

Comp 322/422 - Software Development for Wireless and Mobile Devices

Fall Semester 2019 - Week 9

Dr Nick Hayward

DEV week assessment

Course total = 25%

- begin development of a mobile application from scratch
 - **NOT** a responsive website viewed on a mobile device
 - must apply technologies taught up to and including DEV week, e.g.
 - Apache Cordova, React Native, &c.
 - combine technologies taught to fit your mobile app...
- can be platform agnostic (cross-platform) or specific targeted OS, e.g.
 - cross-platform app that builds for Android and iOS
 - targeted build for Android or iOS
 - consider choice, and explain why?
- outline concept, research conducted to date
- consider applicable design patterns
- are you using any sensors etc?
 - how, why?
- prototyping
 - demo current prototypes
 - any working tests or models etc
- anything else to help explain your mobile app...

DEV Week Demo

DEV week assessment will include the following:

- brief presentation or demonstration of current project work
 - *~ 10 minutes per group*
 - *analysis of work conducted so far*
 - e.g. during semester & DEV week
 - *presentation and demonstration...*
 - outline mobile app
 - show prototypes and designs
 - explain what does & does not work
 - ...

React Native - Layout and Styles

flex and CSS inspired

- UI structure in React Native is achieved using *Flexbox*
 - *originally defined for web development*
- currently used to help with UI layout patterns and designs
- *Flexbox* usage slightly different for React Native
 - *no CSS syntax for styles*
- React Native styles are written, manipulated, and contained in JavaScript
- benefits of component structure to store and abstract our UI layouts and styles

React Native - Layout and Styles - add some flex

intro

Flexbox works the same way in React Native as it does in CSS on the web, with a few exceptions. The defaults are different, with `flexDirection` defaulting to column instead of row, and the `flex` parameter only supporting a single number.

- React Native uses the *flexbox* algorithm
 - *specify layout and design for its components, and their children*
- benefit of *flexbox* layouts
 - *adaptation to multiple screen sizes, aspect ratios, and orientations...*
- for React Native, there tends to be three predominant uses
 - *alignItems*
 - *flexDirection*
 - *justifyContent*

React Native - Layout and Styles - add some flex

flexDirection

- by defining a component's `flexDirection`
 - *setting organisational pattern for its subsequent children*
 - *might be set to a horizontal row or a vertical column*
- by default, `flexDirection` will be set to a column
 - *change to row*

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
  },
});
```

- a `View` with the style for container
 - *will use all of the available screen space*
 - *and render its child components in a row pattern*
 - *cascading from row to row...*

React Native - Layout and Styles - add some flex

justifyContent

- then update this style to define how child components start to fill each row
 - *setting their `justifyContent` value*
- options include
 - *`flex-start`*
 - *`flex-end`*
 - *`space-around`*
 - *`space-between`*

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'flex-end'
  },
});
```

React Native - Layout and Styles - add some flex

alignItems

- align items offers a simple, complementary option to `flexDirection`
- if the direction for the primary axis, set using `flexDirection`, is *column*
 - *alignItems* will define the secondary axis as *row*
- options include
 - *flex-start*
 - *flex-end*
 - *center*
 - *stretch*
- caveat to using the *stretch* value
 - need to ensure no fixed dimensions set for any children of flex component

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    justifyContent: 'flex-start',
    alignItems: 'stretch',
  },
});
```

more layout options

- further options may be specified as props
 - *add to a given component or stylesheet...*
- full details can be found at the following URL,
 - *Layout Props*

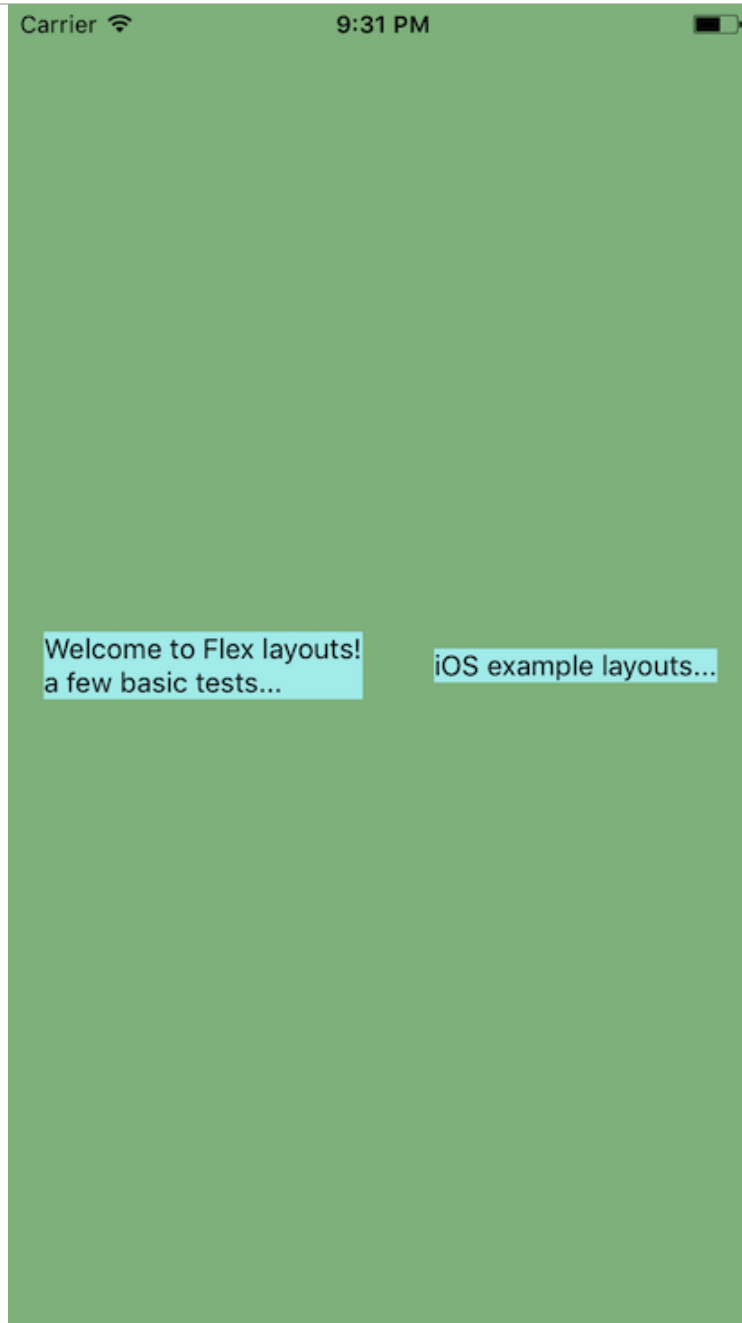
React Native - Layout and Styles - add some flex

basic flex usage - part I

```
...
export default class BasicFlexApp extends Component {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.col}>
          <Text>
            Welcome to Flex layouts!
          </Text>
          <Text>
            a few basic tests...
          </Text>
        </View>
        <View style={styles.col}>
          <Text>
            {instructions}
          </Text>
        </View>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'space-around',
    alignItems: 'center',
    backgroundColor: 'darkseagreen',
  },
  col: {
    flexDirection: 'column',
    backgroundColor: 'paleturquoise',
  },
});
```

Image - React Native - Flex Basics



[React Native Flex Basics](#)

Image - React Native - Flex Basics - List View

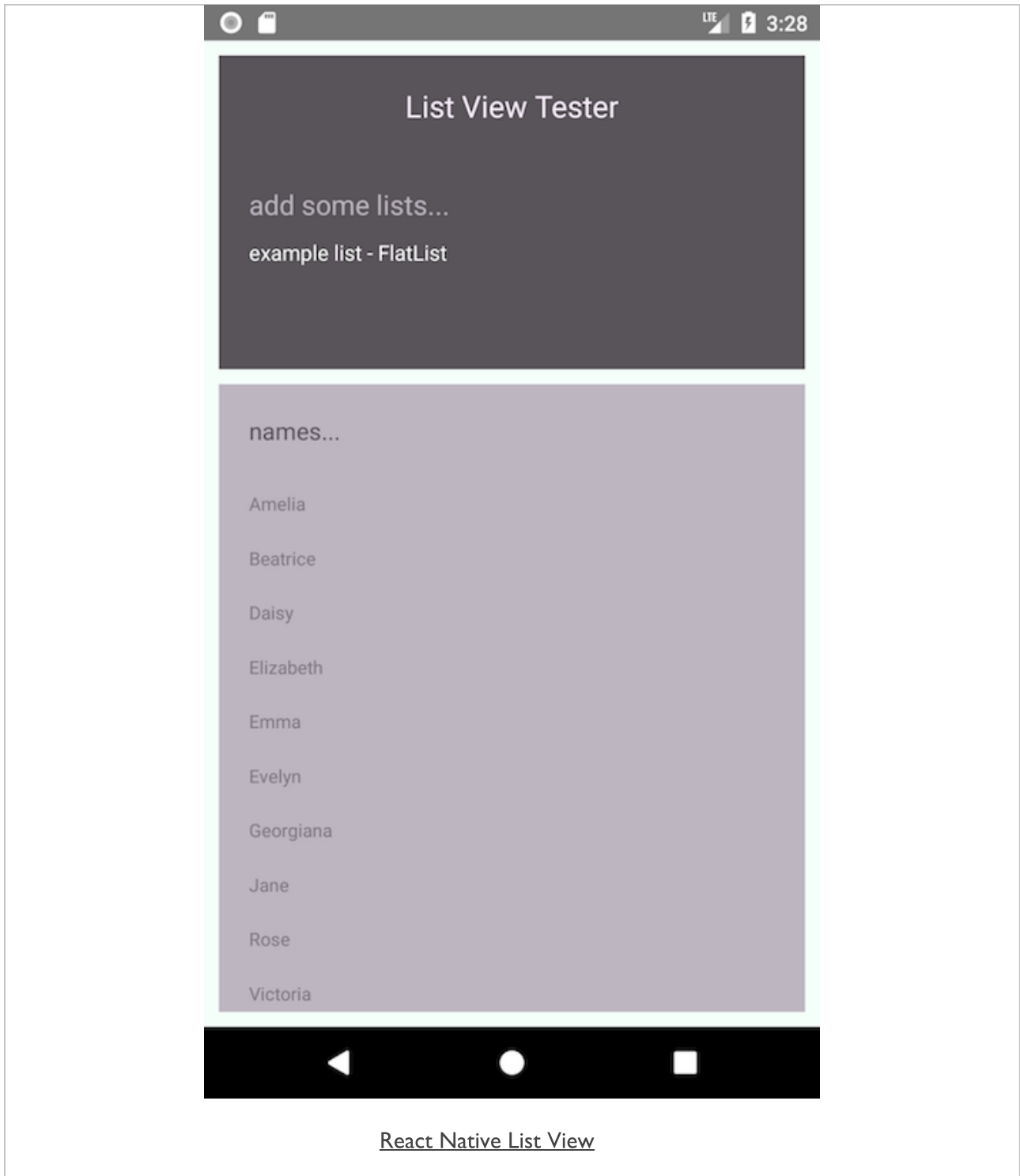


Image - React Native - Flex Basics - Scroll View

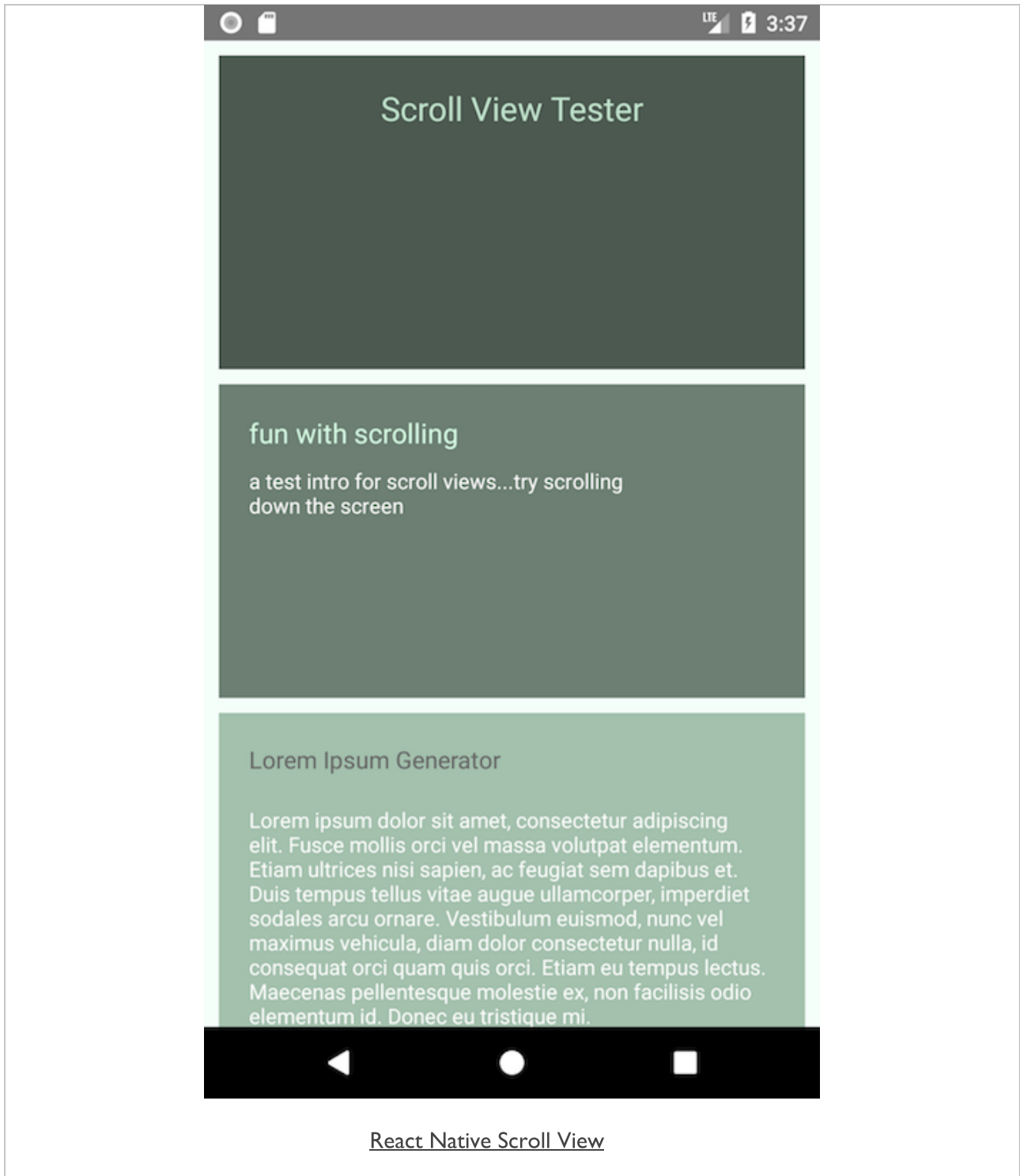


Image - React Native - Styles

text input

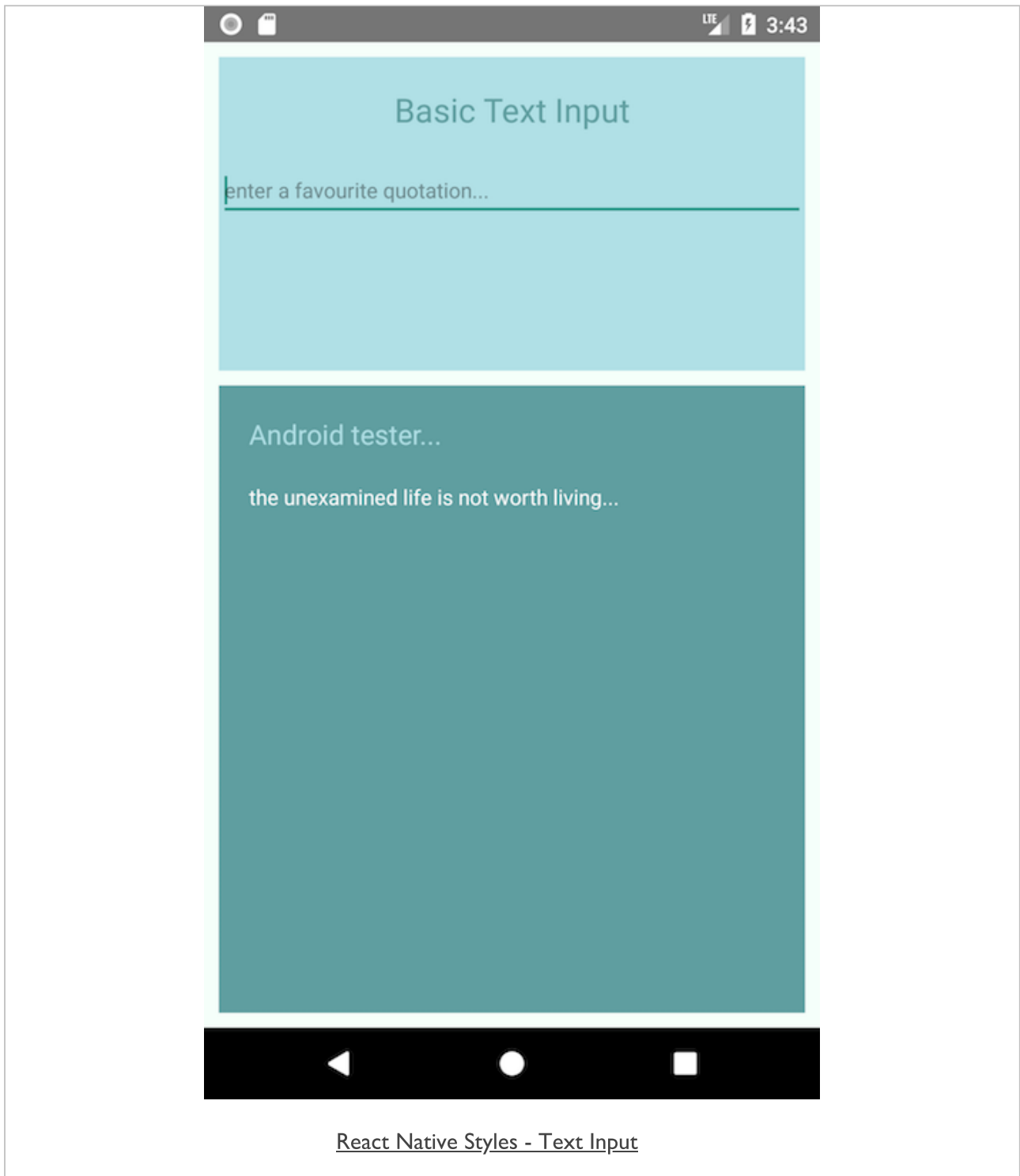
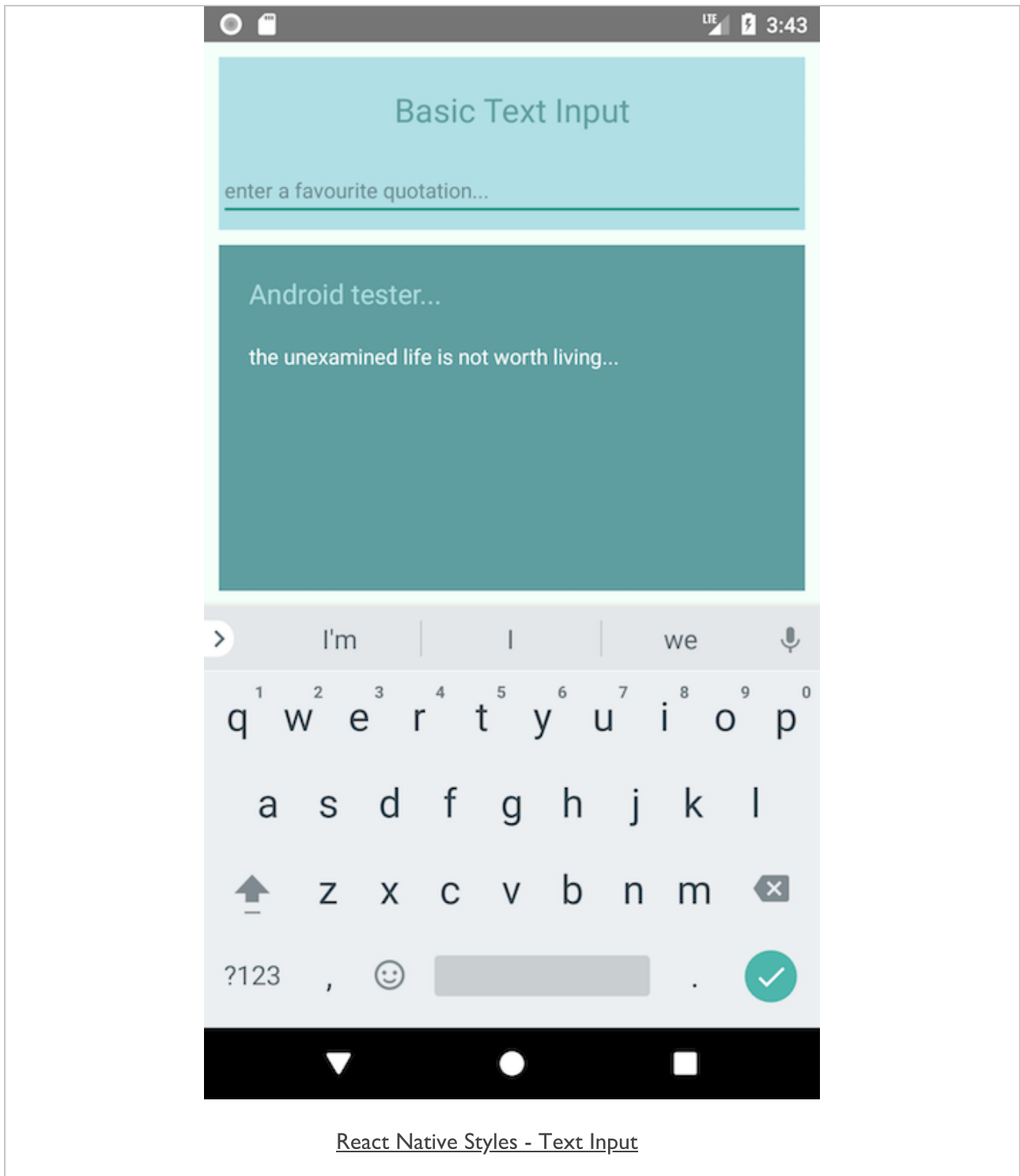


Image - React Native - Styles

text input with keyboard



React Native - Layout and Styles

basic styling

- similar to CSS usage with standard client-side apps
 - *styles are defined and set for colour, size, background colour...*
- property names for these styles specified using a camelCase pattern. e.g.

```
fontWeight  
fontSize  
backgroundColor
```

- styles may be set using a plain JavaScript variable
 - *acts as a container for multiple styles*
- using `StyleSheet.create()`
 - *we can pass an object defining multiple custom style properties*
 - *properties include name/value pairs*
 - *the value is set as an object with the defined styles, e.g.*

```
const styles = StyleSheet.create({  
  headermain: {  
    fontWeight: 'bold',  
    fontSize: 25,  
    color: 'green',  
  },  
});
```

React Native - Layout and Styles

style usage

- to add a style to a component
 - set value of the *style* prop to a standard JavaScript object, e.g.

```
<Text style={styles.headermain}>Main Header</Text>
```

- in this example,
 - simply using the property from the *styles* object
 - this will add the required *style* values for the defined prop

React Native - Layout and Styles

Platform specific styles

```
import { Platform, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    ...Platform.select({
      ios: {
        fontFamily: 'Arial',
        color: 'cadetblue',
      },
      android: {
        fontFamily: 'Roboto',
        color: 'green',
      },
    }),
    textAlign: 'center',
    margin: 10,
    fontSize: 20,
  },
});
```

React Native - Layout and Styles

Style inheritance - part I

- React Native documentation suggests a preferred pattern for setting parent styles
 - *styles may then be inherited for children*
- pattern uses nested components with a custom parent defined with abstracted styles
- child component may then inherit such styles
 - *or override with specific component-level styles*

```
class MyAppText extends Component {
  render() {
    return (
      <Text>
        {this.props.children}
      </Text>
    );
  }
}
```

- e.g. a parent component is created for an app's rendering of basic text
- this will simply return any child text as a default Text component
- we may also create custom styles to add to this new component

```
textdefault: {
  fontSize: 15,
  color: '#000'
}
```

React Native - Layout and Styles

Style inheritance - part 2

- usage may then be as follows,

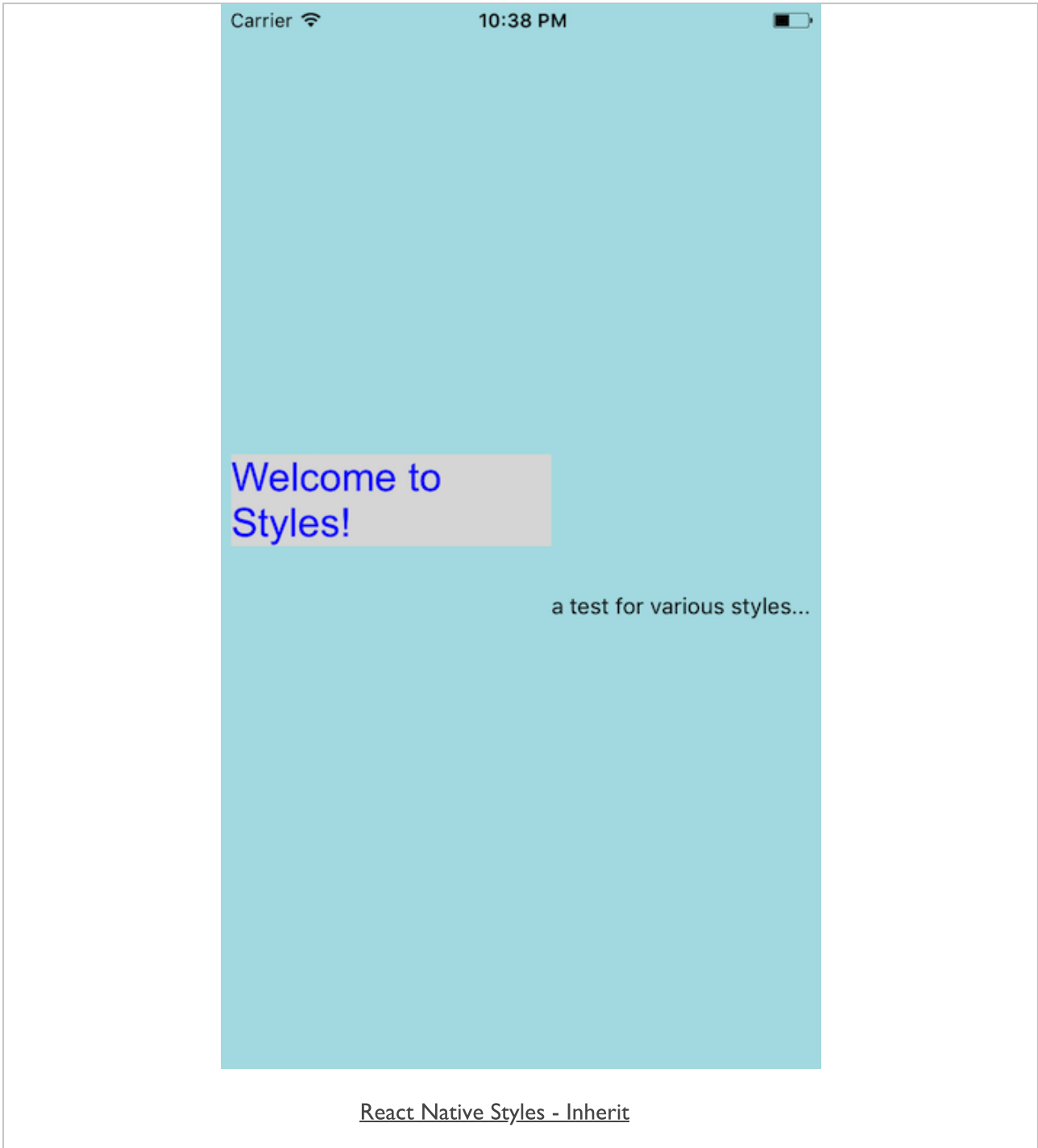
```
<MyAppText style={styles.textdefault}>
  some app text...
  <Text style={styles.welcome}>Welcome to Styles!</Text>
</MyAppText>
```

- the *child* text in the `MyAppText` component
 - initially styled with the `textdefault` styles
- we may then override or supplement these styles
 - e.g. with *specific styles on a given child component*

```
welcome: {
  ...Platform.select({
    ios: {
      fontFamily: 'Arial',
      color: 'blue',
    },
    android: {
      fontFamily: 'Roboto',
      color: 'green',
    },
  }),
  fontSize: 25,
  textAlign: 'auto',
  backgroundColor: '#ddd',
}
```

Image - React Native - Styles

basic styles



React Native Styles - Inherit

Image - React Native - Styles

basic buttons

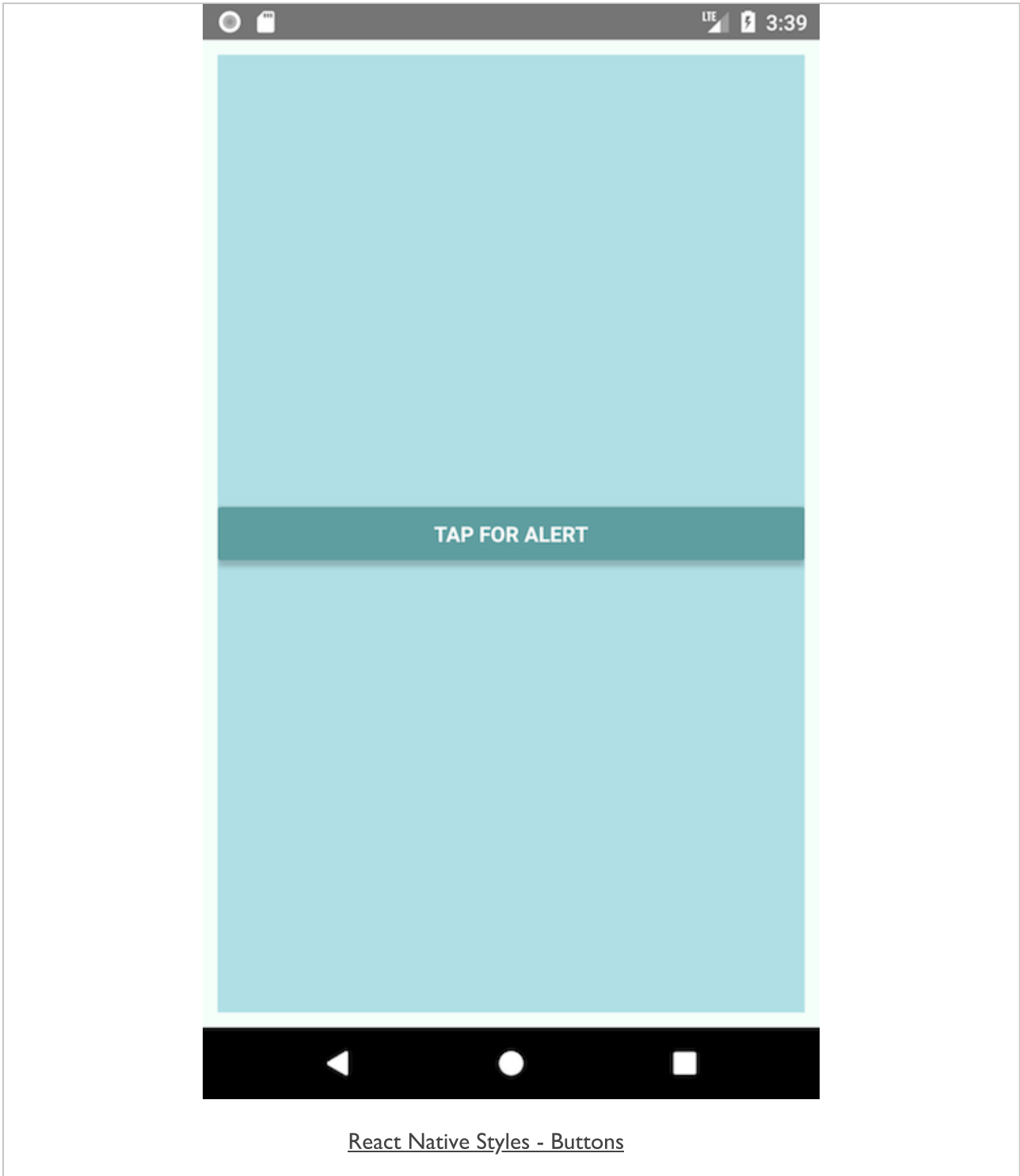


Image - React Native - Styles

basic touchable

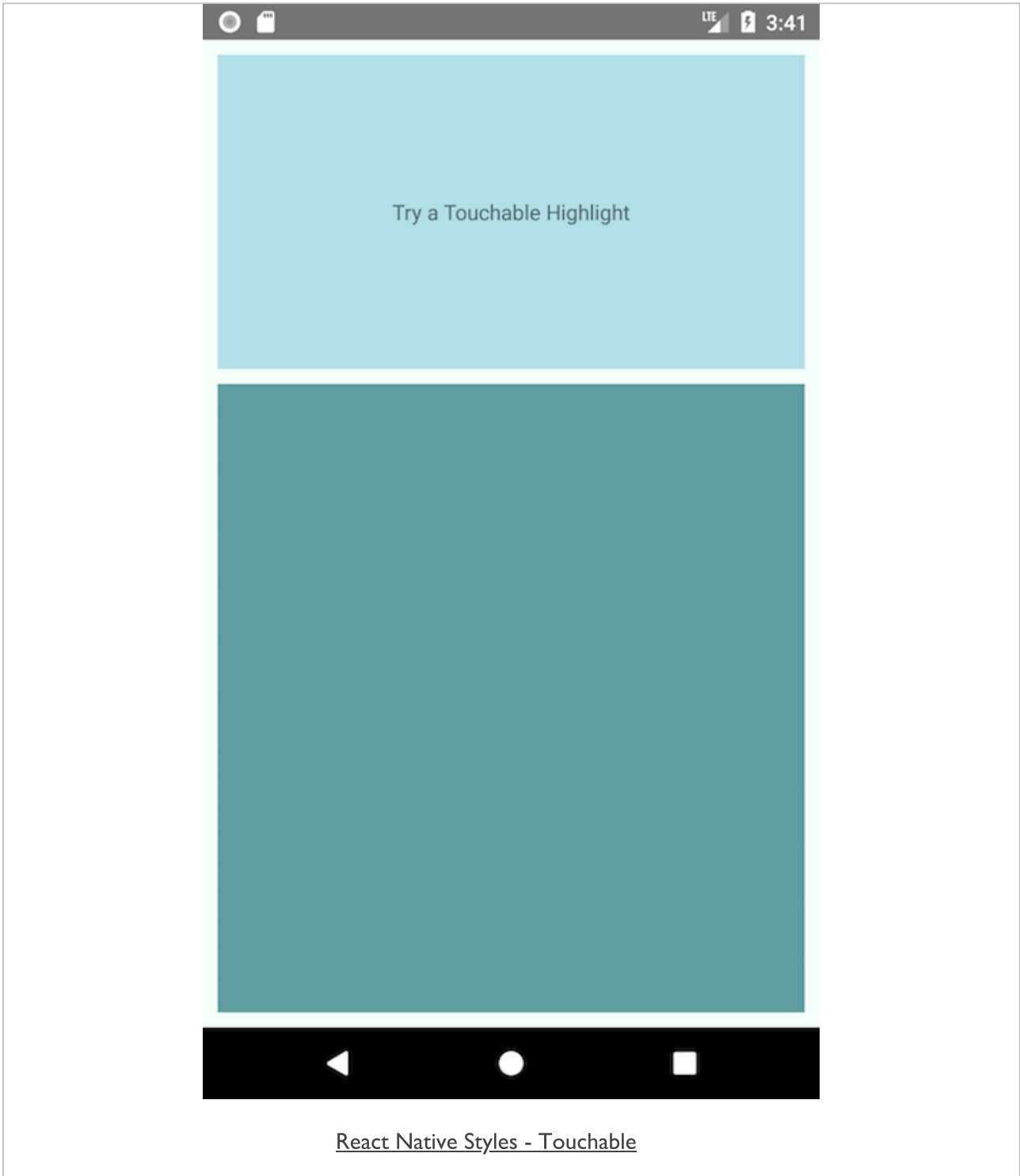
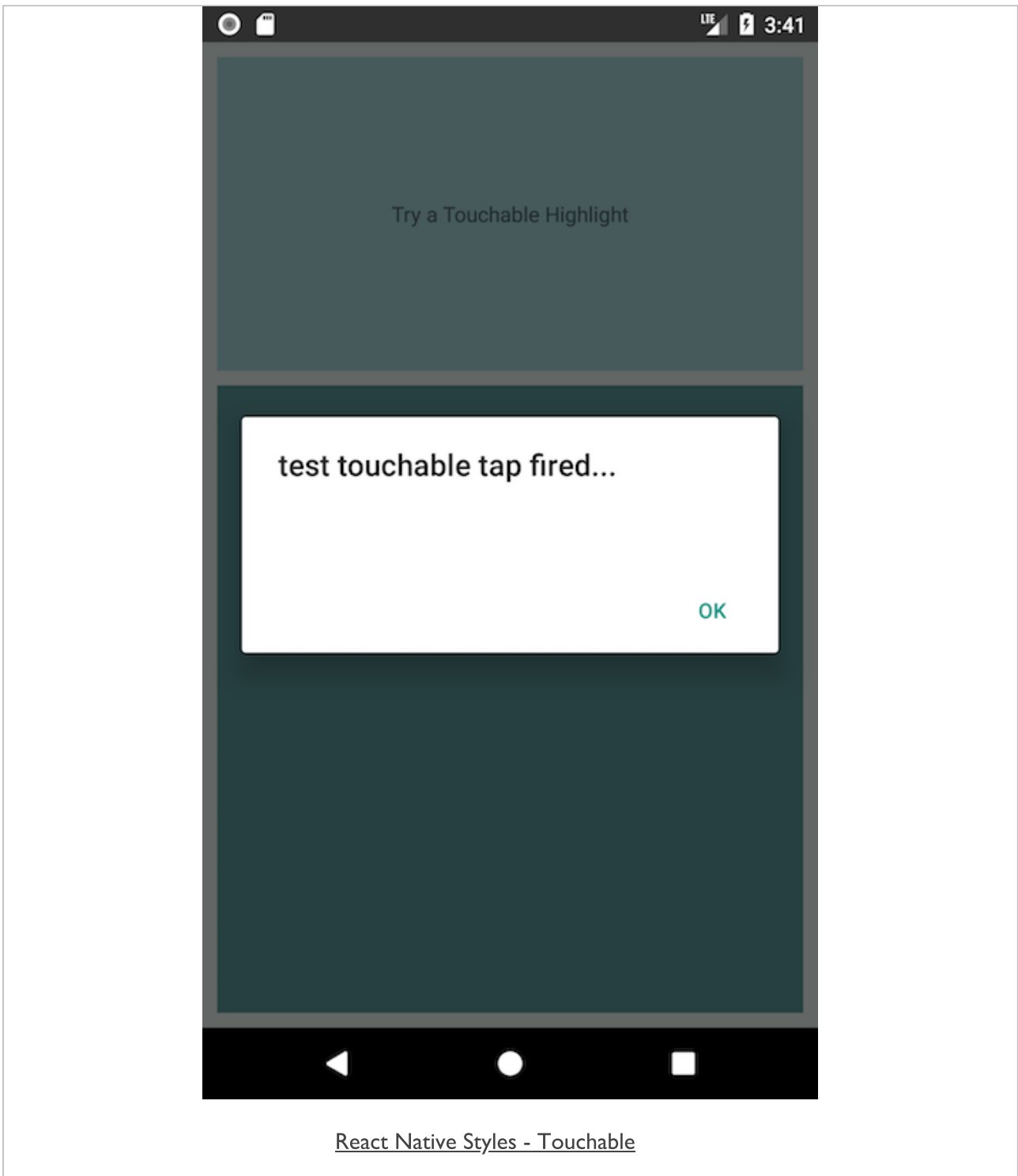


Image - React Native - Styles

basic touchable with alert



React Native - State

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

React Native - State

general usage

- 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 modify state within our app*
- state may be managed within a React Native app
 - *or by using containers such as Redux, MobX...*
- *Redux and MobX are predominantly used with React based apps*
 - *standalone libraries for state management*
- by introducing a container such as *Redux*
 - *circumvent direct management of state with `setState`*
 - *state updates rely upon Redux management.*

React Native - State

state usage - example

- basic example of state usage and maintenance
 - may set a static message using *props*
 - then update a notification using *state*

```
// 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>
    );
  }
}
```

React Native - State

state usage - example outline - part I

- define the required imports for React and React Native
 - *including existing components we need for this basic app*
 - *import `AppRegistry`, `Text`, `View` components*
- define our required custom components
 - *one abstracted for broader re-use*
 - *another for use in the current specific app*
- `Tape` class is an abstracted component
 - *used for rendering passed text with a timer*
 - *constructor instantiates an object with passed props*
 - *e.g. passed text for rendering*
- in the `Tape` class constructor
 - *`super` is used to call parent class' constructor with props provided*
 - *i.e. uses `Component` to setup props*
- then set the initial state on the instantiated object
 - *default to `true` for this component*

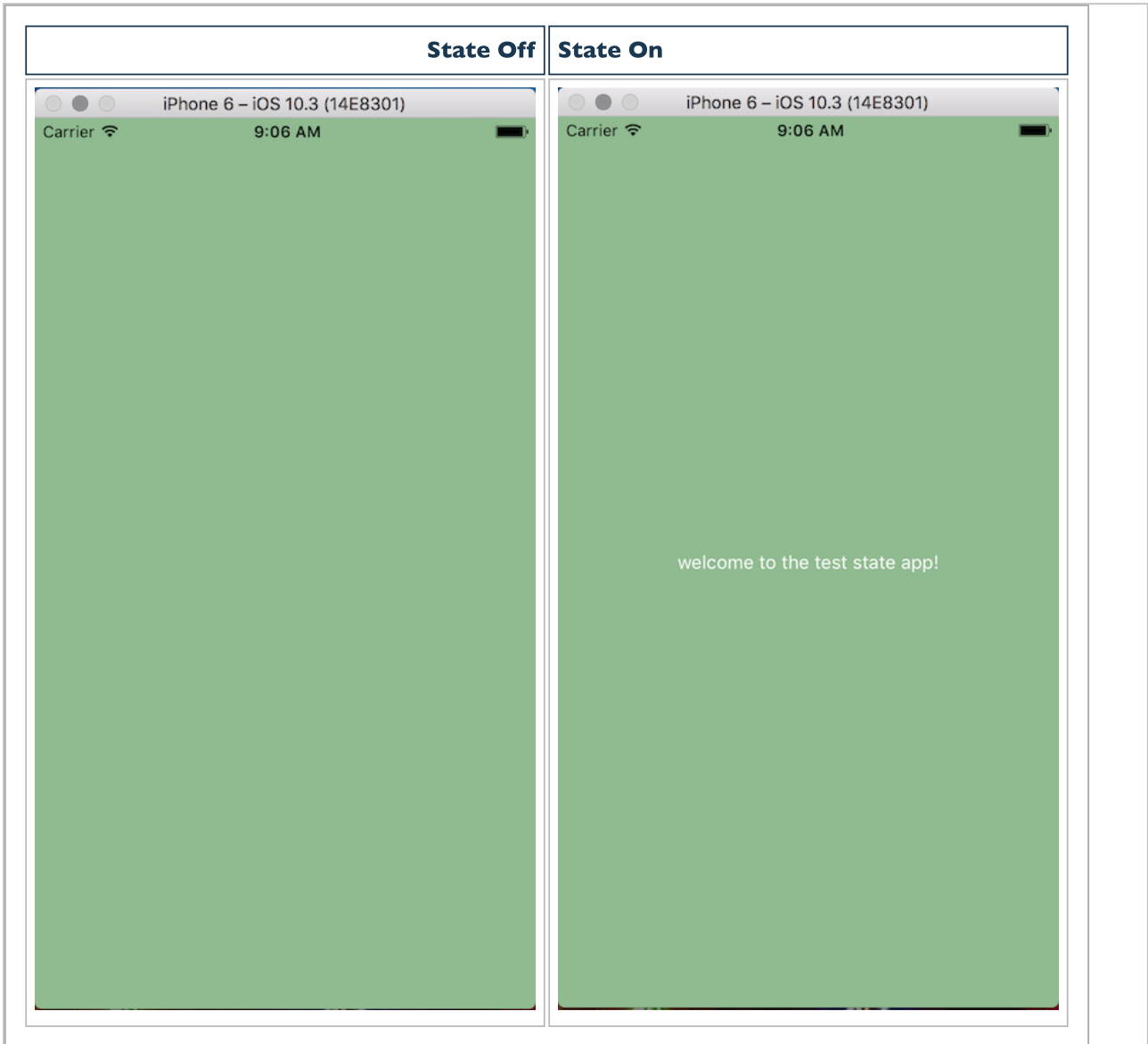
React Native - State

state usage - example outline - part 2

- call the JS function `setInterval ()` to create a basic timer
 - *creates the simple UI animation - delay is set to 1500 milliseconds*
- main focus of this function is to modify state
 - *this may trigger an update*
- call `setState` on the current object
 - *function is called with a passed `updater` and a `callback`*
- `prevState` is available for the `setState` function
 - *used to set state based on previous known state*
- state itself may not necessarily be triggered immediately
 - *React may delay an update until it has a worthwhile queue*
- we can call an immediate callback as this `setState` is registered
- we simply change the boolean value for `showText`
 - *e.g. `false` to `true`, `true` to `false`*
- then call the `render ()` function on the current object
 - *outputting text passed using `props`*
- simply check the boolean value in state
 - *then render a `text` component with `props` text or a blank space*

Image - React Native - State

basic usage



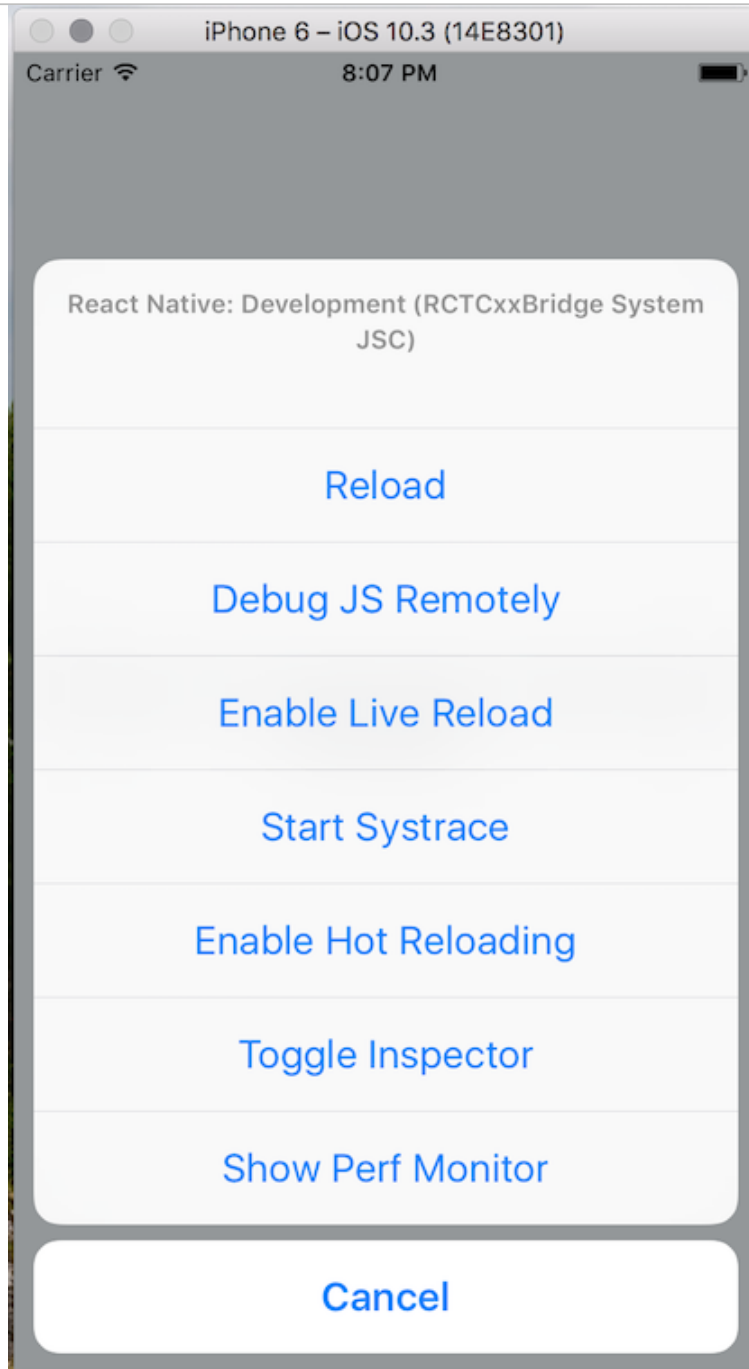
React Native - Debugging an app

Chrome DevTools

- debugging mobile may become problematic, time consuming...
- React Native's *JavaScript* event loop
 - *may be connected to Chrome's DevTools*
 - *DevTools is a quick and useful debugging option*
- use key combinations to show dev menu in simulator
 - *Windows 10 = Ctrl+D*
 - *OS X = Cmd+D*
- various options for testing &c.

Image - React Native - Chrome DevTools

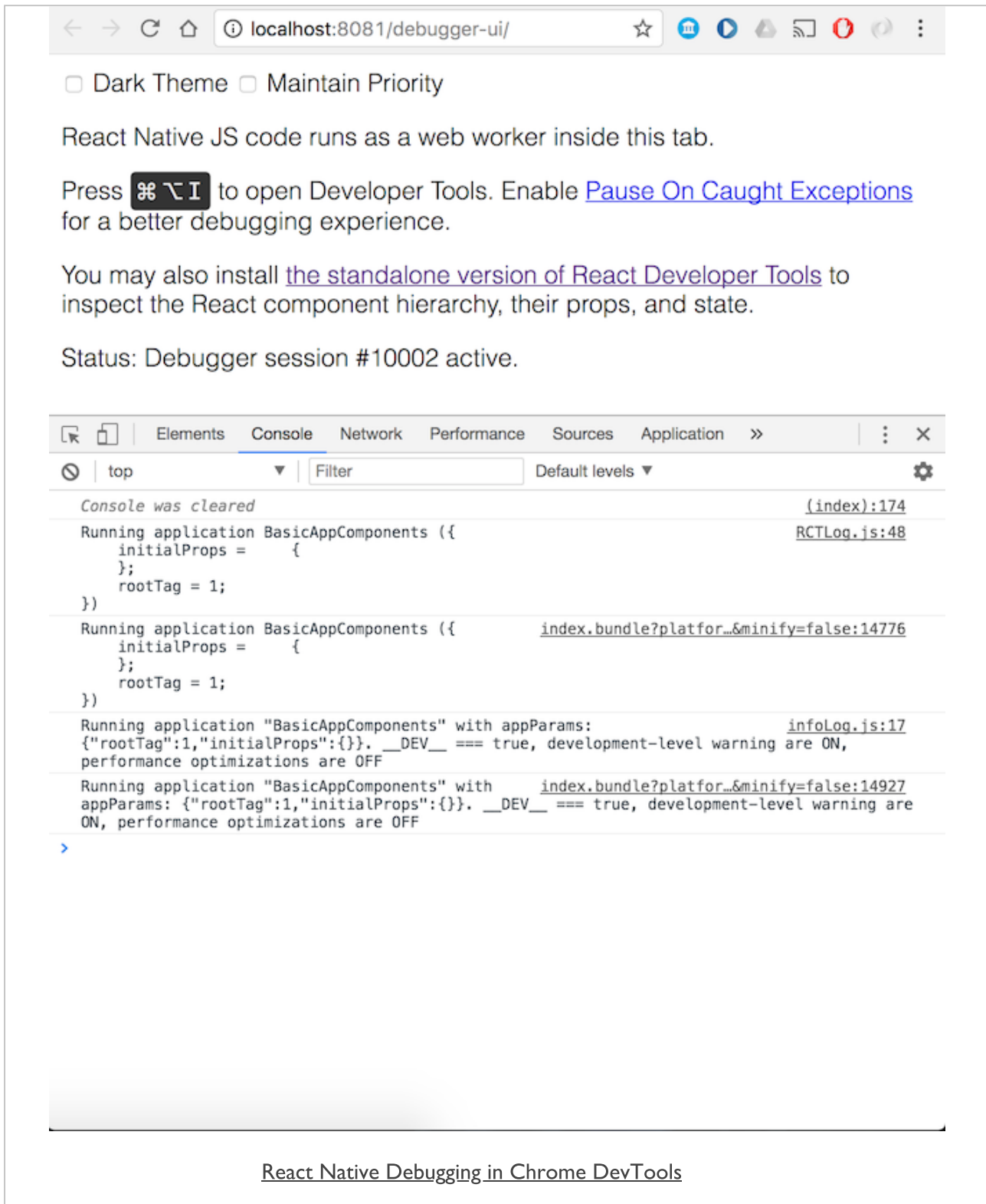
iOS simulator options



[react Native Options in iOS Simulator](#)

Image - React Native - Chrome DevTools

developer tools



Dark Theme Maintain Priority

React Native JS code runs as a web worker inside this tab.

Press **⌘ I** to open Developer Tools. Enable [Pause On Caught Exceptions](#) for a better debugging experience.

You may also install [the standalone version of React Developer Tools](#) to inspect the React component hierarchy, their props, and state.

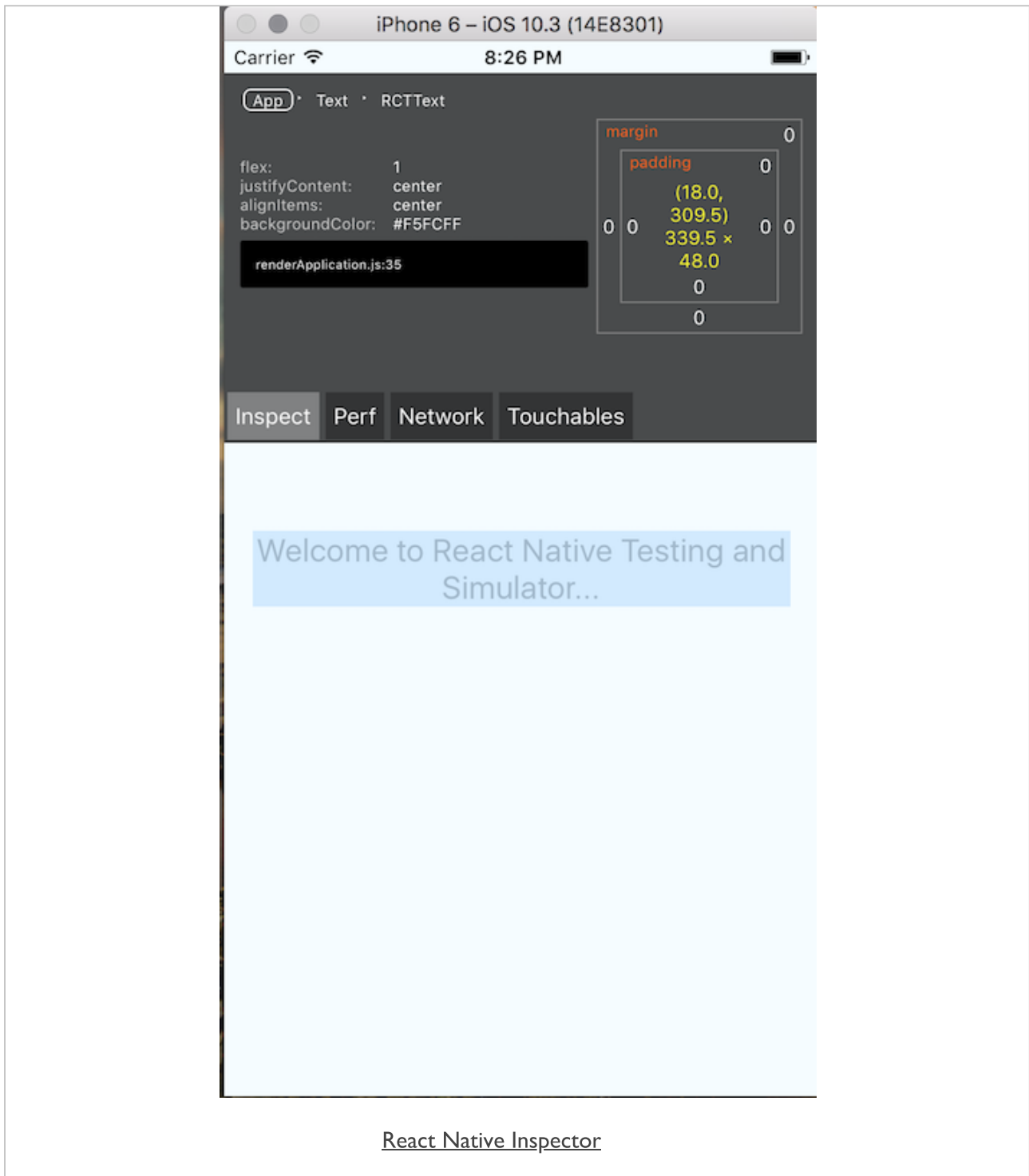
Status: Debugger session #10002 active.

```
Console was cleared (index):174
Running application BasicAppComponents ({
  initialProps =  {
  };
  rootTag = 1;
}) RCTLog.js:48
Running application BasicAppComponents ({
  initialProps =  {
  };
  rootTag = 1;
}) index.bundle?platfor...&minify=false:14776
Running application "BasicAppComponents" with appParams:
{"rootTag":1,"initialProps":{}}. __DEV__ === true, development-level warning are ON,
performance optimizations are OFF infoLog.js:17
Running application "BasicAppComponents" with
appParams: {"rootTag":1,"initialProps":{}}. __DEV__ === true, development-level warning are
ON, performance optimizations are OFF index.bundle?platfor...&minify=false:14927
```

[React Native Debugging in Chrome DevTools](#)

Image - React Native - Debug Options

inspector



React JavaScript Library

a React approach to development - part I

- **unidirectional data flow**
 - *a key concept that React introduced for UI development*
- the UI of an application is now a function of the state of the application
- instead of the need to update the UI directly we can now modify **state**
 - *unlike tradition UI development*
- e.g. in JavaScript we add an eventListener to an element
 - *check for user interaction &c.*
 - *update the UI directly*
- with React we record the event in the UI
 - *then update the state of the component*
- React will then propagate this change to the UI
- it's the change in state that causes components to be updated

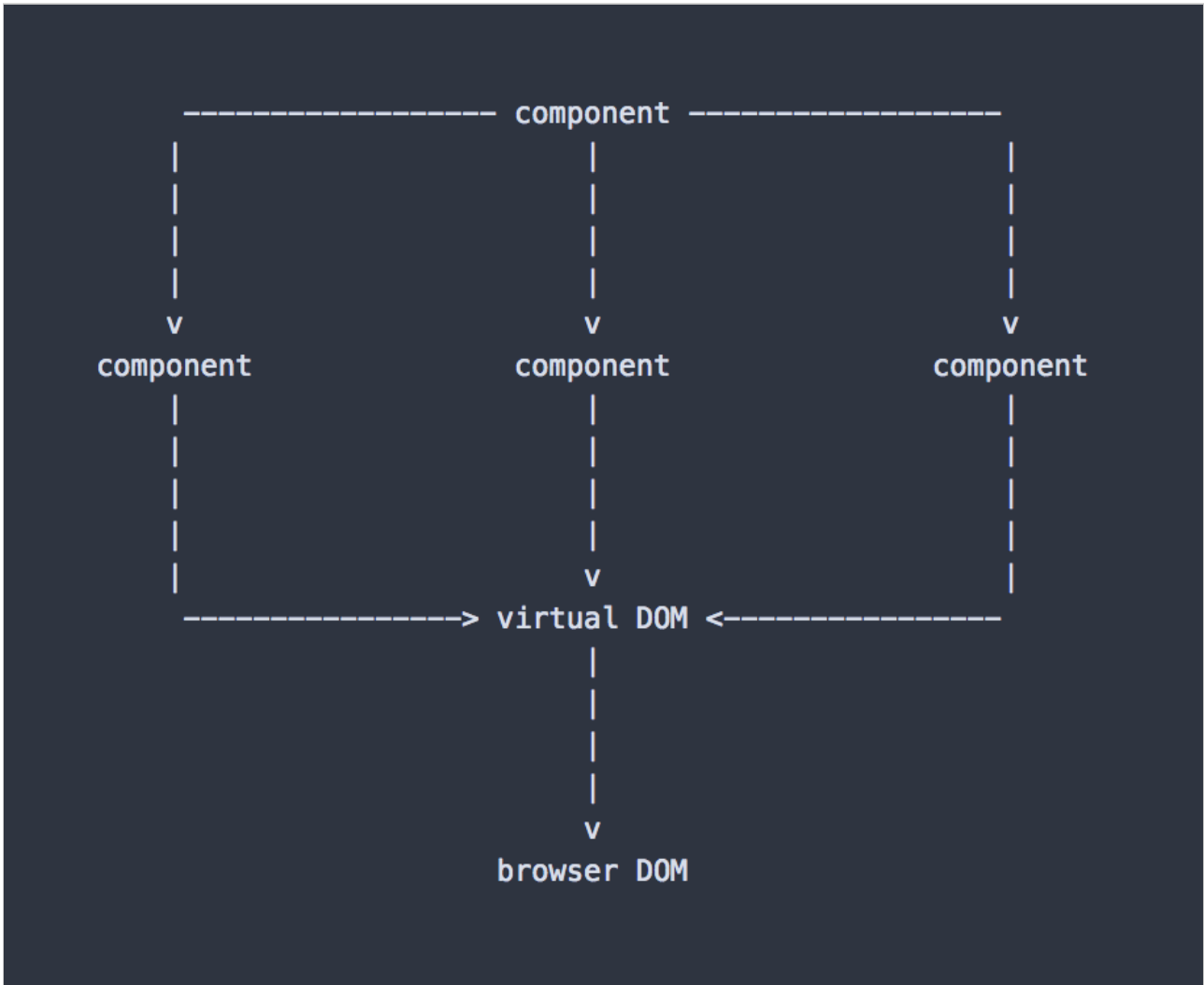
React JavaScript Library

a React approach to development - part 2

- components play a crucial role in React development
- dividing the logic and structure of our UI into **reusable** components
- inherently easier to test and reuse a given component across an application
- **DRY**, or **Don't Repeat Yourself**
 - *becomes key for how we conceive and use components*
- React components also inherently create a declarative pattern and structure
 - *helps with development of these apps*
- useful feedback for the layout and development of an app
 - *tree-like data structure of component usage*
- code inherently becomes easier to read...

React JavaScript Library

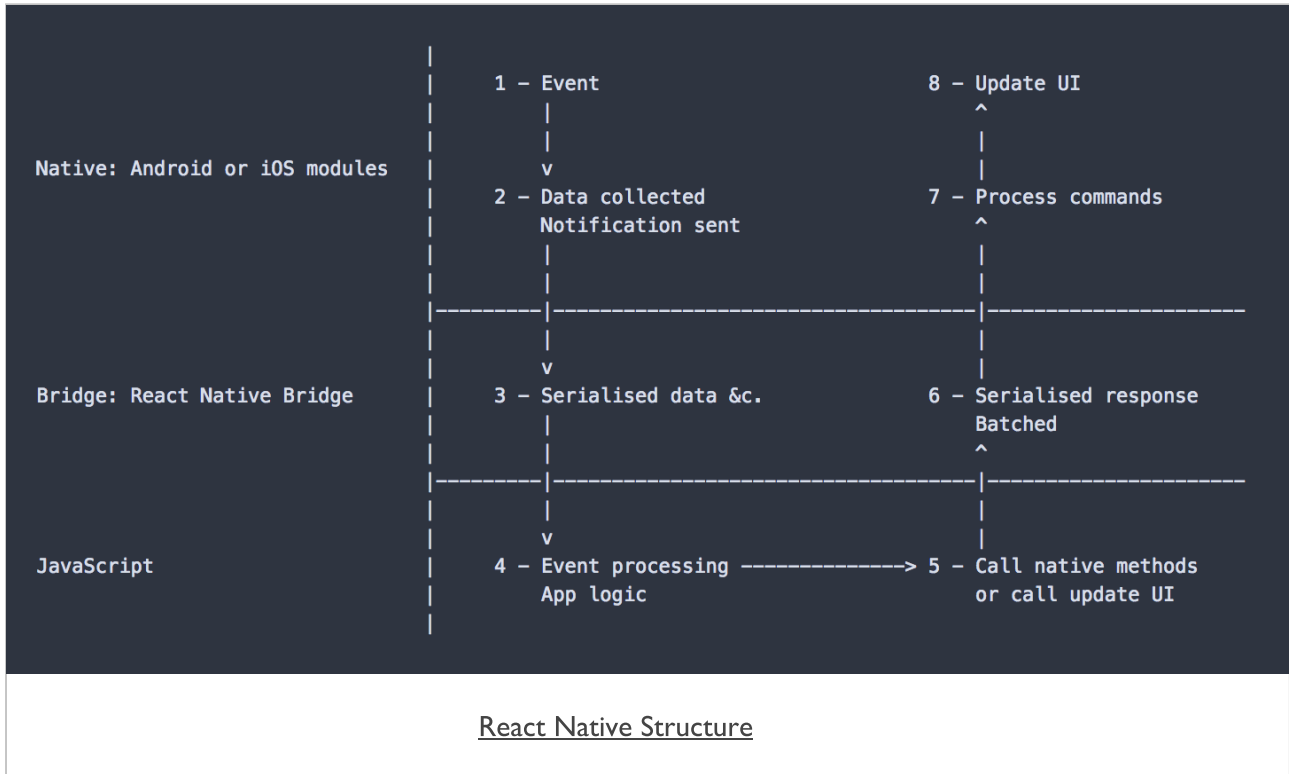
data flow



React Data Flow

Image - React Native - Structure

structural considerations



React Native - Native APIs and Threading

structural considerations

- a separate Native modules thread
 - *used to access and process Native API requests...*
- e.g. access a device's camera, photos, geolocation, gestures...
- JavaScript layer also has a runtime thread
 - *a JavaScript event loop*
- complex calculations can become expensive in the JavaScript layer
- many, consistent UI updates will also become expensive and drag on performance

React JavaScript Library

getting started - part 1

- many different options for using React
- create a new app using React
 - e.g. *Create React App - GitHub*
- add React to an existing app
 - e.g. *using NPM to install React and dependencies*

```
npm init  
npm install --save react react-dom
```

- import React into a project using the standard Node `import` options, e.g.

```
import React from 'react';  
import ReactDOM from 'react-dom';
```

React JavaScript Library

getting started - part 2

- for earlier versions of React and JSX
 - *pre-compile JSX into JavaScript before deploying our application*
 - *used React's JSXTransformer option to compile and monitor JSX for dev projects*
- as React has evolved over the last year
 - *still use this underlying concept*
 - *Babel in-browser JSX transformer for explicit ES6 support (if required...)*
- Babel will add a check to our app to allow us to use JSX syntax
 - *React code then understood by the browser*
- dynamic transformation works well for most test scenarios
 - *preferable to pre-compile for production apps*
 - *should help to make an app faster for production usage*

React JavaScript Library

JSX - intro

- JSX stands for **JavaScript XML**
 - *follows an XML familiar syntax for developing markup within React components*
- JSX is not compulsory within React
 - *might be omitted due to compile requirements for an app*
- JSX may be useful for an app
 - *it makes components easier to read and understand*
 - *its structure is more succinct and less verbose*
- A few defining characteristics of JSX
 - *each JSX node maps to a function in JavaScript*
 - *JSX does not require a runtime library*
 - *JSX does not supplement or modify the underlying semantics of JavaScript*

React Native

JSX intro and usage

- Facebook considers JSX as a XML-like extension to ECMAScript
 - *without any defined semantics*
 - *NOT intended to be implemented by engines or browsers*
 - *not a proposal to incorporate JSX into the ECMAScript spec itself*
 - *used to transform syntax into standard ECMAScript*
- for React Native
 - *these JavaScript objects are passed to the React Native Bridge*
 - *then translated into native components.*
- e.g. a standard `<Text>` component in JSX may be written as follows

```
<Text style={styles.description}>  
  A test React Native app...  
</Text>
```

- JSX will then be transpiled by the React Native bridge into the following JavaScript

```
React.createElement(  
  Text,  
  { style: styles.welcome },  
  "A test React Native app..."  
);
```

React Native

JSX hierarchies

- benefit of JSX with React Native is its use with hierarchies
 - such as a standard `<View>` and nested `<Text>` component structure

```
<View style={styles.container}>
  <Text style={styles.description}>
    A test React Native app...
  </Text>
</View>
```

- transpiled into the following JavaScript

```
React.createElement(
  View,
  null,
  React.createElement(
    Text,
    { style: styles.welcome },
    "A test React Native app..."
  )
);
```

React Native

JSX children

- a primary feature of JSX with React Native
 - *option to pass children to a React component*
 - *enables effective component composition*
- seen regularly with hierarchy composition
 - e.g. hierarchy of `<View>` and `<Text>`

```
<View style={styles.container}>
  <Text style={styles.description}>
    A test React Native app...
  </Text>
</View>
```

- we may create a simple component and encapsulate this structure

```
class Container extends Component {
  render() {
    return (
      <View style={styles.container}>{ this.props.children }</View>
    )
  }
}
```

- then reuse this component as necessary

```
<Container>
  <Text style={styles.description}>
    A test React Native app...
  </Text>
</Container>
```

React Native

JSX props and children

- seen example usage of props with styles, data, and now *children*
- as we pass a standard prop, such as `style`
 - *passing a property to the defined React component*
- property is accessible inside this component using the standard syntax

```
this.props.propName
```

- as we define a component
 - *children is default prop React passes to this component for the hierarchy*
 - *becomes the component reference for any children in this hierarchy*

```
this.props.children
```

React JavaScript Library

JSX - benefits

- why use JSX, in particular when it simply maps to JavaScript functions?
- many of the inherent benefits of JSX become more apparent
 - *as an application, and its code base, grows and becomes more complex*
- benefits can include
 - *a sense of familiarity - easier with experience of XML and DOM manipulation*
 - *eg: React components capture all possible representations of the DOM*
 - *JSX transforms an application's JavaScript code into semantic, meaningful markup*
 - *permits declaration of component structure and information flow using a similar syntax to HTML*
 - *permits use of pre-defined HTML5 tag names and custom components*
 - *easy to visualise code and components*
 - *considered easier to understand and debug*
 - *ease of abstraction due to JSX transpiler*
 - *abstracts process of converting markup to JavaScript*
 - *unity of concerns*
 - *no need for separation of view and templates*
 - *React encourages discrete component for each concern within an application*
 - *encapsulates the logic and markup in one definition*

React JavaScript Library

JSX - composite components

- example React component might allow us to output a custom heading

```
class OutputHeading extends Component {
  render() {
    return (
      // render passed props `output` value, pass heading size...
      <Text style={this.props.style}>{this.props.output}</Text>
    );
  }
}
```

- currently return a standard Text component
- now update this example to work with dynamic values
- JSX considers values dynamic if they are placed between curly brackets { . . }
 - *treated as JavaScript context*

```
<OutputHeading output='Component Heading Tester' style={styles.heading3} />
```

React JavaScript Library

JSX - conditionals

- a component's markup and its logic are inherently linked in React
- this naturally includes *conditionals*, *loops*...
- adding `if` statements directly to JSX will create invalid JavaScript
 1. ternary operator

```
...  
this.state.isComplete ? 'is-complete' : ''  
...
```

2. variable

```
getIsComplete: function() {  
  return this.state.isComplete ? 'is-complete' : '';  
},  
render() {  
  var isComplete = this.getIsComplete();  
  return (  
    <Test complete={isComplete}>...</Test>  
  );  
}
```

3. function call

```
getIsComplete: function() {  
  return this.state.isComplete ? 'is-complete' : '';  
},  
render() {  
  return (  
    <Test complete={this.getIsComplete()}>...</Test>  
  );  
}
```

- to handle React's lack of output for *null* or *false* values
 - use a *boolean value* and follow it with the desired output

React JavaScript Library

JSX - special considerations for attributes - part I

- in JSX, there are special considerations for attribute
 - *key*
 - *ref*
- l. *key*
- an optional unique identifier that remains consistent throughout render passes
- informs React so it can more efficiently select when to reuse or destroy a component
- helps improve the rendering performance of the application.
- eg: if two elements already in the DOM/View need to switch position
 - *React is able to match the keys and move them*
 - *does not require unnecessary re-rendering of the complete DOM/View*

React JavaScript Library

JSX - special considerations for attributes - part 2

2. `ref`

- `ref` permits parent components to easily maintain a reference to child components
 - *available outside of the render function*
- to use `ref`, simply set the attribute to the desired reference name

```
render() {  
  return (  
    <TextInput ref='myInput' ... />  
  );  
}
```

- able to access this `ref` using the defined `this.refs.myInput`
 - *access anywhere in the component*
 - *object accessed through this `ref` known as a backing instance*
- **NB:** not the actual DOM/View
 - *a description of the component React uses to create the view when necessary*

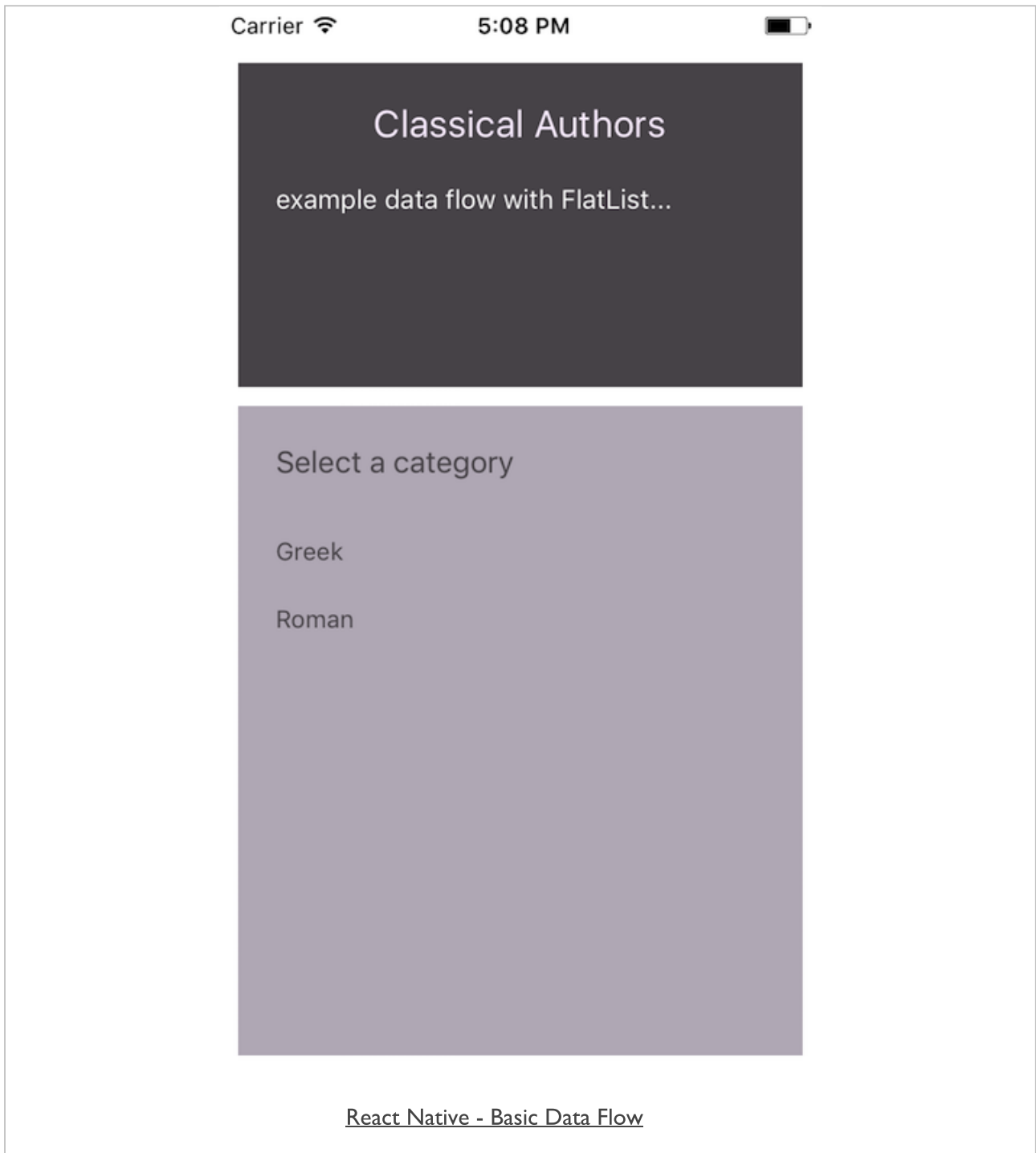
React JavaScript Library

data flow

- data flows in one direction in React
 - *namely from **parent to child***
- helps to make components nice and simple, and predictable as well
- components take *props* from the parent, and then render
- if a *prop* has been changed, for whatever reason
 - *React will update the component tree for that change*
 - *then re-render any components that used that property*
- Internal state also exists for each component
 - *state should only be updated within the component itself*
- we can think of data flow in React
 - *in terms of *props* and *state**

Image - React Native - Data Flow

basic data flow with FlatList



React Native - Data Flow

basic data flow with FlatList - example

```
// custom abstracted component - expects props for list data...
class ListClassics extends Component {
  render() {
    return (
      <FlatList
        data={this.props.data}
        renderItem={({item}) => <Text style={styles.listItem}>{item.key}</Text>}
      />
    );
  }
}

// default component - use View container, render list &c. with passed props...
export default class DataFlow extends Component {
  render() {
    let classics = [{ key: 'Greek'}, {key: 'Roman'}];
    return (
      <View style={styles.container}>
        <View style={styles.headingBox}>
          <Text style={styles.heading1}>
            {intro.heading}
          </Text>
          <Text style={styles.content}>
            {intro.description}
          </Text>
        </View>
        <View style={styles.listBox}>
          <ListClassics data={classics} />
        </View>
      </View>
    );
  }
}
```

React JavaScript Library

data flow - props - part 1

- props can hold any data and are passed to a component for usage
- set props on a component during instantiation

```
let classics = [{ key: 'Greek'}, {key: 'Roman'}];  
<ListClassics classics={classics}/>
```

- also use the setProps method on a given instance of a component

```
var ListClassics = React.createClass({  
  render: function() {  
    return (  
      <li className="classic">{this.props.classics}</li>  
    );  
  }  
});  
  
var classics = [{ key: 'Greek'}];  
var listClassics = React.render (  
  <ListClassics/>,  
  document.getElementById('example')  
);  
  
listClassics.setProps({ classics: classics });
```

React Native

data flow - setNativeProps

- React Native has a similar option called `setNativeProps`
- React.js may directly manipulate a DOM node
- likewise, we may need to directly modify or manipulate a mobile app
- React Native documentation recommend such usage as follows,

Use setNativeProps when frequent re-rendering creates a performance bottleneck

- not recommended for frequent use
- we may need to use it for
 - *regular animation updates*
 - *form management*
 - *graphics...*
- use with care

React Native

data flow - setNativeProps example

- define function for clearTextInput

```
clearTextInput = () => {  
  this._textInput.setNativeProps({text: ''});  
}
```

- call clearTextInput () function on touch press

```
<Button  
  onPress={this.clearTextInput}  
  title='Tap to clear text'  
  color='#585459'  
>
```

- add TextInput component and define reference

```
<TextInput  
  //arrow function call to set value to current component...  
  ref={component => this._textInput = component}  
  style={styles.textInput}  
  placeholder={this.state.quoteInput}  
  onChangeText={(quoteText) => this.setState({quoteText})}  
  selectionColor='#585459'  
>
```


Image - React Native - Data Flow

setNativeProps example - default

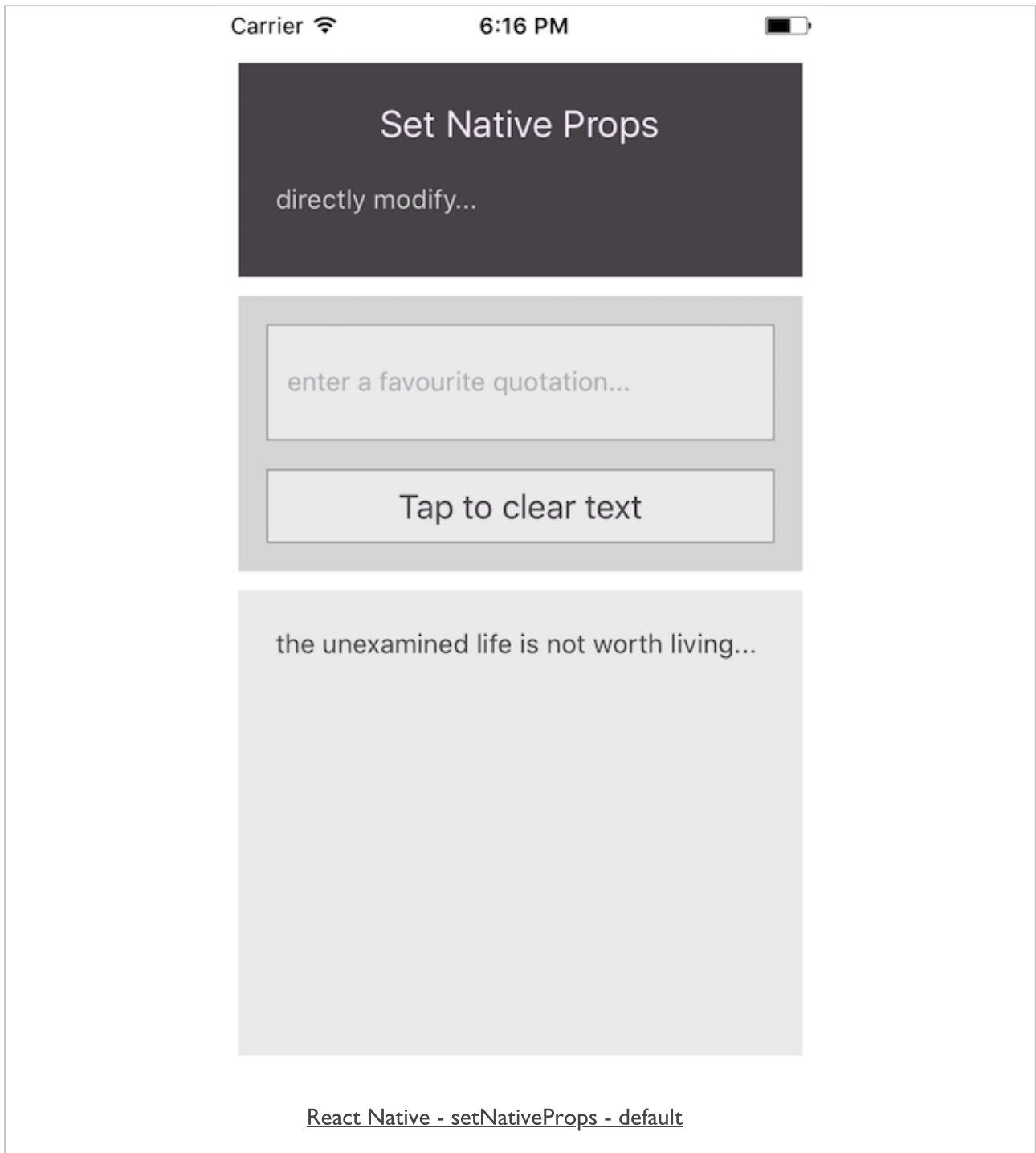


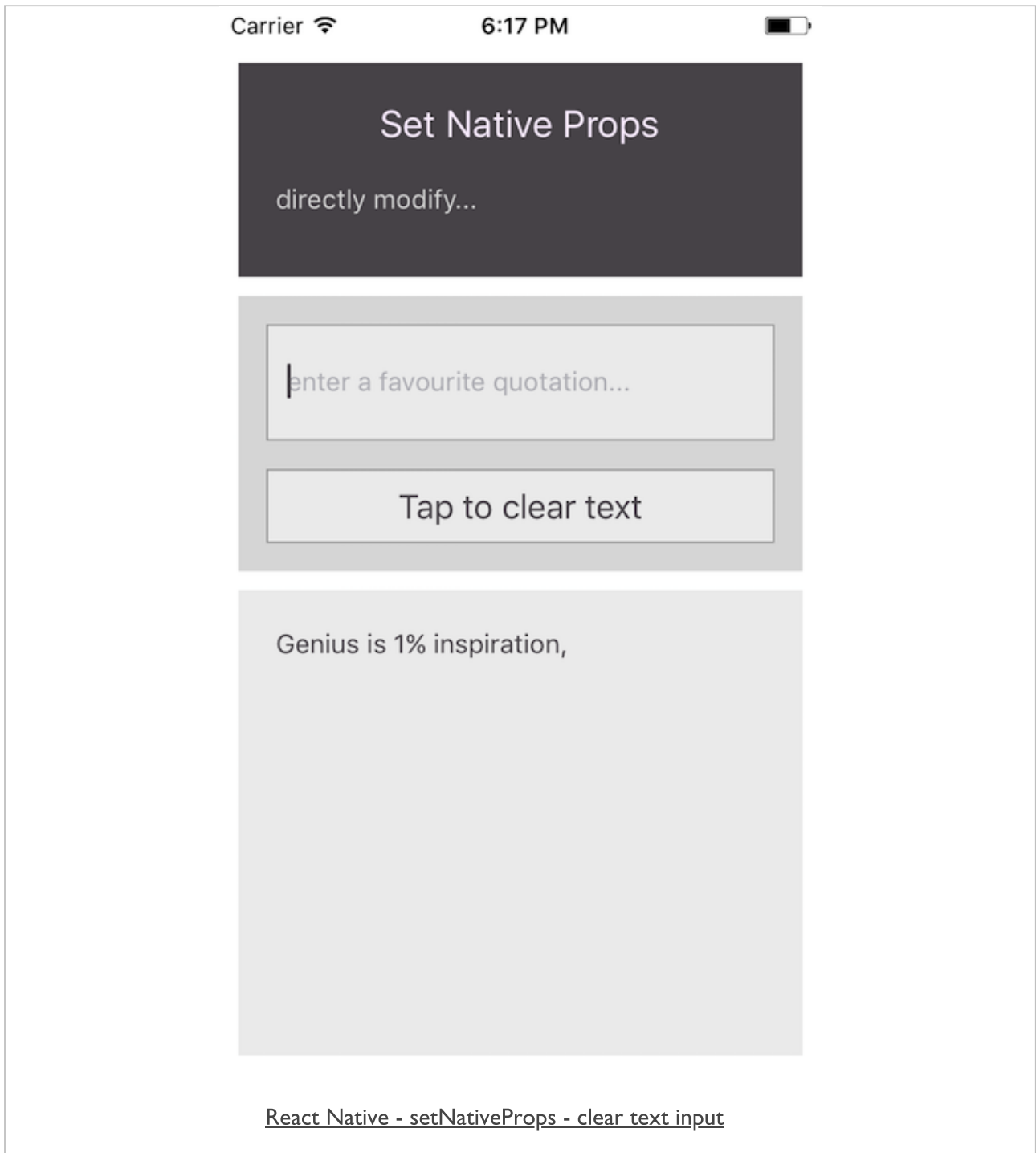
Image - React Native - Data Flow

setNativeProps example - add quote



Image - React Native - Data Flow

setNativeProps example - clear text input



Extra notes - Cordova, JS, React Native...

lots of extra notes on course website and GitHub Notes repository,

- <https://github.com/csteach422/notes/>

including,

- Cordova
- CSS
- Data store - incl. MongoDB and updates with Firebase
- design
- HTML & HTML5
- JS - incl. intro, core, logic, async...
- JS patterns
- React Native
- various - incl. Git & Heroku, Heroku & MongoDB

plus more notes will be added on Cordova, React Native, data stores, API usage...

...& see source code examples in the course's **source** repository on GitHub,

- <https://github.com/csteach422/source/>

References

- Cordova API docs
 - *config.xml*
 - *Globalization*
 - *Hooks*
 - *Merges*
 - *Network Information*
- React Native
 - *MDN - super*
 - *React JS - Component Lifecycle*
 - *React JS - componentDidUpdate*
 - *React JS - shouldComponentUpdate*
 - *React Native - Layout Props*