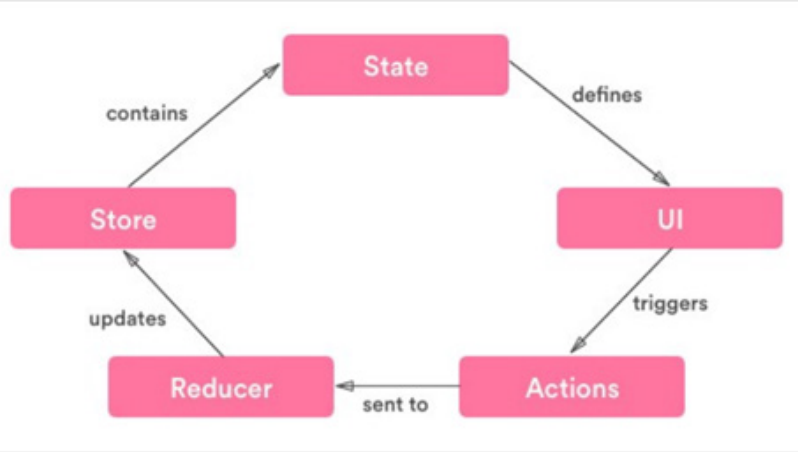
Actions: "are payloads of information that send data from your application to your store. They are the only source of information for the store." this means if any state change necessary the change required will be dispatched through the actions.

Reducers: "Actions describe the fact that something happened, but don't specify how the application's state changes in response. This is the job of reducers." when an action is dispatched for state change its the reducers duty to make the necessary changes to the state and return the new state of the application.

Store: with help of reducers a store can be created which holds the entire state of the application it is recommended to use a single store for the entire application than having multiple stores which will violate the use of redux which only has a single store.

Components (UI): This is where the UI of the application is kept.

Middlewares: middleware can be used for a variety of things, including asynchronous API calls. Middleware sounds much more complicated than it really is. The only way to really understand middleware is to see how the existing middleware works and try to write your own. we will cover middleware in the next blog.



Redux workflow in simple schematic

Steps for Implementing Redux in React Native app

We will follow these step-by-step instructions to create our React Native with Redux

Step 1: Create a Basic React Native app

Step 2: Running app on device

Step 3: Add simple counter into the App.js.

Step 4: Install the necessary packages to connect your app with redux.

Step 5: Create the necessary folders inside Root.

Step 6: Create Actions and Reducer function.

Step 7: Create a Redux Store.

Step 8: Pass the Store to the React Native app.

Step 9: Connect React Native app to Redux store.

Step 10: Test your app

1. Create a basic React Native app

First, make sure you have all pre-requisites to create a react-native app as per the official documentation.

Create a blank react-native app (Replace myApp with your own name)

```
$ react-native init myApp
```

This will create a basic React-native app which you can run in a device or sim ulator. (either Android or iOS)

Step 2: Running app on device

To run the iOS app, you should specify simulator

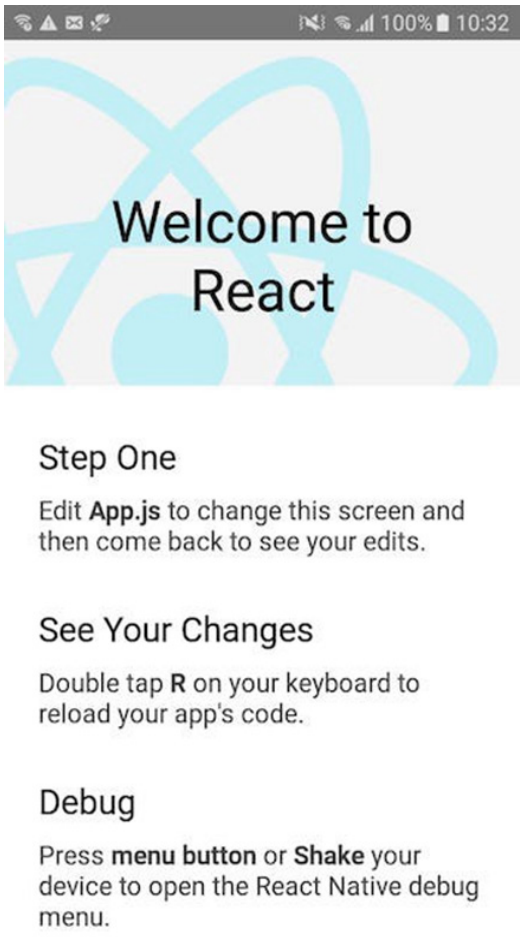
```
react-native run-ios --simulator="iPhone X"
```

To run Android app, you should start emulator first, either via Android Studio or

adb , then call

```
react-native run-android
```

You'll see the default start screen



Default start screen for React Native app

Step 3: Add simple counter into the App.js.

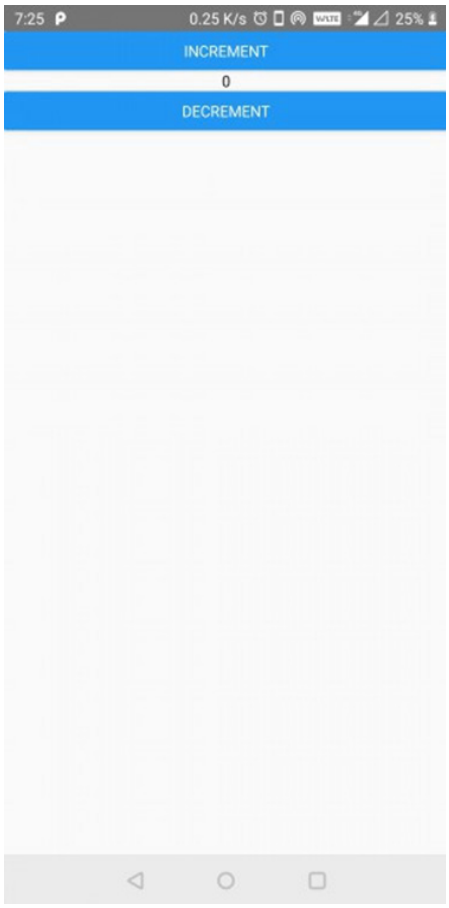
Now we'll changeApp.js to contain the UI required to render on the application with all the styling's as shown below. The UI will contain two button's and a text to show the count. All the class actions will be accessed using the this variable and all the state variable will be accessed using this.state which contain the current state of the app. Within the button onPress field, the actions will be called which are defined on the class in App.js.

So after adding this counter your App.js looks something like this.

```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 *
 * @format
 * @flow
 */
import React, { Fragment, Component } from 'react';
import {
  SafeAreaView,
  StyleSheet,
  View,
  Button,
  Text
} from 'react-native';

  class App extends Component {
  state = { count: 0 };
```

```
  decrementCount() {
    let { count } = this.state;
    count++;
    this.setState({
      count
    })
  }
  incrementCount() {
    let { count } = this.state;
    count++;
    this.setState({
      count
    })
  }
  render() {
    const { count } = this.state;
    return (
      <View styles={styles.container}>
        <Button
          title="increment"
          onPress={() => this.incrementCount()}
        />
        <Text>{count}</Text>
        <Button
          title="decrement"
          onPress={() => this.decrementCount()}
        />
```

```
            </View>
          );
        }
      };
const styles = StyleSheet.create({
        container: {
          flex: 1,
          justifyContent: 'center',
          alignItems: 'center'
        }

      });
Export default App;
```



Step 4: Install the necessary packages to connect your app with redux

In order to get redux to work with react-native few packages need to be installed as shown below

npm

```
npm install react-redux
npm install redux
```

yarn

```
yarn add react-redux
yarn add redux
```

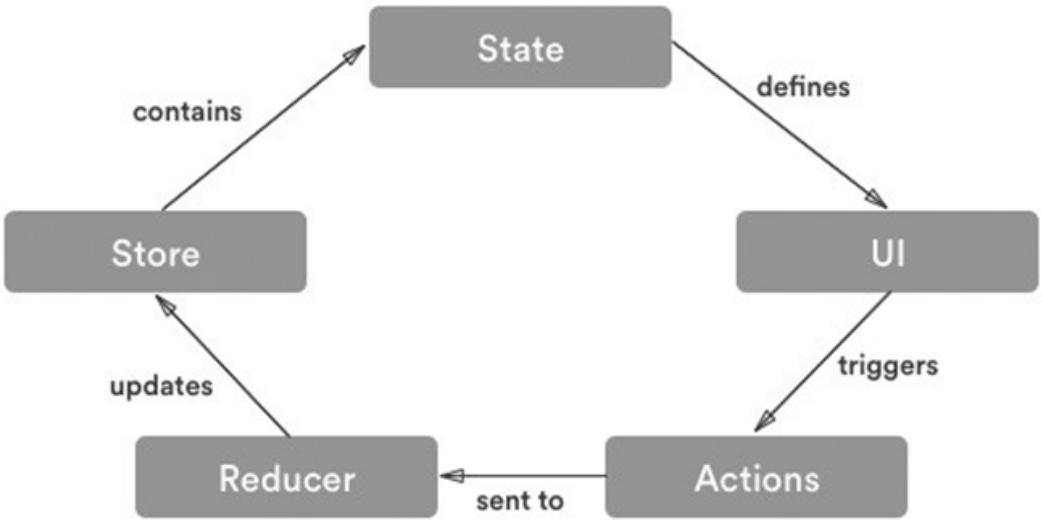
React Redux is the official React binding for Redux. It lets your React components read data from a Redux store, and dispatch actions to the store to update data.

As of React Native 0.18, React Redux 5.x should work with React Native. If you have any issues with React Redux 5.x on React Native, run npm ls react and make sure you don't have a duplicate React installation in your node_modules. We recom mend that you use npm@3.xwhich is better at avoiding these kinds of issues.

The React Redux docs are now published at https://react-redux.js.org .
If you want to know how it works, check out this link

Step 5: Create the necessary folders inside Root

Let's keep this image in mind to help us understand which parts of Redux are implemented at what step.



To work with redux you can follow this folder structure for better code quality

Create the following folders.

1. **actions**
2. **constants**
3. **reducers**
4. **components**

Step 6: Create Constants, Actions, and Reducer function

6.1 Inside the constants folder, create one file called index.js. Add the following line inside index.js.

```
export const COUNTER_CHANGE = 'COUNTER_CHANGE'
```

6.2 Inside the actions folder, create one file called counts.js. Add the following code inside counts.js.

Actions are JavaScript objects that represent payloads of information that send data from your application to your Redux store.

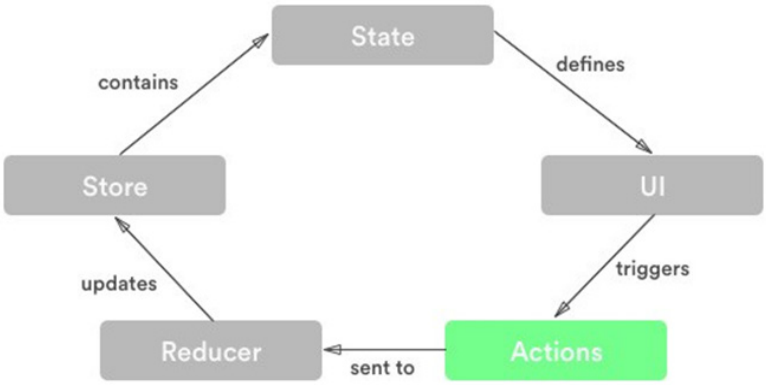
Actions have a type and an optional payload. In our case, the type will be COUNTER_CHANGE, and the payload count which we are assigning into our count variable in store.

```
import { COUNTER_CHANGE } from '../constants';
export function changeCount(count) {
return {
type: COUNTER_CHANGE,
payload: count
}
}
```

The changeCount() function returns an action. Now based on that action, reducers function's case is executed.

But, we need to connect this action to our App.js component somehow. Otherwise, we cannot add the data into the Redux store. Also, we need to first create a store. But before that, we also need to create a reducer function. So, first,

create a reducer → then create a store and → then connect the React Native application to the Redux store.



Actions created ...

6.3 Inside reducers function, create one file called countReducer.js. Add the following code inside it.

A reducer is a pure function that takes the previous state and an action as arguments and returns a new state. The reducer is instrumental in keeping the current state of count updated throughout our app as it changes.

```
import { COUNTER_CHANGE } from '../constants';
const initialState = {
count: 0
};
const countReducer = (state = initialState, action) => {
switch(action.type) {
case COUNTER_CHANGE:
return {
```

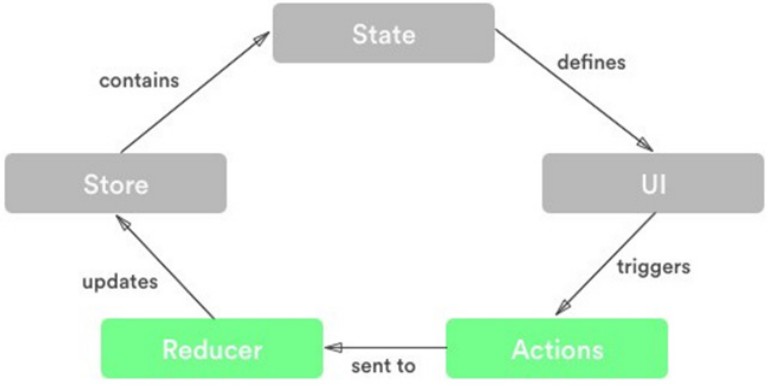
```
...state,
count:action.payload
};
default:
return state;
}
}
export default countReducer;
```

So, here, we have defined the function called countReducer that accepts the two arguments.

1. **state**
2. **action**

The first time, it will take the initial state of our application, and then we pass whatever argument, it takes that argument and operates based on the case execution. The second argument is action, which consists of type and payload. The payload is the count value, which assigned to count variable in store.

Remember here—we have returned a new state and not existing state. So we have modified the state in pure manner.



Actions and reducer created

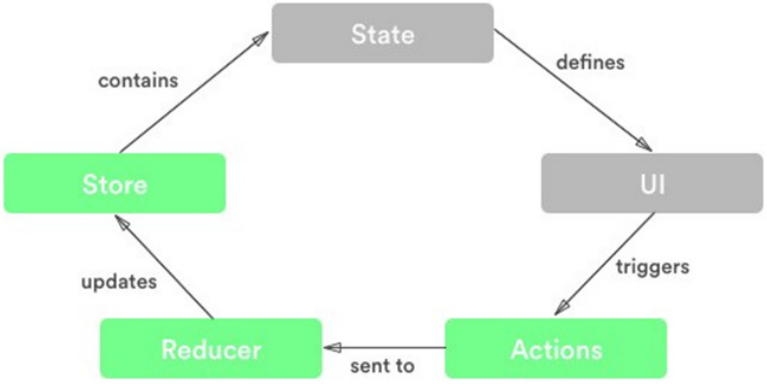
Step 7:-Create a Redux Store.

Inside the root folder, create one folder called store and inside this folder add one file called configureStore.js add the following code.

```
import { createStore, combineReducers } from 'redux';
import countReducer from '../reducers/countReducer';
const rootReducer = combineReducers(
{ count: countReducer }
);
const configureStore = () => {
return createStore(rootReducer);
}
export default configureStore;
```

Here, we have created the redux store and passed the reducer to that store. This store will ideally contain all the data that handles the app state.

The combineReducer function combines all the different reducers into one and forms the global state. So this is the global state of our whole application



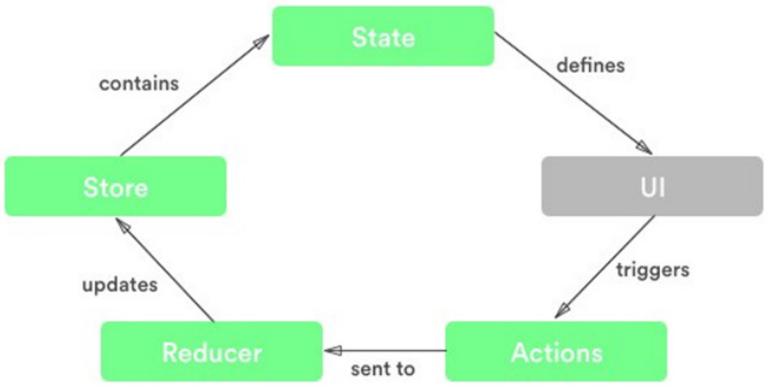
Step 8: Pass the Store to the React Native app.

Inside the root folder, you will find one file called index.js and inside that file add the following code.

```
import { AppRegistry } from 'react-native';
import React from 'react';
import App from './App';
import { name as appName } from './app.json';
import { Provider } from 'react-redux';


import configureStore from './store';


const store = configureStore()


const RNRedux = () => (
 <Provider store = { store }>
  <App />
 </Provider>
)


AppRegistry.registerComponent(appName, () => RNRedux);
```

view rawReduxIndex.js hosted with ❤ by GitHub

It is almost the same as React web application, in which we pass the Provider as a root element and pass the store and then via react-redux's connect() function, we can connect the any react component to redux store.



Store connected to the app

Step 9: Connect React Native app to Redux store.

Finally, we connect our App.js component to the Redux store. For that, we need the connect() function from the react-redux library.

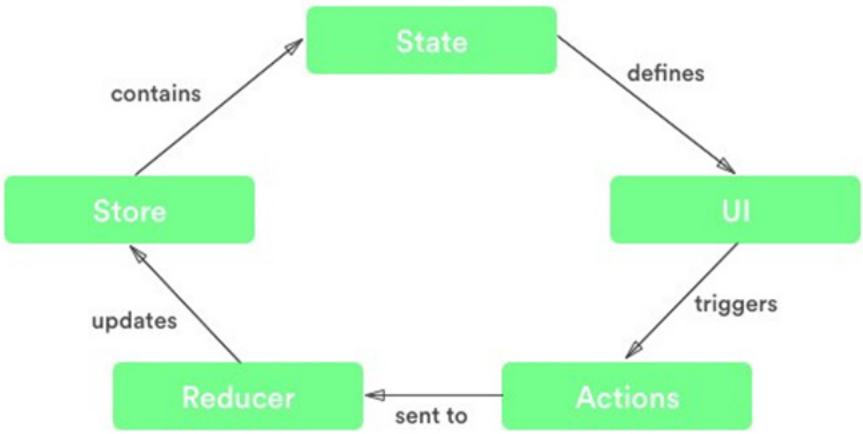
```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 *
 * @format
 * @flow
 */


import React, { Component } from 'react';
```

```
import {
  StyleSheet,
  View,
  Button,
  Text
} from 'react-native';
import { connect } from 'react-redux';
import { changeCount } from './actions/counts';
import { bindActionCreators } from 'redux';



class App extends Component {
  decrementCount() {
    let { count, actions } = this.props;
    count--;
    actions.changeCount(count);
  }
  incrementCount() {
    let { count, actions } = this.props;
    count++;
    actions.changeCount(count);
  }
  render() {
    const { count } = this.props;
    return (
      <View styles={styles.container}>
```

```
        <Button
          title="increment"
          onPress={() => this.incrementCount()}
        />
        <Text>{count}</Text>
        <Button
          title="decrement"
          onPress={() => this.decrementCount()}
        />
      </View>
    );
  }
};


const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  }
});


const mapStateToProps = state => ({
  count: state.count,
});
```

```
const ActionCreators = Object.assign(
  {},
  changeCount,
);
const mapDispatchToProps = dispatch => ({
  actions: bindActionCreators(ActionCreators, dispatch),
});
export default connect(mapStateToProps, mapDispatchToProps)(App)
```

view rawReduxApp.js hosted with by GitHub



And finally, store is now connected to UI via State

Connect which needs to be imported from react-redux and with the help of connect the properties will be mapped to the UI component.

mapStateToProps will contain all the state properties which will be available to the components of the app whatever properties that need to be accessed from the UI components need to be written into the **mapStateToProps** helper function.

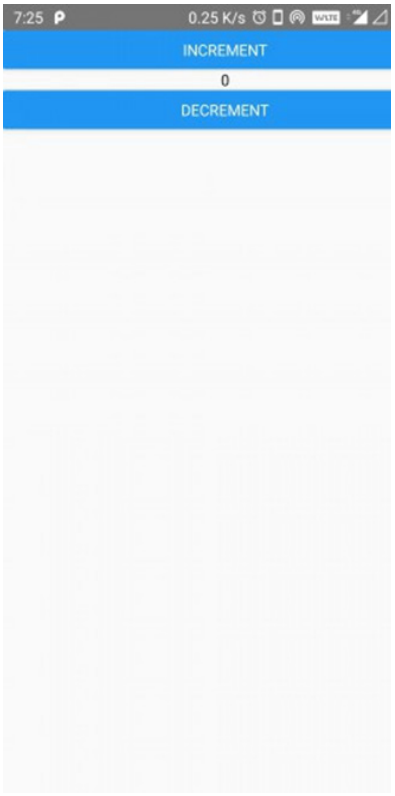
mapDispatchToProps will contain all the necessary functions wrapped with action creators which will be mapped to the props of the app and can be called directly. The directory containing the below file is **./actions/counts.js**

Here, when we click on the increment or decrement button, we get the count value from props and change it and then send it to the action with that value. Now that action returned the object with the action type and payload and based on the type the reducer will be executed and add that values inside the store.

Now, if the store's values are changed, then we need to update the UI based on the new values. That is why the **mapStateToProps** function is created. So, when the store's count get the new value, render function executed again and update the UI.

Step 10: Test your app:

Once you run the application the below UI will be rendered.



Basic Redux Example

Once you click on any of the buttons it will correctly increment and decrement the count. Although, what we just did is nothing fancy for an app, remember everything is going through the whole **"action-reducer-store-state"** flow. This flow can handle large and complex apps, so you don't get stuck in **"props"** loops.

(If you are from an **Angular** background, for ease of understanding, you can understand actions as something similar to methods in services which we can call from any component or page. Reducers are also similar functions, but they change the central store of the app. Store can be assumed as something similar to a $rootscope object which stores all the data at one place, and can be accessed from any page. Don't draw a direct comparison between these things, this is just to help you understand the concept in an easier way.)

Hope this blog helps you to implement redux with react native apps.

Conclusion

you learned how to implement Redux in your React Native app. You learnt what are reducers, actions and store, how the flow of data is managed in a Redux based app.