Components & State

What is a Component?

Components & State

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

Conceptually, React components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

Components & State

React class (stateful) vs function (stateless)

```
class MyComponent extends React.Component {
  render() {
    return (
     <Text>Hello World</Text>
    )
  }
}
```

```
const MyComponent = () => (
  <Text>Hello World</Text>
)
```

```
<MyComponent />
```

Components & State

props

React / React Native components accept arbitrary inputs (called "props") that can be used within the components.

```
class MyComponent extends React.Component {
  render() {
    return (
     <Text>{this.props.name}</Text>
    )
  }
}
```

```
const MyComponent = (props) => (
  <Text>{props.name}</Text>
)
```

```
<MyComponent name="Harry" />
```

Components & State

React class (stateful) vs function (stateless)

stateful - class
A stateful component either contains state or needs to hook into lifecycle methods

state is data that is private and fully controlled by the component.

```
import React from 'react';
import { Text } from 'react-native';

class App extends React.Component {
  state = { name: 'Chris' }

   // lifecycle methods

  render() {
    return (
      <Text>{this.state.name}</Text>
    )
  }
}

<App />
```

Components & State

React class (stateful) vs function (stateless)

stateless - function

```
import React from 'react';
import { Text } from 'react-native';

const App = () => (
  <Text>Hello World</Text>
);
```

```
<App />
```

Components & State

stateful vs stateless

stateless - es2015 arrow function with explicit return

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

const App = () => {
  return (
    <Text>Hello World</Text>
  );
};
```

Components & State

lifecycle concepts

A stateful React component's lifecycle contains distinct phases for creation and deletion. These are referred to as mounting and unmounting. You can also think of them as "setup" and "cleanup".

Components & State

lifecycle concepts - mounting

When we create (or mount) a stateful React component, we will trigger the component lifecycle which will create a new React / React Native component

Components & State

lifecycle methods - in order of initialization

Class creation

Mounting

```
constructor() or property initializers
componentWillMount()
render()
componentDidMount()
```

read more about property initializers

react documentation about property initializers

Components & State

mounting occurs and lifecycle methods are triggered whenever we create a React or React Native class

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

class App extends React.Component {
  constructor() {}

  componentWillMount() {}

  componentDidMount() {}

  render() {
    return(
      <View>
        <Text>{this.state.name}</Text>
      </View>
    )
  }
}
```

Components & State

lifecycle methods

Component destruction

Unmounting

`componentWillUnmount()`

Components & State

lifecycle methods

Updating component state and props

Updating state

```
shouldComponentUpdate()
componentWillUpdate()
render()
componentDidUpdate()
```

Updating props

```
componentWillReceiveProps()
shouldComponentUpdate()
componentWillUpdate()
render()
componentDidUpdate()
```

Components & State

Creating state

stateful - state using property initializers to declare state

```
import React from 'react';
import { Text } from 'react-native';

class App extends React.Component {
  state = { name: 'Chris' }

  render() {
    return(
      <Text>{this.state.name}</Text>
    )
  }
}
```

Components & State

Creating state

stateful - using constructor to declare state

```
class Person extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'Chris',
    };
  }

  render() {
    return (
      <View>
        <Text>{this.state.name}</Text>
      </View>
    );
  }
}
```

Components & State

composing reusable components - class

```
class Name extends React.Component {
  render() {
    return <Text>Dave</Text>;
  }
}

export default class extends React.Component {
  render() {
    return (
      <Name />
    );
  }
}
```

Components & State

composing reusable components - stateless function

```
const Name = () => <Text>Dave</Text>;

class App extends React.Component {
  render() {
    return (
      <Name />
    );
  }
}
```

Components & State

props

```
<Person name="Chris" />

<App data={['a', 'b', 'c']} />

<User loggedIn={false} isActive />

<User loggedIn={false} isActive={true} />
```

Components & State

stateful vs stateless

props - receiving props in a class

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

class App extends Component {
  render() {
    return (
      <Person name="Chris" />
    )
  }
}

class Person extends React.Component {
  render() {
    return(
      <View>
        <Text>{this.props.name}</Text>
      </View>
    )
  }
}
```

Components & State

stateful vs stateless

props - receiving props in a stateless function

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

class App extends Component {
  render() {
    return (
      <Person name="Chris" />
    )
  }
};

const Person = (props) => (
  <View>
    <Text>{props.name}</Text>
  </View>
);
```

Components & State

stateful vs stateless

props - destructuring in stateless component

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

class App extends Component {
  render() {
    return (
      <Person age={33} name="Chris" />
    )
  }
};

const Person = ({ name, age }) => (
  <View>
    <Text>{name} {age}</Text>
  </View>
);
```

Components & State

stateful vs stateless

props - receiving child components as props

```
const RedContainer = (props) => (
  <View style={{ backgroundColor: 'red' }}>
    { props.children }
  </View>
);

class App extends React.Component {
  render() {
    return (
      <RedContainer>
        <Text>Hello World</Text>
      </RedContainer>
    );
  }
}
```

Components & State

stateful vs stateless

Default props - class

```
class Person extends React.Component {
  static defaultProps = {
    name: 'Chris',
  }

  render () {
    return (
      <View>
        <Text>{this.props.name}</Text>
      </View>
    )
  }
};

export default App = () => <Person name="Jim" />;
```

Components & State

stateful vs stateless

Default props - stateless component

```
const Person = ({ name }) => (
  <View>
    <Text>{name}</Text>
  </View>
);

Person.defaultProps = {
  name: 'Mike'
};

export default App = () => <Person />;
```

Components & State

stateful vs stateless

propTypes

adding prop-types module

```
npm install --save prop-types
```

Components & State

stateful vs stateless

propTypes - class declaration

```
import PropTypes from 'prop-types';


class Person extends React.Component {
  static propTypes = {
    name: PropTypes.string.isRequired,
  }

  render() {
    return (
      <View>
        <Text>{this.props.name}</Text>
      </View>
    );
  }
};

class App extends React.Component {
  render() {
    return <Person name="Bob" />;
  }
}
```

Components & State

stateful vs stateless

propTypes - stateless component

```
import PropTypes from 'prop-types';

 const Person = ({ name }) => (
   <View>
     <Text>{name}</Text>
   </View>
 );

 Person.propTypes = {
   name: PropTypes.string.isRequired,
 };

 class App extends React.Component {
   static navigationOptions = {
     title: 'Playground',
   }
   render() {
     return <Person name="Bob" />;
   }
 }
```

Components & State

updating state

setState

setState enqueues changes to the component state and tells React that this component and its children need to be re-rendered with the updated state

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

class App extends React.Component {
  state = { name: 'Chris' }

  updateName = () => {
    this.setState({ name: 'Amanda' })
  }

  render() {
    return(
      <View>
        <Text onPress={this.updateName}>{this.state.name}</Text>
      </View>
    )
  }
}
```

Components & State

updating state

setState

setState enqueues changes to the component state and tells React that this component and its children need to be re-rendered with the updated state

SetState

```
shouldComponentUpdate()
componentWillUpdate()
render()
componentDidUpdate()
```

Components & State

setState

setState forces a rerendering of any component accessing the changed state.

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

class App extends React.Component {
  state = { name: 'Chris' }

  updateName = () => {
    this.setState({ name: 'Amanda' })
  }

  render() {
    return (
      <Person name={this.state.name} onPress={this.updateName} />
    )
  }
}

const Person = ({ onPress, name }) => (
  <View>
    <Text onPress={onPress}>{name}</Text>
  </View>
)
```

Components & State

setState

setState receives an
optional callback.

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

class App extends React.Component {
  state = { name: 'Chris' }

  updateName = () => {
    this.setState({ name: 'Amanda' }, () => {
      console.log('name is set!');
    })
  }

  render() {
    return (
      <Text onPress={this.updateName}>{this.state.name}</Text>
    )
  }
}
```

Components & State

setState

State Updates May Be Asynchronous

If you have multiple instances of setState being called at once, it may be a good idea to use the setState callback.

this.state may be updated asynchronously, so you should not rely on its value for calculating the next state.

```
class App extends React.Component {
  state = { value: 0 }

  increment = () => {
    this.setState((state, props) => ({
      value: state.value + 1,
    }));
  }

  render() {
    return (
      <Text onPress={this.increment}>{this.state.value}</Text>
    )
  }
}
```

Components & State

conditional rendering

```
state = { loggedIn: false }

login = () => {
  this.setState({ loggedIn: true })
}

render() {
  const { loggedIn } = this.state;

  if (loggedIn) {
    return <Text>Logged In</Text>
  }

  return <Text onPress={this.login}>Please Log In</Text>
}
```

Components & State

conditional rendering

```
state = { loggedIn: false }

login = () => {
  this.setState({ loggedIn: true })
}

render() {
  const { loggedIn } = this.state;

  return (
    <View>
      { loggedIn && <Text>Logged In</Text> }
      { !loggedIn && <Text onPress={this.login}>Please Log In</Text> }
    </View>
  );
}
```

Components & State

conditional rendering

creating components on the fly.

```
import {
  Platform,
  TouchableHighlight,
  TouchableNativeFeedback,
  View,
  Text
} from 'react-native';

render() {
  let Button = TouchableHighlight;
  if (Platform.OS === 'android') {
    Button = TouchableNativeFeedback;
  }

  return (
    <View>
      <Button onPress={console.log}>
        <Text>Hello!</Text>
      </Button>
    </View>
  );
}
```

Components & State

conditional rendering

filtering data in render.

```
state = {
  sport: 'basketball'
}

render() {
  let data = [{ name: 'chris', sport: 'baseball' }, { name: 'James', sport: 'basketball' }]

  data = data.filter(d => d.sport === this.state.sport)

  return (
    <View>
      {
        data.map(d => <Text>{d.name}</Text>)
      }
    </View>
  );
}
```

Components & State creating variables in render - class

```
class extends React.Component {
  state = {
    sport: 'basketball'
  }

  render() {
    const greeting = `${this.state.sport} is my favorite sport`;

    return (
      <Text>{greeting}</Text>
    );
  }
}
```

Components & State

creating variables in render - stateless component

```
const App = ({ sport }) => {
  const greeting = `${sport} is my favorite sport`;

  return (
    <Text>{greeting}</Text>
  );
};
```

Components & State

updating with forceUpdate()

```
state = { loggedIn: false };

updateState = () => {
  this.forceUpdate();
}

updateLoggedIn = () => {
  this.state.loggedIn = true;
}

render() {
  return (
    <View>
      <Text onPress={this.updateLoggedIn}>Log In</Text>
      <Text onPress={this.updateState}>Force Update</Text>
      { this.state.loggedIn && <Text>Logged In</Text>}
    </View>
  );
}
```

ReactJS fundamentals

lifecycle - componentWillMount

componenetWillMount is invoked immediately before mounting occurs. It is called before render(), therefore setting state synchronously in this method will not trigger a re-rendering.

Documentation recommends using either constructor or property initializers instead.

There is even a discussion about to whether this method should be deprecated.

```
componentWillMount() {
  this.setState({
    startDateTime: new Date(Date.now())
  });
}

render() {
  return (
    <Text>{this.state.startDateTime.toLocaleString()}</Text>
  );
}
```

Components & State

lifecycle - componentDidMount

componenetDidMount is called immediately after the render() method has taken place.

This is a great place to perform Ajax calls, timeouts, or any other operations.

This lifecycle method is used a lot in real world applications.

```
componentDidMount() {
  fetchFromApi()
    .then(data => this.setState({ data }))
}
```

Components & State

lifecycle - componentDidMount

componenetDidMount is called immediately after the render() method has taken place.

This is a great place to perform Ajax calls, timeouts, or any other operations.

This lifecycle method is used a lot in real world applications.

```
componentDidMount() {
  this.interval = setInterval(() => {
    this.setState({
      tick: this.state.tick + 1,
    });
  }, 1000);
}
```

```
render() {
  return (
    <Text>{this.state.tick}</Text>
  );
}
```

Components & State

lifecycle - componentWillUnmount

componenetWillUnmount is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests.

```
componentDidMount() {
  this.interval = setInterval(() => {
    this.setState({
      tick: this.state.tick + 1,
    });
  }, 1000);
}

componentWillUnmount() {
  clearInterval(this.interval);
}

render() {
  return (
    <Text>{this.state.tick}</Text>
  );
}
```

Components & State

lifecycle methods -
called in this order
when setState is called

state changes

```
shouldComponentUpdate()
componentWillUpdate()
render()
componentDidUpdate()
```

Components & State

lifecycle methods - called in this order when props have changed and when this is not an initial rendering.

prop changes

```
componentWillReceiveProps()
shouldComponentUpdate()
componentWillUpdate()
render()
componentDidUpdate()
```

Components & State

lifecycle - shouldComponentUpdate

`returns boolean`

shouldComponentUpdate is always called before render and enables the component to figure out if a re-rendering is needed or can be skipped.

```
state = {
  tick: 0,
}

shouldComponentUpdate(nextProps, nextState) {
  if (this.state.tick !== nextState.tick) {
    return true;
  }
  return false;
}
```

```
tick = () => {
  this.setState({
    tick: this.state.tick + 1,
  });
}

render() {
  return (
    <Text onPress={this.tick}>{this.state.tick}</Text>
  );
}
```

Components & State

lifecycle - componentWillReceiveProps

componentWillReceiveProps is only called when the props have changed and when this is not an initial rendering. This method enables us to update the state depending on the existing and upcoming props.

```
componentDidMount() {
  this.fetchFromApi(this.props.id);
}

componentWillReceiveProps(nextProps) {
  if (this.props.id !== nextProps.id) {
    this.fetchFromApi(nextProps.id);
  }
}

fetchFromApi(id) {
  fetch(`https://someurl/${id}`)
}
```

Component Patterns

Presentational & Container Components

Presentational & Container Components

A container component does data fetching and then renders its corresponding sub-component.

container components:

- Are concerned with how things work.
- May contain both presentational and container components inside but never have any styling applied.
- Provide the data and behavior to presentational or other container components.
- Call Flux actions and provide these as callbacks to the presentational components.
- Are often stateful, as they tend to serve as data sources.

Presentational & Container Components

presentational components:

- Are concerned with how things look.
- May contain both presentational and container components inside, and usually have some markup and styling of their own.
- Often allow containment via this.props.children.
- Have no dependencies on the rest of the app, such as Flux actions or stores.
- Don’t specify how the data is loaded or mutated.
- Receive data and callbacks exclusively via props.
- Rarely have their own state (when they do, it’s UI state rather than data).
- Are written as functional components unless they need state, lifecycle hooks, or performance optimizations.

Presentational & Container Components

Benefits:

- Better separation of concerns. You understand your app and your UI better by writing components this way.
- Better reusability. You can use the same presentational component with completely different state sources, and turn those into separate container components that can be further reused.
- Presentational components are essentially your app's "palette". You can put them on a single page and let the designer tweak all their variations without touching the app's logic. You can run screenshot regression tests on that page.

Presentational & Container Components

Say you have a component that displays comments. You didn't know about container components. So, you put everything in one place:

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

class CommentList extends React.Component {
  state = {comments: [] }

  componentDidMount() {
    fetchSomeComments(comments =>
      this.setState({ comments: comments }));
   }

  render() {
    return (
      <View>
        {this.state.comments.map(c => (
          <Text>{c.body}–{c.author}</Text>
        ))}
      </View>
    )
  }
}
```

Presentational & Container Components

Lets pull out data-fetching into a container component.

```
import React from 'react';
import { CommentList } from ‘./CommentList‘;

class CommentListContainer extends React.Component {
  state = {comments: [] }

  componentDidMount() {
    fetchSomeComments(comments =>
      this.setState({ comments: comments }));
   }

  render() {
    return (
      <CommentList comments={this.state.comments} />
    )
  }
}
```

Presentational & Container Components

Let’s rework CommentList to take comments as a prop.

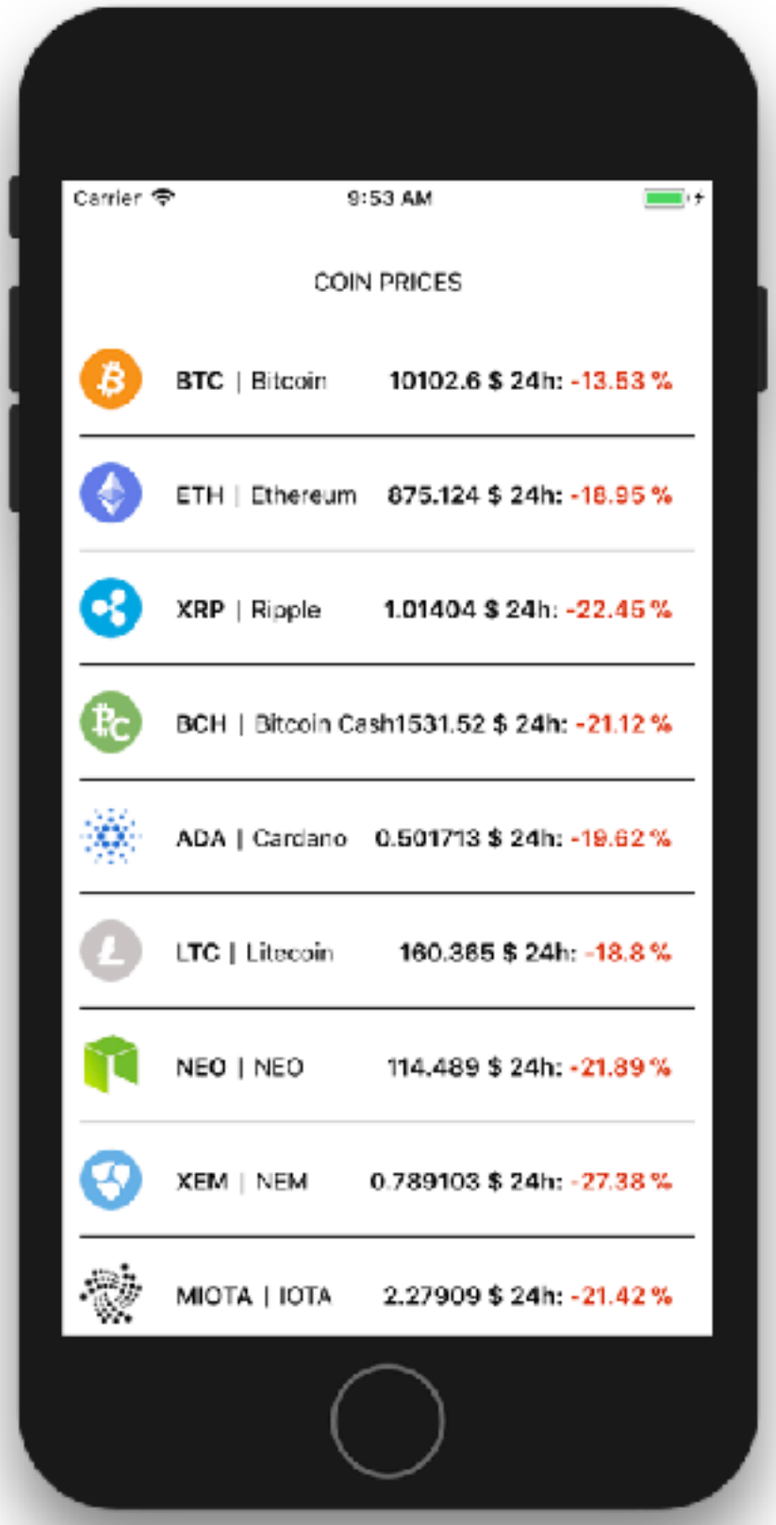
```
import { View, Text } from ‘react-native';

const CommentList = props => (
    <View>
      {props.comments.map(c => (
        <Text>{c.body}—{c.author}</Text>
      ))}
    </View>
);
```

Presentational & Container Components

So, what did we get?

- We've separated our data-fetching and rendering concerns.
- We've made our CommentList component reusable.
- We've given CommentList the ability to set PropTypes and fail loudly.

Lets build a screen



https://github.com/hgale/ComponentAndState

Lets build a screen

Pull request with changes:

https://github.com/hgale/ComponentAndState/pull/1

Higher-Order Components

Higher-Order Components

A higher-order component is just a function that takes an existing component and returns another component that wraps it.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

HOCs are common in third-party React libraries, such as Redux’s connect

Higher-Order Components

For example, say you have a CommentList component that subscribes to an external data source to render a list of comments:

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" is some global data source
      comments: DataSource.getComments()
    };
  }
    componentDidMount () {
    // Subscribe to changes
    DataSource.addChangeListener(this.handleChange);
  }

   componentWillUnmount () {
    // Clean up listener
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // Update component state whenever the data source changes
    this.setState({
      comments: DataSource.getComments()
    });
  }

  render() {
    return (
      <View>
        {this.state.comments.map((comment) => (
          <Comment comment={comment} key={comment.id} />
        ))}
      </View>
    );
  }
}
```

Higher-Order Components

Later, you write a component for subscribing to a single blog post, which follows a similar pattern:

```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
       blogPost: DataSource.getBlogPost(props.id)
    };
  }
   componentDidMount () {
    // Subscribe to changes
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount () {
    // Clean up listener
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // Update component state whenever the data source changes
    this.setState({
      blogPost: DataSource.getComments()
    });
  }

  render() {
    return ( <TextBlock text={this.state.blogPost} />);
  }
}
```

Higher-Order Components

CommentList and BlogPost aren't identical — they call different methods on DataSource, and they render different output. But much of their implementation is the same:

- On mount, add a change listener to DataSource.
- Inside the listener, call setState whenever the data source changes.
- On unmount, remove the change listener.

You can imagine that in a large app, this same pattern of subscribing to DataSource and calling setState will occur over and over again. We want an abstraction that allows us to define this logic in a single place and share them across many components. This is where higher-order components excel.

We can write a function that creates components, like CommentList and BlogPost, that subscribe to DataSource. The function will accept as one of its arguments a child component that receives the subscribed data as a prop. Let's call the function withSubscription:

```
const CommentListWithSubscription = withSubscription( {
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription( {
  BlogPost,
  (DataSource, props) => DataSource.getComments(props.id)
);
```

Higher-Order Components

The first parameter is the wrapped component. The second parameter retrieves the data we're interested in, given a DataSource and the current props.

When CommentListWithSubscription and BlogPostWithSubscription are rendered, CommentList and BlogPost will be passed a data prop with the most current data retrieved from DataSource:

```
function withSubscription(WrappedComponent, selectData) {
  // ...and returns another component...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... that takes care of the subscription...
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnMount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... and renders the wrapped component with the fresh data!
      // Notice that we pass through any additional props
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

How do we share state between screens?

Redux

Redux

Redux is a predictable state container for JavaScript apps.

Provides a single source of truth for any data that is being used in multiple parts of your application.

Redux

Basic Concepts

Out of the box, State / data is managed within a class

```
class App extends React.Component {
  state = {
    name: 'Chris'
  }

  render() {
    return (
      <Text>{this.state.name}</Text>
    );
  }
}
```

Redux

Basic Concepts

Data is also passed down as props

```
const Person = ({ name }) => <Text onPress>{name}</Text>

class App extends React.Component {
  state = {
    name: 'Chris'
  }

   updateState= (state) => this.setState({.///})

  render() {
    return (
      <Person updateState name={this.state.name} />
    );
  }
}
```

Redux

With Redux, all data is stored in an object and passed down as props.

```
store = {
  people: {
    data: []
  },
  locations: {

  },
  friends: {

  },
}
```

Basic Concepts

```
class App extends Component {
  render() {
    return (
      {
        this.props.people.data.map((person, index) => {
          return (
            <Text>{person.name}</Text>
          )
        })
      }
    )
  }
}
```

Redux

Provides a single source of truth for any data that is being used in multiple parts of your application.

This usually means only a single store for the entire application.

Redux

Redux store

Store is a JS Object

Usually key/value map

```
store = {
  people: {

  },
  locations: {

  },
  friends: {

  },
}
```

Redux

Redux store is composed of reducers (other javascript objects).

```
store = {
  people: {
      data: [],
    isFetching: false,
    isLoaded: false,
  },
  locations: {

  },
  friends: {

  },
}
```

```
// peopleReducer.js

const people = {
  data: [],
  isFetching: false,
  isLoaded: false,
}

const peopleStore = (state = people, action) => {
  // reducer logic goes here
}

export default peopleStore
```

Redux

Redux store is composed of reducers (other javascript objects).

Reducers takes in an old state and outputs a new state

```
// peopleReducer.js

const people = {
  data: [],
  isFetching: false,
  isLoaded: false,
}

export default peopleStore = (state = people, action) => {
  switch (action.type) {
    case: 'IS_LOADED':
      return {
        ...state,
        isLoaded: true,
        data: action.data
      }
  }
}
```

Redux

Data in components is read only.

Data can only be changed with Actions. These actions
get passed to our reducers, where the state is changed and
propagated down to the any connected components.

Redux

Actions are plain objects.

```
export function fetchAction() {
  return {
    type: 'IS_FETCHING',
  };
}
```

Redux

Actions may or may not take arguments.

```
// actions.js

export function getUserAction(id) {
  return {
    type: 'GET_USER',
    id: id,
  };
}
```

```
// peopleReducer.js

const people = {
  isFetching: false, peopleArray: [],
};

const peopleStore = (state = people, action) => {
  switch (action.type) {
    case 'IS_FETCHING':
      return {
        ...state,
        isFetching: true,
      };
    default:
      return state;
  }
};

export default peopleStore;
```

```
// configureStore.js

import { createStore } from 'redux'
import rootReducer from './reducers'

export default function configureStore() {
  let store = createStore(rootReducer)
  return store
}
```

```
// actions.js

export function fetchPeople() {
  return {
    type: 'IS_FETCHING',
  };
}
```

```
// in the app

import { fetchPeople } from './actions'

const App = () => (
  <Text
    onPress={this.props.fetchPeople}
  >
    Fetch People
  </Text>
);

// wrap the component in the connect function
// to connect the redux state
export default connect(
  (state) => ({
    people: state.people
  }),
  (dispatch) => ({
    fetchPeople: dispatch(fetchPeople())
  })
)(App)
```

Redux

reducers should be pure functions

Given the same input produce the same output.

No side effects.

Relies on no external mutable state.

Relies solely on the value of the arguments of the function.

Cannot modify the value of the values passed to them.

Redux

Impure functions

```
var count = 0;

function increaseCount(val) {
    count += val;
}
```

```
function getData(val) {
  fetch(`http://www.someurl.com/${val}`)
    .then(data => data.json())
    .then(json => json.people);
}
```

Redux

Pure functions

```
function increaseCount(count, val) {
    count += val;
    return count;
}
```

```
function filterDataArray(data, filterType) {
  return data.filter(d => d.type === filterType)
}
```

Redux

Example reducer

```
// reducers/people.js

import { ADD_PERSON, DELETE_PERSON } from '../constants';

const initialState = { peopleArray: [{ name: 'Chris' }] }

export default function peopleReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_PERSON:
      return {
        peopleArray: [...state.peopleArray, action.person],
      };
    case DELETE_PERSON:
      return {
        peopleArray: state.peopleArray.filter(p => p.name !== action.person.name),
      };
    default:
      return state;
  }
}
```

Redux

Example root reducer using combineReducers

The combineReducers helper function turns an object whose values are different reducing functions into a single reducing function you can pass to createStore.

```
// reducers/index.js

import { combineReducers } from 'redux'
import people from './people'

const rootReducer = combineReducers({
    people
})

export default rootReducer
```

Redux

Example actions

```
// actions.js

import { ADD_PERSON, DELETE_PERSON } from './constants';

export function addPerson(person) {
  return {
    type: ADD_PERSON,
    person,
  };
}

export function deletePerson(person) {
  return {
    type: DELETE_PERSON,
    person,
  };
}
```

Redux

Creating the store

`createStore` creates a Redux store that holds the complete state tree of your app.

```
// configureStore.js

import { createStore } from 'redux'
import rootReducer from './reducers'

export default function configureStore() {
  let store = createStore(rootReducer)
  return store
}
```

Redux

Creating the app entry point and tying the store into the app.

Provider makes the Redux store available to the connect() calls in the component hierarchy below.

```
// index.ios.js

import React from 'react'
import {
  AppRegistry
} from 'react-native'

import { Provider } from 'react-redux'
import configureStore from './configureStore'
import App from './app'

const store = configureStore()

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

AppRegistry.registerComponent('RNRedux', () => RNRedux)
```

Redux

So far we've used three redux specific methods. `combineReducers` and `createStore` from redux `Provider` from react-redux

Redux

Wiring redux up to the UI

```
// app.js

import React from 'react';

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

class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <Text>Hello from my app</Text>
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    marginTop: 100,
  }
})

export default App
```

Redux

Hooking the component into redux store using the `connect` method from react-redux

`connect` connects a React component to a Redux store.

```
// app.js

import { connect } from 'react-redux'

///

export default connect(
  (state) => ({ people: state.people }),
)(App)
```

Redux

Manipulating the redux state

update imports

```
// app.js
import { connect } from 'react-redux';
import { addPerson } from './actions';
```

Redux

Manipulating the redux state

update UI

```
// app.js

<View style={styles.container}>
  <Text style={styles.title}>People</Text>

  <TextInput
    onChangeText={text => this.updateInput(text)}
    style={styles.input}
    value={this.state.inputValue}
    placeholder="Name"
  />

  <TouchableHighlight
    underlayColor="#ffa012"
    style={styles.button}
    onPress={this.addPerson}
  >
    <Text style={styles.buttonText}>Add Person</Text>
  </TouchableHighlight>

  {
    this.props.people.peopleArray.map((person, index) => (
      <View key={index} style={styles.person}>
        <Text>Name: {person.name}</Text>
      </View>
    ))
  }
</View>
```

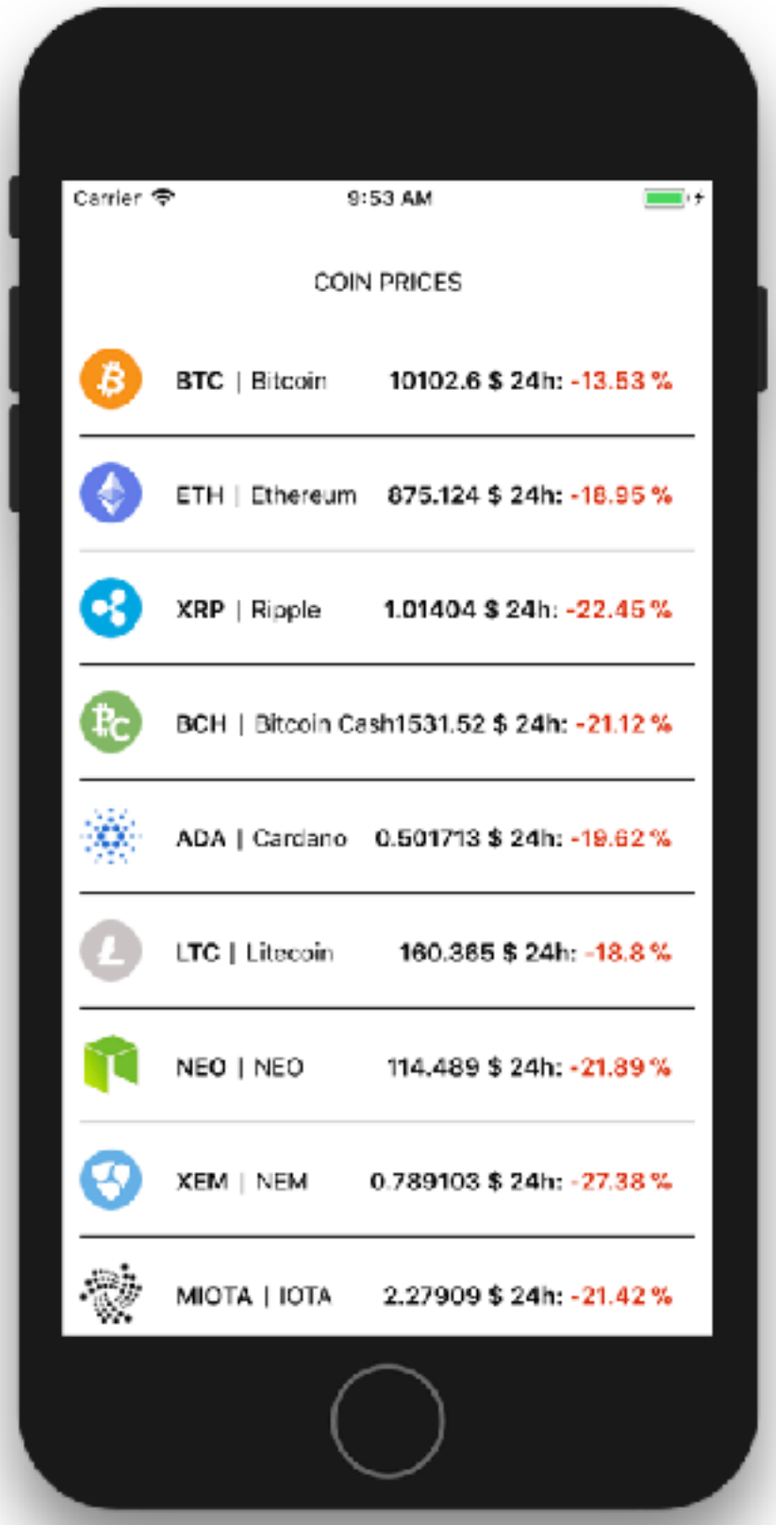
Redux

Manipulating the redux state

update connect function

```
// app.js

function mapStateToProps (state) {
  return {
    people: state.people
   }
}

function mapDispatchToProps (dispatch) {
  return {
    dispatchAddPerson: (person) => dispatch(addPerson(person)),
  }
}

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(App)
```

Rebuild With Redux



https://github.com/hgale/ComponentAndState

Redux

An example of a more complicated app (single screen + tabbed navigation) that uses React Native Navigation, Redux:

https://github.com/hgale/ReactNativeNavigationExample