

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

---

Fall Semester 2018 - 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 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

---

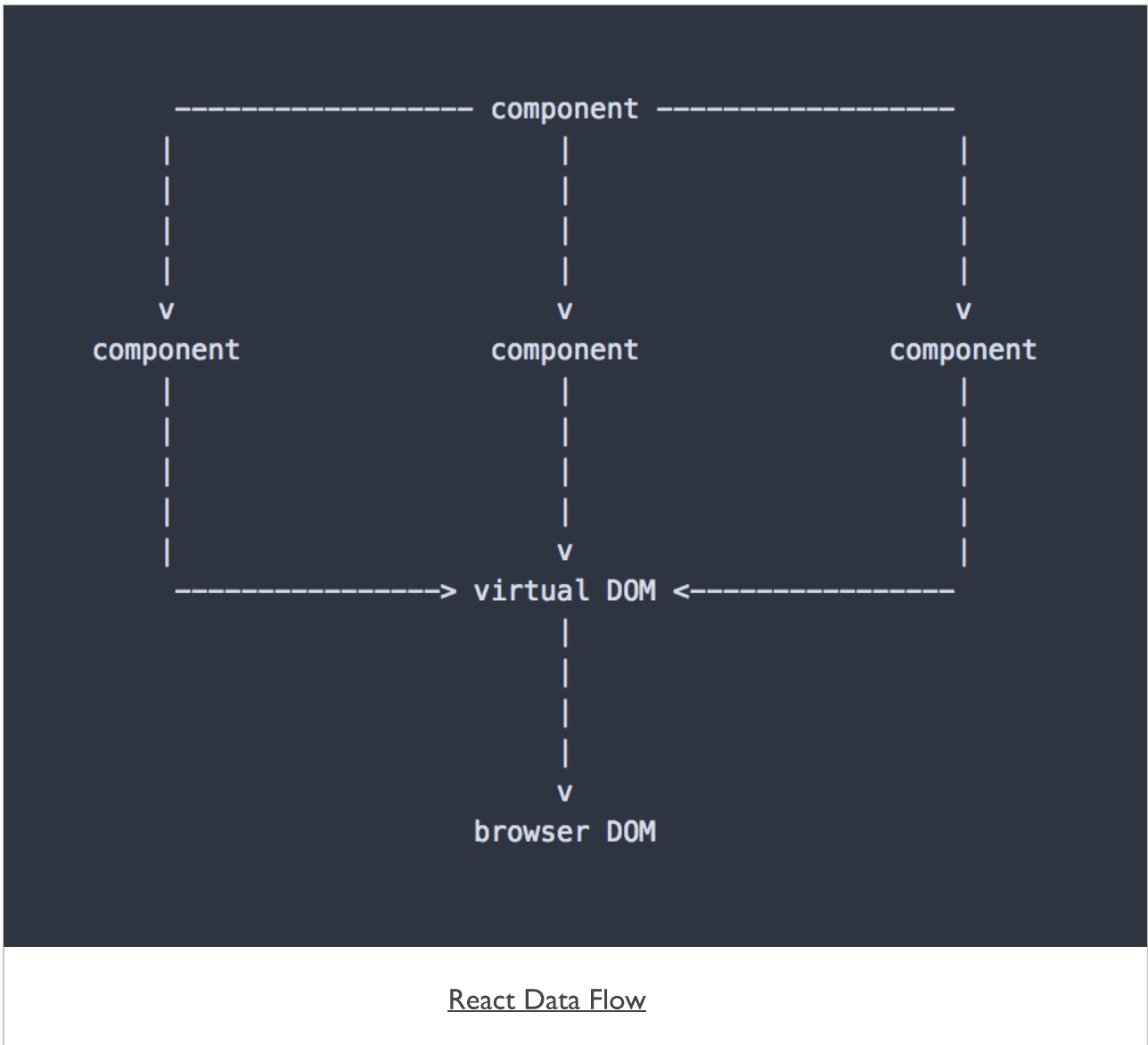
## a React approach to development - part 2

- 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

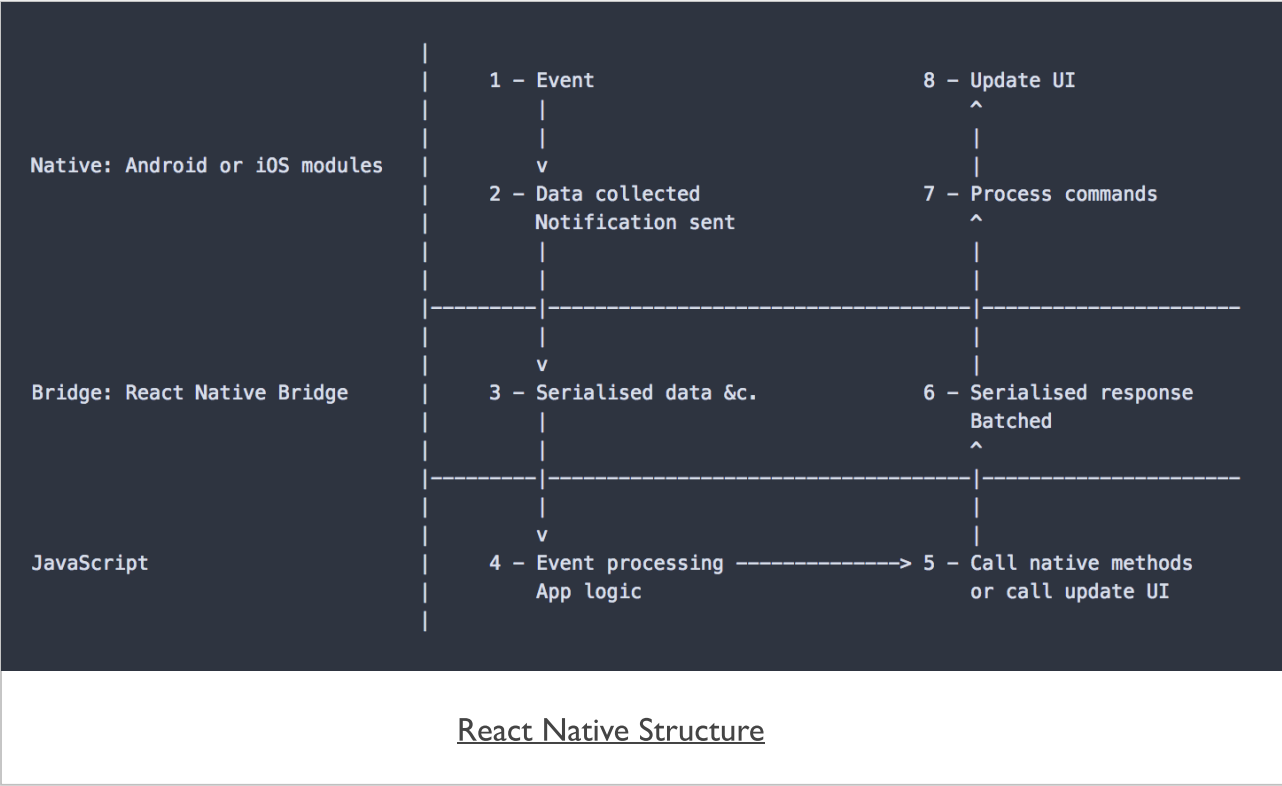
---

## data flow



# 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 I***

- 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

---

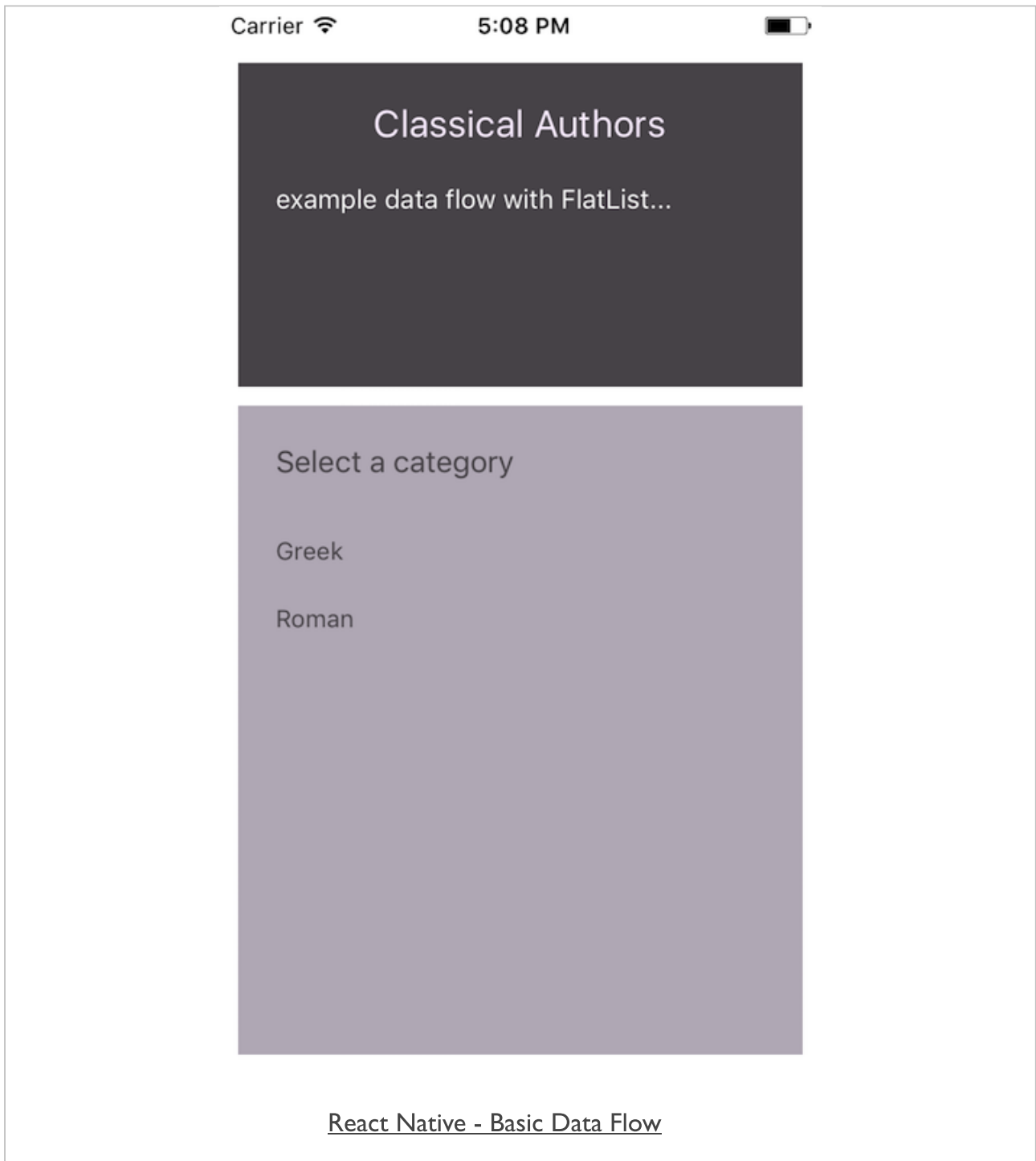
## ***data flow***

- 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 I***

- 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'  
>
```

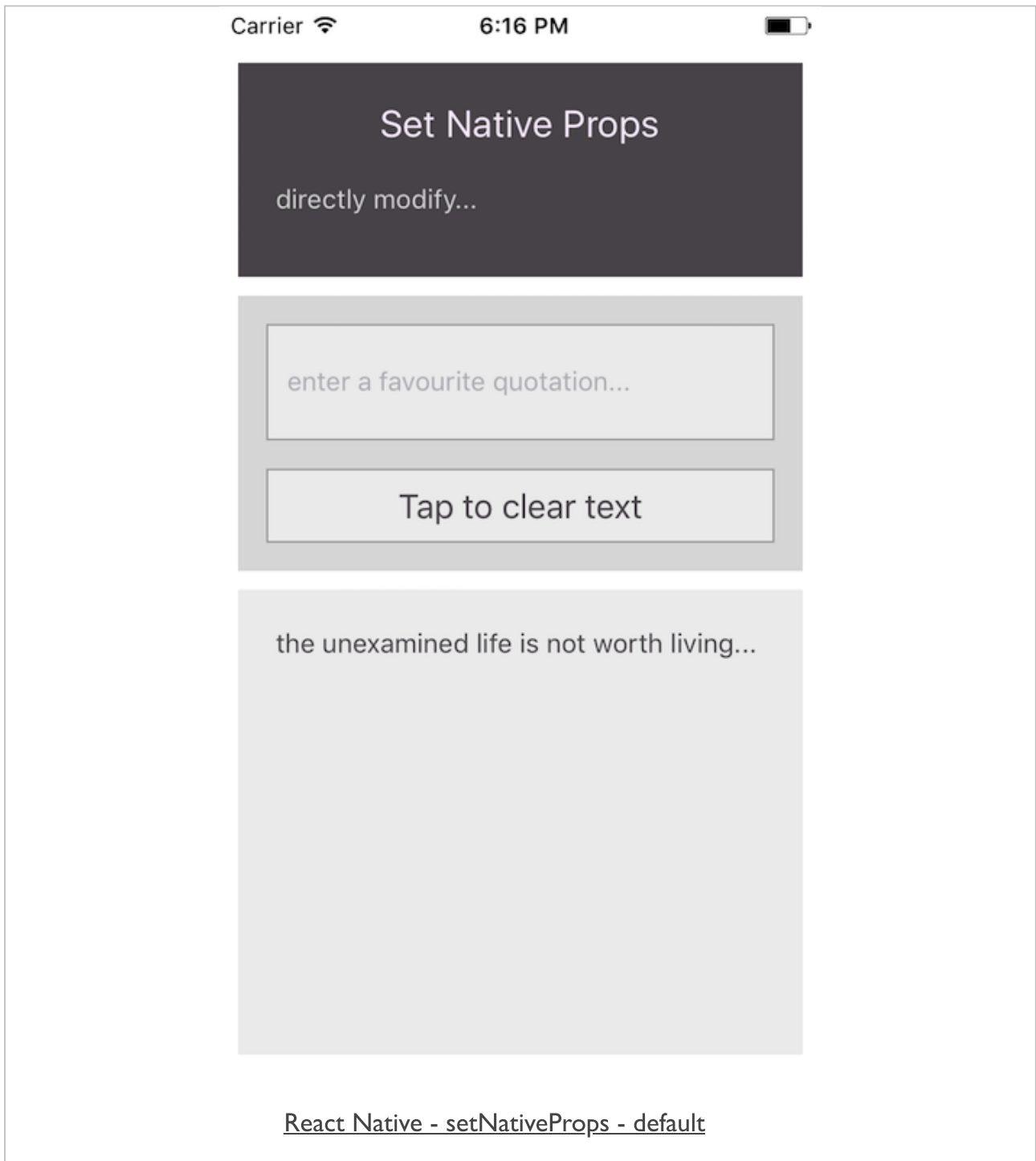
- add TextInput component and define reference

```
<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

---

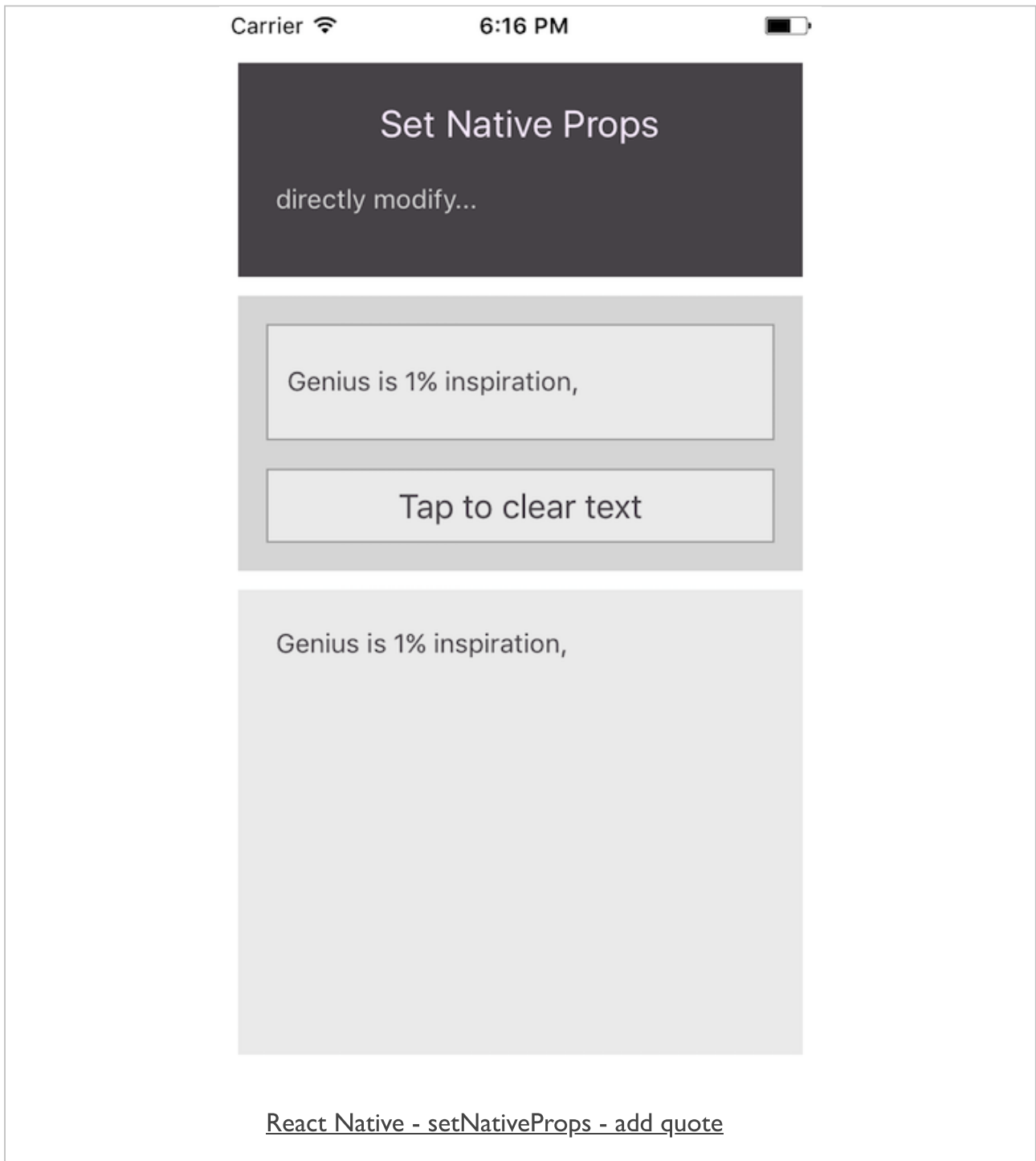
## *setNativeProps example - default*



# Image - React Native - Data Flow

---

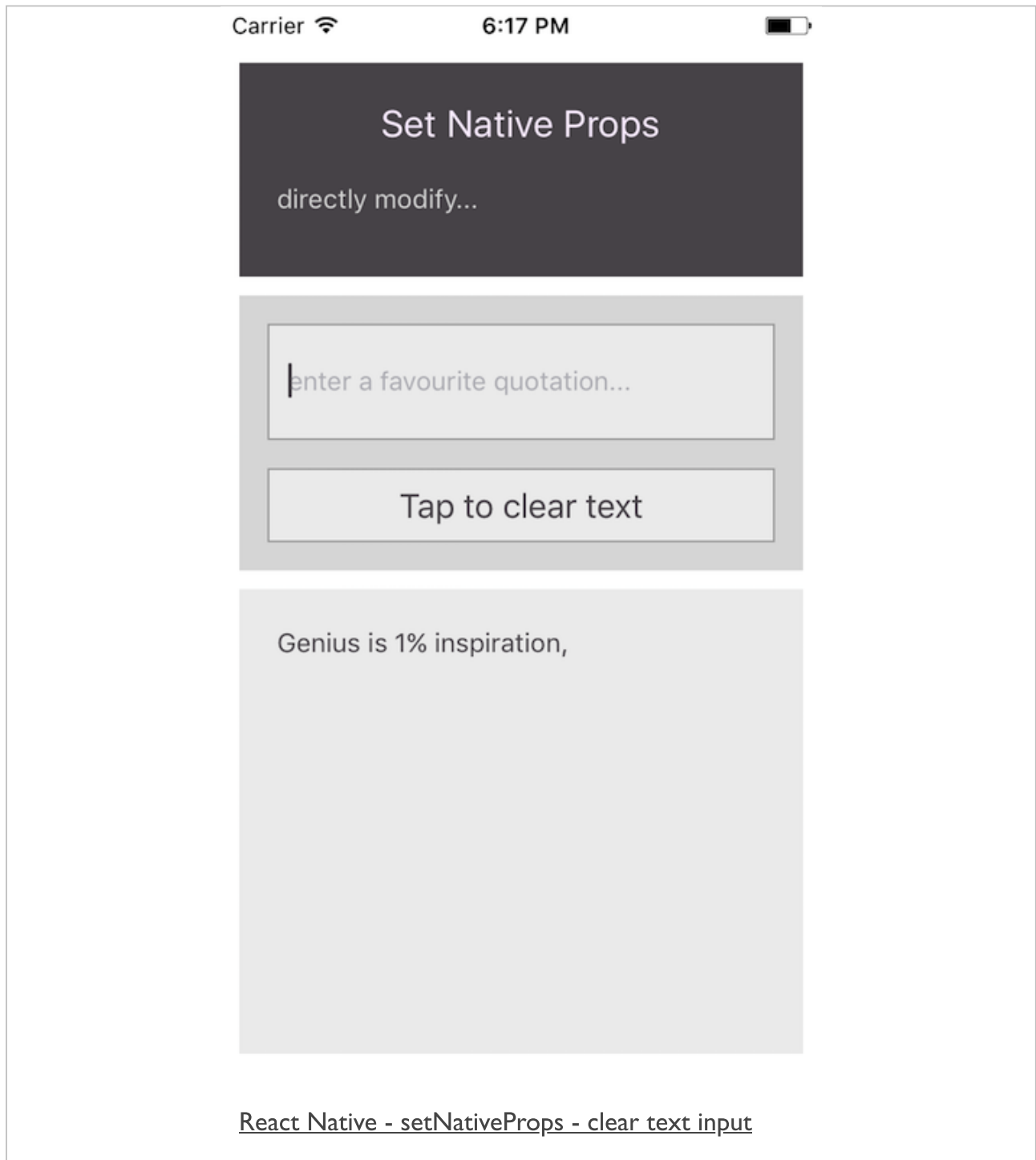
***setNativeProps example - add quote***



# 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/>

# React JavaScript Library - non-ES6

---

## **state - intro - part I**

- a component in React is able to house *state*
- *State* is inherently different from *props* because it is internal to the component
- it is particularly useful for deciding a view state on an element
  - *eg: we could use state to track options within a hidden list or menu*
  - *track the current state*
  - *change it relative to component requirements*
  - *then show options based upon this amended state*
- **NB:** considered bad practice to update state directly using `this.state`
  - *use the method `this.setState`*
- try to avoid storing computed values or components directly in *state*
- focus upon using simple data
  - *directly required for given component to function correctly*
- considered good practice to perform required calculations in the `render` function
- try to avoid duplicating `prop` data into `state`
  - *use the `props` data instead*

# React JavaScript Library - non-ES6

---

## state - intro - part 2

```
var EditButton = React.createClass({
  getInitialState: function() {
    return {
      editShow: true
    };
  },
  render: function() {
    if (this.state.editShow == false) {
      alert('edit button will be turned off...');
    }
    return (
      <button className="button edit" onClick={this.handleClick}>Edit</button>
    );
  },
  handleClick: function() {
    //handle click...
    alert('edit button clicked');
    //set state after button click
    this.setState({ editShow: false });
  }
});
```



# React Native - State

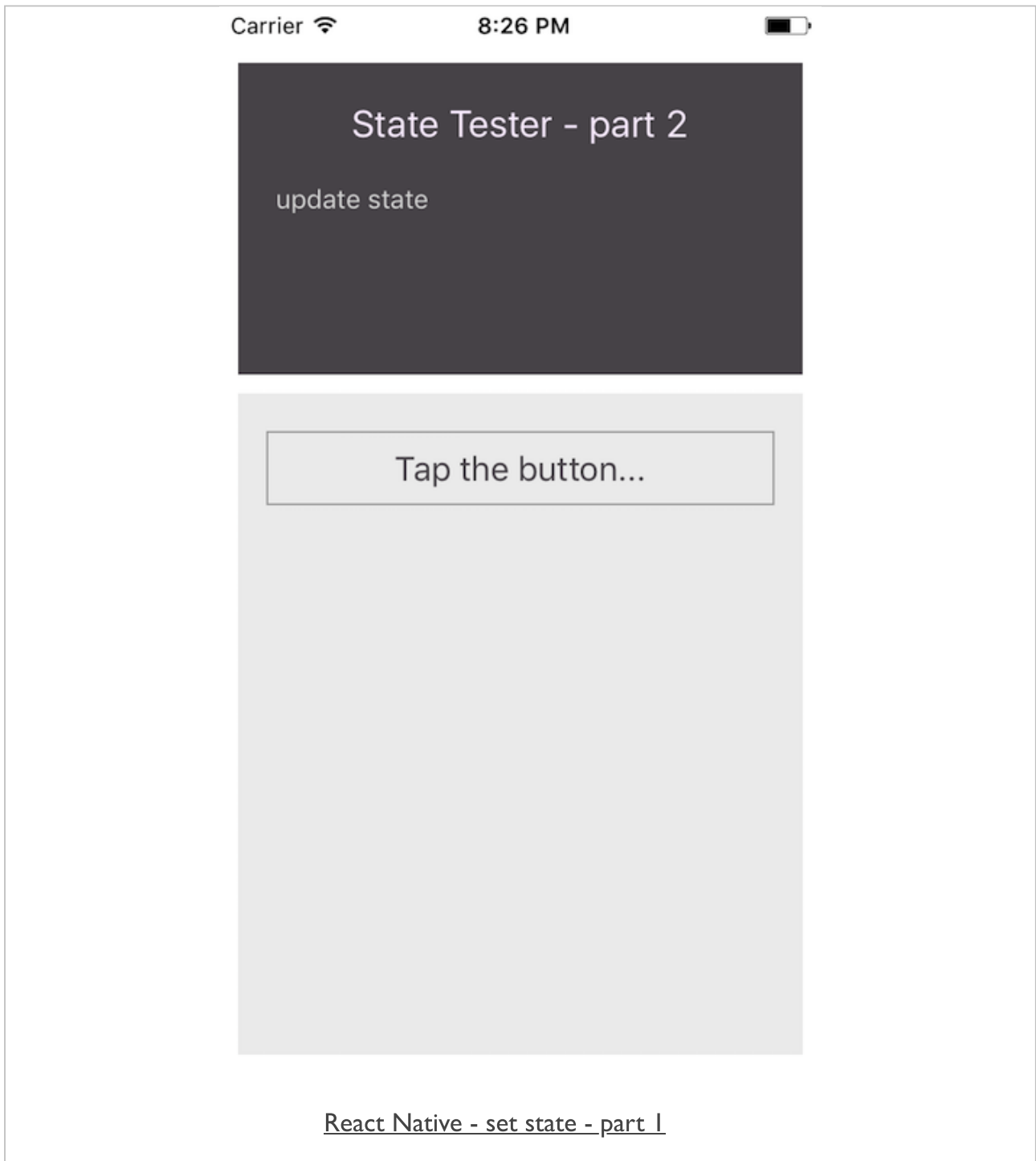
## component and constructor

```
// abstracted component for rendering *tape* text
class EditButton 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 set up
    super(props);
    // set initial state - e.g. text is shown
    this.state = { editShow: true };
  }
  // custom function to modify state on button click
  handleClick = () => {
    //set state after button click
    this.setState({ editShow: false });
  }
  // component render - check state of component...
  render() {
    if (this.state.editShow == false) {
      return (
        <Text style={styles.content}>
          Button has been removed...
        </Text>
      );
    } else {
      return (
        <View style={styles.buttonBox}>
          <Button
            onPress={this.handleClick}
            title={this.props.title}
            color='#585459'
          />
        </View>
      );
    }
  }
}
```

# Image - React Native - Set State

---

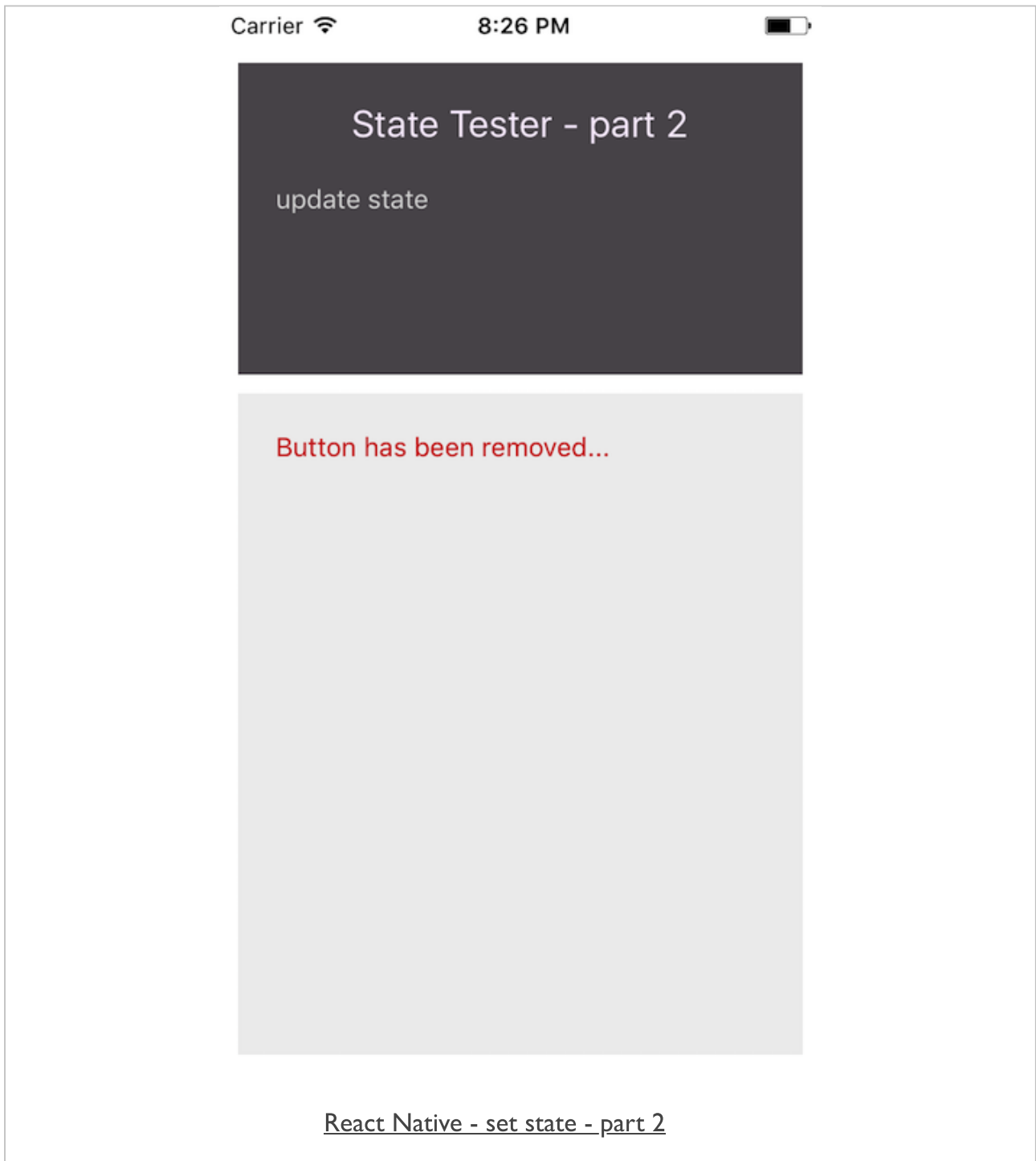
*component and constructor*



# Image - React Native - Set State

---

***component and constructor***



# React JavaScript Library - non-ES6

---

## state - intro - part 3

- when designing React apps, we often think about
  - **stateless children** and a **stateful parent**

*A common pattern is to create several stateless components that just render data, and have a stateful component above them in the hierarchy that passes its state to its children via props.*

### React documentation

- need to carefully consider how to identify and implement this type of component hierarchy
  1. Stateless child components
    - components should be passed data via *props* from the parent
    - to remain stateless they should not manipulate their *state*
    - they should send a callback to the parent informing it of a change, update etc
    - parent will then decide whether it should result in a *state* change, and a re-rendering of the DOM
  2. Stateful parent component
    - can exist at any level of the hierarchy
    - does not have to be the root component for the app
    - instead can exist as a child to other parents
    - use parent component to pass *props* to its children
    - maintain and update state for the applicable components

# React Native - Components

---

## ***stateful versus presentational***

- with React and React Native
  - *compose existing components*
  - *as well as create our own custom components*
- two important concepts and component types in React and React Native
- **stateful**
  - *stateful is a central point in memory*
  - *used to store information about the app or a component's state*
  - *also maintains the ability to modify and update*
- **stateless**
  - *stateless will calculate its internal state*
  - *it should not directly change or mutate this state*
  - *inherent benefit is that we now maintain a clear, transparent record*
  - *given the same inputs, it will always return the same output*

# React Native - Components

---

## *presentational*

- presentational components in a UI
  - *often a reflection of passed or received data*
  - *e.g. a list output of data or some text output for the user to read...*
- React Native UI composed of many smaller blocks
- each block should also be reusable, e.g.

```
class Heading extends Component {  
  render() {  
    return(  
      <View style={styles.headingBox}>  
        <Text style={styles.heading}>  
          { this.props.text }  
        </Text>  
      </View>  
    )  
  }  
}
```

- this component may now be reused for headings in the UI
- component itself does not have any state
- simply a *presentational* or *functional* component
- component is a pure function of props passed from its parent
  - *it does not mutate its arguments*

# React Native - Components

---

## *presentational and functional*

- consider such presentational components from their pure functional context
- rewrite our Heading component as follows,

```
function Heading(props) {  
  return (  
    <View style={styles.headingBox}>  
      <Text style={styles.heading}>  
        { this.props.text }  
      </Text>  
    </View>  
  )  
}
```

# React JavaScript Library - non-ES6

---

## state - intro - part 4

### 1. props vs state

- *in React, we can often consider two types of model data*
- *includes `props` and `state`*
- *most components normally take their data from `props`*
- *allows them to render the required data*
- *as we work with users, add interactivity, and query and respond to servers*
- *we also need to consider the `state` of the application*
- *`state` is very useful and important in React*
- *also important to try and keep many of our components stateless*

### 2. state

- *React considers user interfaces, UIs, as simple state machines*
- *acting in various states and then rendering as required*
- *in React, we simply update a component's state*
- *then render the new corresponding UI*



# React JavaScript Library - non-ES6

---

## state - intro - part 5

### I. How state works

- if there is a change in data in the application
  - *perhaps due to a server update or user interaction*
  - *quickly and easily inform React by calling `setState(data, callback)`*
- this method allows us to easily merge data into `this.state`
  - *re-renders the component*
- as re-rendering is finished
  - *optional `callback` is available and is called by React*
- this `callback` will often be unnecessary
  - *it's still useful to know it is available*

# React JavaScript Library - non-ES6

---

## state - intro - part 6

### 2. In state

- try to keep data in state to a minimum
  - *consider minimal possible representation of an application's state*
  - *helps build a stateful component*
- state should try to just contain minimal data
  - *data required by a component's event handlers to help trigger a UI update*
  - *if and when they are modified*
- such properties should also normally only be stored in `this.state`
- as we render the updated UI
  - *simply compute required information in the `render()` method based on this state*
  - *avoids need to keep computed values in sync in state*
  - *instead relying on React to compute them for us*

### 3. out of state

- in React, `this.state` should only contain minimal data
- minimum necessary to represent an application's UI state
- should contain
  - *computed value/values*
  - *React components*
  - *duplicated data from props*

# React JavaScript Library - non-ES6

---

## ***state - an example app - part I***

- a simple app to allow us to test the concept of stateful parent and stateless child components
- resultant app outputs two parallel `div` elements
- allow a user to select one of the available categories
- then view all of the available *authors*

```
//static test data...
var AUTHORS = [
  {id:1, category: 'greek', categoryId:1, author: 'Plato'},
  {id:2, category: 'greek', categoryId:1, author: 'Aristotle'},
  {id:3, category: 'greek', categoryId:1, author: 'Aeschylus'},
  {id:4, category: 'roman', categoryId:2, author: 'Livy'},
  {id:5, category: 'greek', categoryId:1, author: 'Euripides'},
  {id:6, category: 'roman', categoryId:2, author: 'Ptolemy'},
  {id:7, category: 'greek', categoryId:1, author: 'Sophocles'},
  {id:8, category: 'roman', categoryId:2, author: 'Virgil'},
  {id:9, category: 'roman', categoryId:2, author: 'Juvenal'}
];
```

- start with some static data to help populate our app
- `categoryId` used to filter unique categories
  - *again to help get all of our authors per category*

# React JavaScript Library - non-ES6

---

## **state - an example app - part 2**

- for `stateless` child components
  - *need to output a list of filtered, unique categories*
  - *then a list of authors for each selected category*
- first child component is the `CategoryList`
  - *filters and renders our list of unique categories*
  - *`onClick` attribute is included*
  - *state is therefore passed via callback to the `stateful` parent*

# React JavaScript Library

---

## state - an example app - part 3

```
//output unique categories from passed data...
var CategoryList = React.createClass({
  render: function() {
    var category = [];
    return (
      <div id="left-titles" className="col-6">
        <ul>
          {this.props.data.map(function(item) {
            if (category.indexOf(item.category) > -1) {
            } else {
              category.push(item.category);
              return (
                <li key={item.id} onClick={this.props.onCategorySelected.bind(null, item.category)}>
                  {item.category}
                </li>);
              }, this)}
          )}
        </ul>
      </div>
    );
  }
});
```

- the component is accepting props from the parent component
  - then informing this parent of a required change in state
  - change reported via a callback to the `onCategorySelected` method
  - does not change *state* itself
  - it simply handles the passed data as required for a React app

# React JavaScript Library - non-ES6

---

## state - an example app - part 4

- need to consider our second stateless child component
  - renders the user's chosen authors per category
  - user clicks on their chosen category
  - a list of applicable authors is output to the right side div

```
var AuthorList = React.createClass({
  render: function() {
    return (
      <div id="right-titles" className="col-md-6 col-sm-6 col-xs-6">
        <ul>
          {this.props.authors.map(function(item) {
            return (
              <li key={item.id}>{item.author}</li>
            );
          })}
        </ul>
      </div>
    );
  }
});
```

- this component does not set any state
- simply rendering the passed props data for viewing

# React JavaScript Library - non-ES6

---

## **state - an example app - part 5**

- to handle updates to the DOM, we need to consider our stateful parent
- this component passes the app's data as props to the children
- handles the setting and updating of the state for app as well
- as noted in the React documentation,

*State should contain data that a component's event handler may change to trigger a UI update.*

- for this example app
  - *only need to store the `selectedCategoryAuthors` in state*
  - *enables us to update the UI for our app*

# React JavaScript Library - non-ES6

---

## state - an example app - part 6

```
var Container = React.createClass({
  getInitialState: function() {
    return {
      selectedCategoryAuthors: this.getCategoryAuthors(this.props.defaultCategoryId)
    };
  },
  getCategoryAuthors: function(categoryId) {
    var data = this.props.data;
    return data.filter(function(item) {
      return item.categoryId === categoryId;
    });
  },
  render: function() {
    return (
      <div className="container col-md-12 col-sm-12 col-xs-12">
        <CategoryList data={this.props.data} onCategorySelected={this.onCategorySelected} />
        <AuthorList authors={this.state.selectedCategoryAuthors} />
      </div>
    );
  },
  onCategorySelected: function(categoryId) {
    this.setState({
      selectedCategoryAuthors: this.getCategoryAuthors(categoryId)
    });
  }
});
```



# React JavaScript Library - non-ES6

---

## **state - an example app - part 7**

- our `stateful` parent component sets its initial state
  - *including passed data and app's selected category for authors*
- helps set a default state for the app
  - *we can then modify as a user selects their chosen category*
- callback for this user selected category is handled in the `onCategorySelected` method
  - *updates the app's state for the chosen `categoryId`*
  - *then leads to the app re-rendering the DOM for any changes*
- we still have computed data in the app's state
  - *as noted in the React documentation,*

*this.state should only contain the minimal amount of data needed to represent your UIs state...*

- we should now move our computations to the `render` method of the parent component
  - *then update state accordingly*

# React JavaScript Library - non-ES6

---

## *state - an example app - part 8*

```
var Container = React.createClass({
  getInitialState: function() {
    return {
      selectedCategoryId: this.props.defaultCategoryId
    };
  },
  render: function() {
    var data = this.props.data;
    var selectedCategoryAuthors = data.filter(function(item) {
      return item.categoryId === this.state.selectedCategoryId;
    }, this);
    return (
      <div className="container col-md-12 col-sm-12 col-xs-12">
        <CategoryList data={this.props.data} onCategorySelected={this.onCategorySelected} />
        <AuthorList authors={selectedCategoryAuthors} />
      </div>
    );
  },
  onCategorySelected: function(categoryId) {
    this.setState({selectedCategoryId: categoryId});
  }
});
```

- state is now solely storing the categoryId for our app
- can be modified and the DOM re-rendered correctly

# React JavaScript Library - non-ES6

---

## ***state - an example app - part 9***

- we can then load this application
  - *passing data as props to the Container*
  - *data from JSON Authors*

```
var buildLibrary = React.render (  
  <Container data={AUTHORS} defaultCategoryId='1' />,  
  document.getElementById('library')  
);
```

- DEMO - state example

# React Native - Components

---

## *stateful example - part I*

- also create a simple example with React Native components
- start with a standard component structure for a stopwatch

```
class Stopwatch extends Component {  
  render() {  
    return (  
      <View>  
        <Text>Stopwatch</Text>  
      </View>  
    )  
  }  
}
```

# React Native - Components

---

## ***stateful example - part 2***

- need to define the initial state for this component
- couple of options, including
  - *constructor and class properties*
- e.g. constructor usage,

```
constructor(props) {  
  super(props);  
  this.state = {  
    seconds: 0  
  };  
}
```

# React Native - Components

---

## *stateful example - part 3*

- also create additional getter methods for other stopwatch values, e.g. minutes.

```
get watchMinutes() {  
  return (  
    this.state.seconds / 60  
  )  
}
```

- then reference seconds and minutes in the render function, e.g.

```
render() {  
  return (  
    <View>  
      <Text>Stopwatch: {`${this.watchMinutes} : ${this.state.seconds}`}</Text>  
    </View>  
  )  
}
```

# React Native - Components

---

## ***stateful example - part 4***

- still need to inform React of a change in state
  - *for each second that passes whilst the stopwatch is active*
- the state is immutable
  - *we can only update it by executing the `setState` function*
- in the component, add the following for a second counter for the stopwatch

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

# React JavaScript Library

---

## **state - minimal state - part I**

- to help make our UI interactive
  - use React's *state* to trigger changes to the underlying data model of an application
  - need to keep a minimal set of mutable state
- **DRY**, or *don't repeat yourself*
  - often cited as a good rule of thumb for this minimal set
- need to decide upon an absolute minimal representation of the state of the application
  - then compute everything else as required
  - eg: if we maintain an array of items
  - common practice to calculate array length as needed instead of maintaining a counter



# React JavaScript Library

---

## **state - minimal state - part 2**

- as we develop an application with React
  - *start dividing our data into logical pieces*
  - *then start to consider which is state*
- for example,
  - *is it from props?*
  - *if yes, this is probably not state in React*
  - *does it update or change over time? (eg: due to API updates etc)*
  - *if yes, this is probably not state*
  - *can you compute the data based upon other state or props in a component?*
  - *if yes, it is not state*
- need to decide upon our minimal set of components that mutate, or own state
  - *React is based on the premise of one-way data flow down the hierarchy of components*
  - *can often be quite tricky to determine*
- initially, we can check the following
  - *each component that renders something based on state*
  - *determine the parent component that needs the state in the hierarchy*
  - *a common or parent component should own the state*
    - *NB: if this can't be determined*
    - *simply create a basic component to hold this state*
    - *add component at the top of the state hierarchy*

## References

---

- Cordova API docs
  - *config.xml*
  - *Globalization*
  - *Hooks*
  - *Merges*
  - *Network Information*
- React Native
  - *MDN - super*
  - *React Native - Layout Props*