React Native - Basics - State

- Dr Nick Hayward

A brief intro to the basics of *State* in React Native app development.

Contents

Intro

React and React Native manage data using either `props` or `state`.

`Props` are set by the parent, and remain immutable for a component's lifetime.

If we need to modify data whilst an app is running, we can use `state`.

With React, there is distinct pattern to `state` usage, which is as follows,

- `state` should be initialised in the *constructor* for a component &c.
- `setState` may then be used to modify and update `state`

General usage

We can use `state` to manage data within an app, from basic UI updates to data from a remote DB or API.

As the data is updated, we can likewise modify `state` within our app.

`state` may be managed within a React Native app or by using containers such as *Redux*. Such containers help manage and control data flow within an app, in particular as the app scales to meet greater requests on `state`.

By introducing a container such as *Redux*, we circumvent direct management of `state` with `setState`. Instead, `state` updates rely upon *Redux* management.

`state` usage

A basic example of `state` usage and maintenance may set a static message using `props`, and then update a notification using `state`.

e.g.

```
// import React, Component module as Component from base React
import React, { Component } from 'react';
// import Text as Text &c. from React Native
import { AppRegistry, Text, View } from 'react-native';

// abstracted component for rendering *tape* text
class Tape extends Component {
  // instantiate object - expects props parameter, e.g. text & value
  constructor(props) {
    // calls parent class' constructor with `props` provided - i.e. uses Component to
setup props
    super(props);
    // set initial state - e.g. text is shown
    this.state = { showText: true };
```

```
    // set timer for tape output
    setInterval(() => {
      // update state - pass `updater` and use callback (optional for setState)
      // `updater` prevState is used to set state based on previous state
      this.setState(prevState => {
        // setState callback - guaranteed to fire after update applied
        return { showText: !prevState.showText };
      });
    }, 1500);
  }

  // call render function on object
  render() {
    // set display boolean - showText if true, else output blank...
    let display = this.state.showText ? this.props.text : ' ';
    return (
      // output text component with text from props or blank...
      <Text>{display}</Text>
    );
  }
}

export default class TickerTape extends Component {
  render() {
    return (
      // create View container - then instantiate Tape objects - pass text props
      <View>
        <Tape text="welcome to the test state app!" />
      </View>
    );
  }
}

// register app Root - component for appKey, component to run (component provider to
return...)
AppRegistry.registerComponent('BasicAppState', () => TickerTape);
```

In this example, we define the required imports for React and React Native, including existing components we need for this basic app.

- `AppRegistry` - entry point for JavaScript to enable a React Native app to run...
  - added as part of `init` command for React Native apps
- `Text` - used to display text within an app
- `View` - a UI container for displaying content (basic requirement for UI development with React Native)
  - supports layout structures with flexbox, style, touch, accessibility...

Then, we define our required custom components. One abstracted for broader re-use, the other for use in the current specific app.

The `Tape` class is an abstracted component for rendering passed text with a timer. The constructor for this class instantiates an object with passed `props`, e.g. passed text for rendering.

Within this constructor, `super` is used to call the parent class' constructor with `props` provided - i.e. uses Component to setup props. We can then set the initial `state` on the instantiated object, which will default to `true` for this component.

Then, we can call the JS function `setInterval()` to create a basic timer, which creates the simple UI animation. A delay is set to 1500 milliseconds.

The main focus of this function is to modify `state`, which will trigger an update. So, we can call `setState` on the current object. This function is called with a passed `updater` and a callback.

`updater` prevState is available for the `setState` function, and is used to set state based on previous known state. `state` itself may not necessarily be triggered immediately, and React may delay an update until it has a worthwhile queue. However, we can call an immediate callback as this `setState` is registered. In this example, we simply change the boolean value for `showText`. e.g. false to true, true to false.

We can then call the `render()` function on the current object, outputting text passed using `props`. We simply check the boolean value in `state`, and then render a `text` component with `props` text or a blank space.

The default component (exported module) for this app is set to the `TickerTape` class, which renders a `view` container with the custom component for `Tape`. We can simply pass `props` for the required text to render.

References

- MDN - super