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

Fall Semester 2019 - Week 11

Dr Nick Hayward

React JavaScript Library - non-ES6

state - intro - part 1

- 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 });
  }
});
```

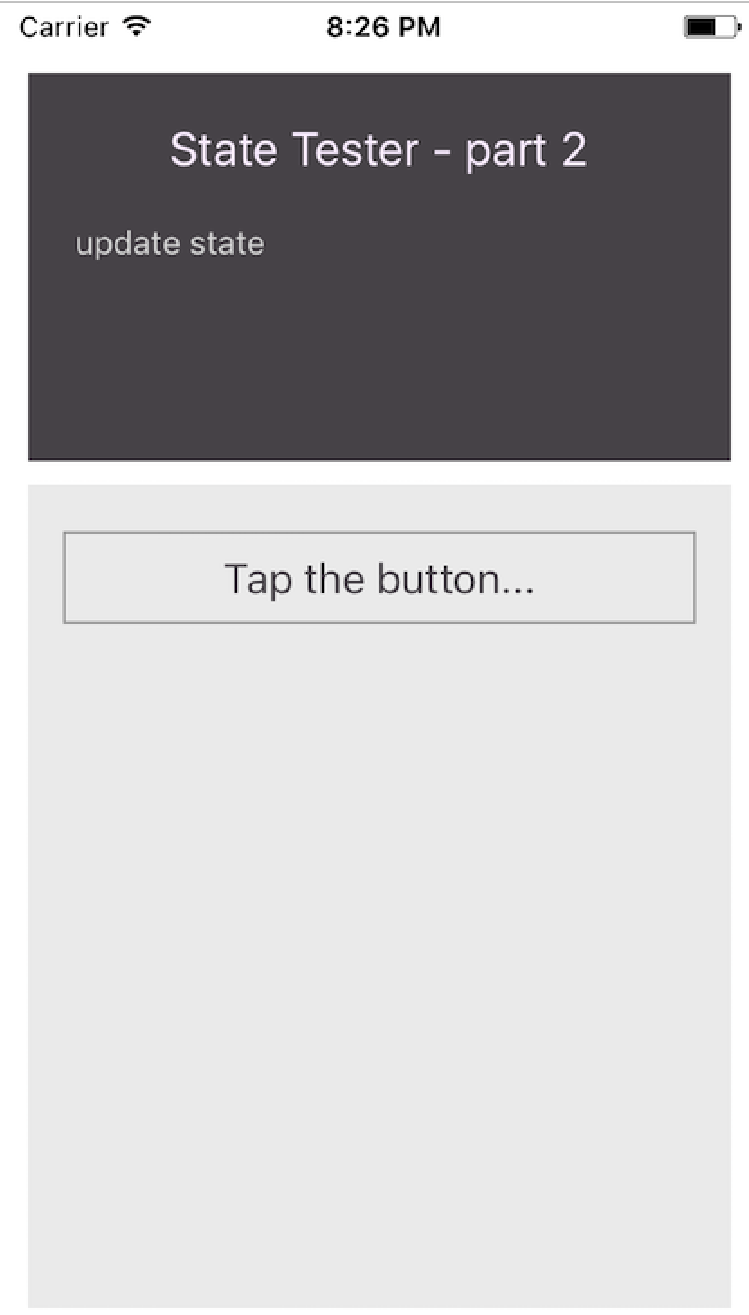
React Native - State

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 Componen
    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

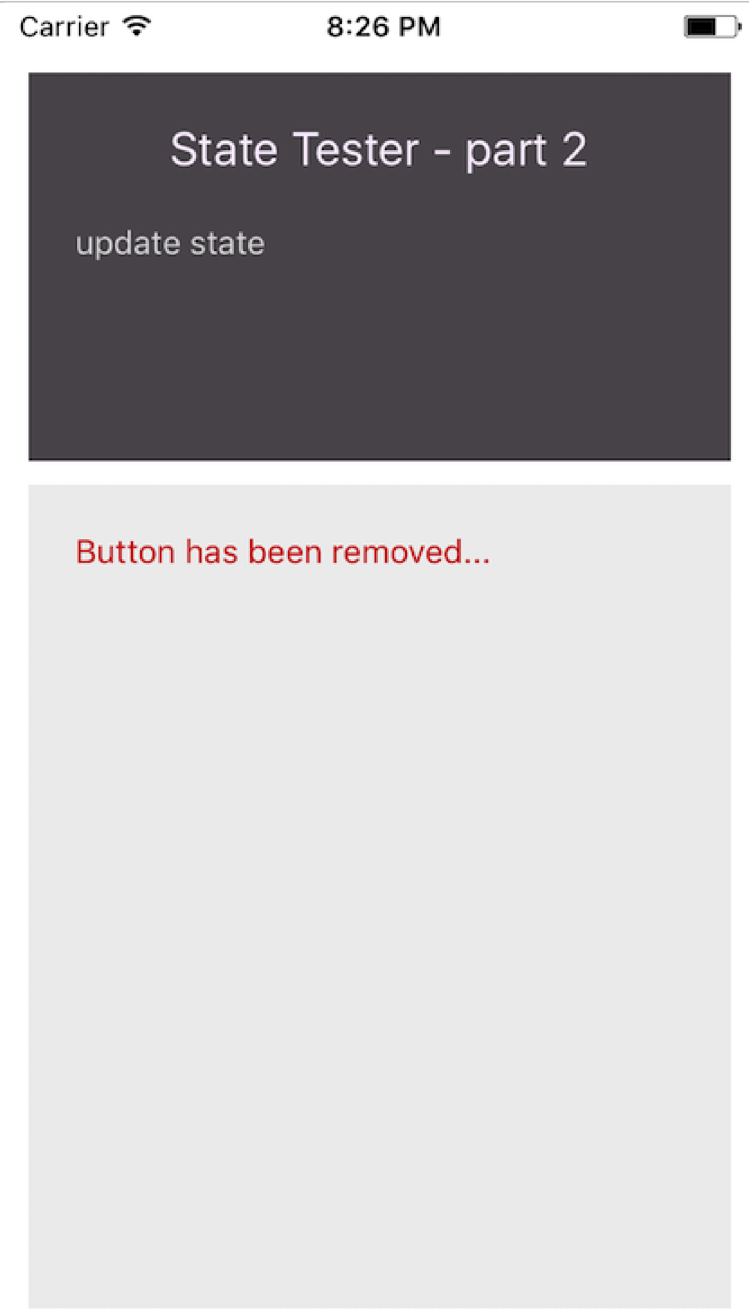
component and constructor



React Native - set state - part 1

Image - React Native - Set State

component and constructor


Carrier 8:26 PM

State Tester - part 2

update state

Button has been removed...

React Native - set state - part 2

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*
     - *arent 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

1. 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 1

- 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, it
            {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.defaultCatego
        };
    },
  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.onCategorySel
      <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.onCategoryS
        <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

Fun Exercise - State Usage

Watch the following gaming demo,

- Blocks

Then, consider the following relative to **state**

- how is state being used to initially define the application?
- how is state being updated to modify the game?
- how is state being used to keep scores in the game?
- how is state used to define difficulty levels in the game?

React Native - stateful example - part 1

- 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 1

- 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*

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

React Native - Lifecycle methods

mounting

- create stateful components in React and React Native
  - *monitor and use various lifecycle hooks*
  - *in addition to the `setState()` method...*
- start by considering component rendering
  - *better known as **mounting***
  - *various methods to cover each stage of component lifecycle*
- `componentWillMount`
  - *called immediately before component mounting*
  - *not recommended by Facebook's own documentation*
  - *better to use constructor for setting values &c.*
  - *calls to `setState` in this method will not trigger re-rendering*
- `componentDidMount`
  - *called after component mounting*
  - *use this method to initialise timers, any event listeners, fetch data, &c.*
  - *calls to `setState` will trigger re-render*
- `componentWillUnmount`
  - *called just before the component is unmounted and destriyed*
  - *normally use this method for component cleanup &c.*
  - *e.g. removing timers, stopping data requests, API calls &c.*

React Native - Lifecycle methods

updating

- components in React will be updated as and when their state is changed
  - *or if the parent component passes different props*
- we can take advantage of this data flow and pattern
  - *executing any required logic before a component gets updated...*
- React provides methods for such points in a components lifecycle
  - *thereby allowing us to handle updates*
- `componentWillReceiveProps`
  - *useful method to trigger a change in state due to a change in props*
  - *may also use this method to help collate changes in props*
  - *i.e. before and after updates, e.g.*

```
...
componentWillReceiveProps(updatedProps) {
  if (updatedProps !== this.props) {
    ...
  }
}
```

- `shouldComponentUpdate`
  - *React will usually re-render a component for each change in state*
  - *this method allows us to specify whether a component should update, how, &c.*
  - *e.g. re-render a component only for a specific update*
  - *return `false` from this method - a component will not be re-rendered*

React Native - Platform Structure

cross-platform

- React Native gives us a default directory and script structure
  - *part of the structure for a newly initialised app*
- modify stucture as app grows in complexity and scope
- React Native provides app initialisation files
  - *index.js & App.js*
- create a custom directory for app, e.g.
  - *`src` or `app` &c.*
  - *add directories for UI components, assets, scripts for APIs...*
- import `App.js` from `src` &c. directory

```
import App from './src/App';
```

React Native - Platform Structure

Android & iOS

- then start to add platform specific requirements
  - *including components, styles, images...*
- customisation is being encouraged with the `Platform` component. e.g.

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

- add checks to the logic of our app to add platform specific customisations,

```
const titles = Platform.select({
  ios: 'iOS custom title...',
  android: 'Android custom title...',
});
```

- to use this in our app's code
  - *do not need to specify iOS or Android*
  - *simply add the required output for `titles`. e.g.*

```
...
<View>
  <Text>{titles}</Text>
</View>
...
```

React Native - component usage

StatusBar

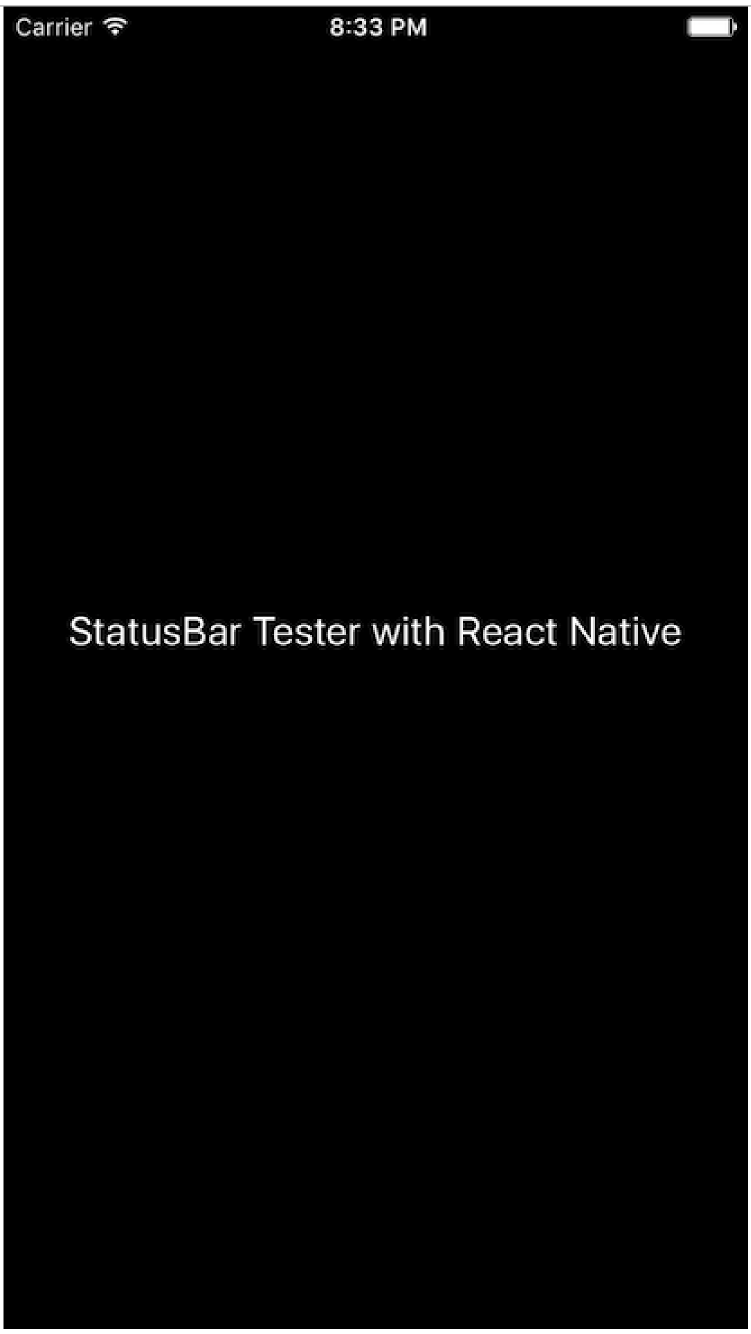
- add customisation to our app's *Status Bar*
  - *top bar with network icon, data, battery status, notification icons &c*
- various customisation options for each platform
  - *animate this bar*
  - *modify its colour*
  - *add custom style to match the current mode or status within our app*
- simple modification is to update the background colour
  - *from light to dark, and vice versa...*
  - *e.g. inform user of status change by animating the colour change and update*
- need to import the *StatusBar* component
  - *add an `animated` prop for the component*
  - *and specify a star for the bar itself*
- e.g. set the background colour of the bar to white

```
<StatusBar animated barStyle="light-content" />
```

- we might also set the `barStyle` to dark using the value `dark-content`
  - *sets colour of status bar text*
- we can only use the `barStyle` prop with iOS
- for Android, we can set props for `backgroundColor` and `translucent`
- additional options for working with the *StatusBar*, including static functions
  - *StatusBar*

Image - React Native - Component Usage

StatusBar



StatusBar Tester with React Native

React Native - StatusBar

React Native - component usage

images

- use `Image` component to add images
  - *and various static resources as well*
- `Image` component works with local and remote sources
  - *able to fetch remote images from a specified URL or server address*

```
...
<Image
  style={styles.image}
  resizeMode="contain"
  source={{
    uri: 'http://www.test.com/images/image.png'
  }}
/>
...
```

or

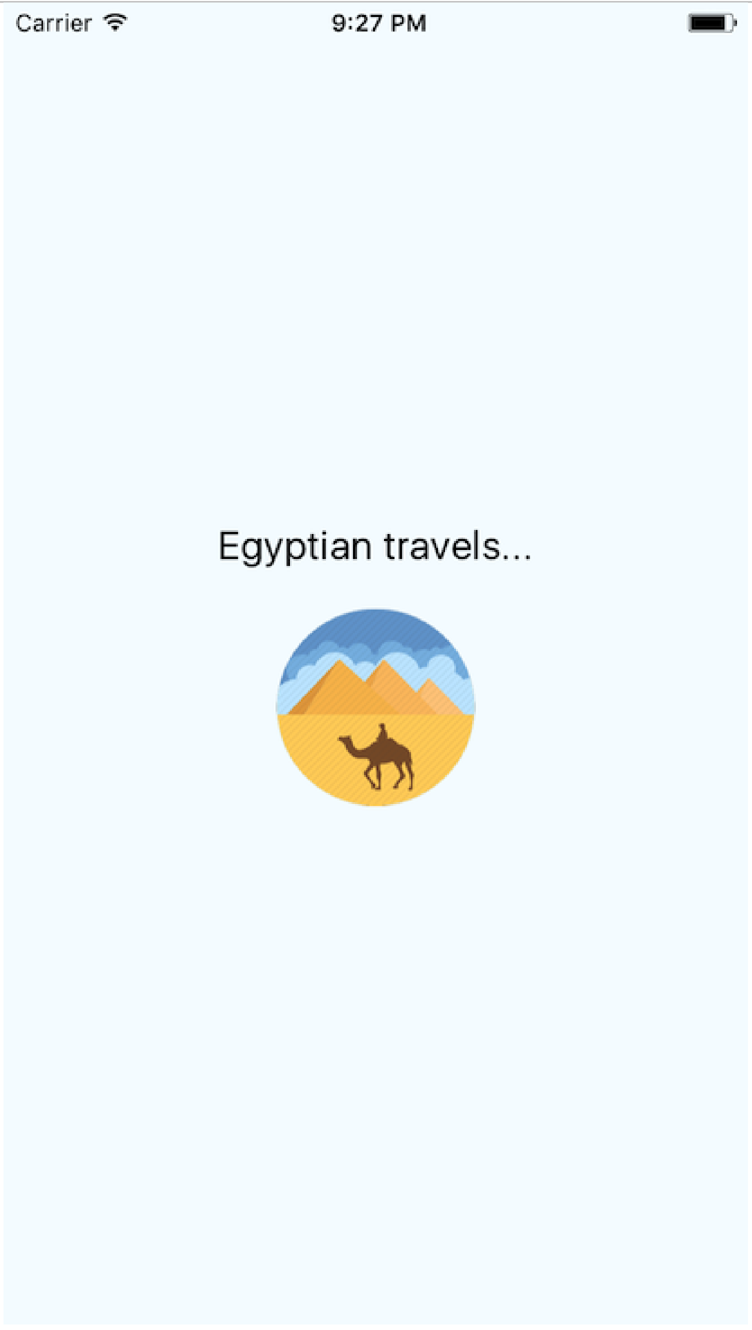
```
<Image
  style={styles.image}
  resizeMode="contain"
  source={require('./images/camel-icon.png')}
/>
```

- `resizeMode` prop may accept various values to help with layout and design
  - *cover, contain, stretch, repeat (only iOS), center*
- also check and use additional lifecycle props with images, including
  - *`onLoad`*
  - *`onLoadEnd`*
  - *`onLoadStart`*
- also get the size of a specifed image before rendering it to the View

```
Image.getSize
```

Image - React Native - Component Usage

Image component



React Native - Image component

React Native - component usage

activity indicator

- `ActivityIndicator` component gives us a default spinning loader for an app
  - *a small default component*
  - *useful for async loading, animations...*
- in addition to standard `View` props - also accepts the following
  - *animating - boolean value to determine whether to spin or not*
  - *color - specify the foreground colour of the spinner*
  - *size - pass `small` or `large` string for iOS, and a size value for Android*

React Native - component usage

activity indicator - example

- might want to use the `ActivityIndicator` to delay showing an image
- add a property to *state* - use as a simple boolean check for loading of the image
- initial *state* set as follows,

```
state = {
  showImage: false,
  loading: false
}
```

- image is not shown by default
  - *and the `ActivityIndicator` is not visible or active either*
- create a function to allow us to update the state
  - *will show the activity indicator and image*
- we're using ES6 classes for these examples
  - *need to start binding our functions as we pass them as props*
  - e.g.

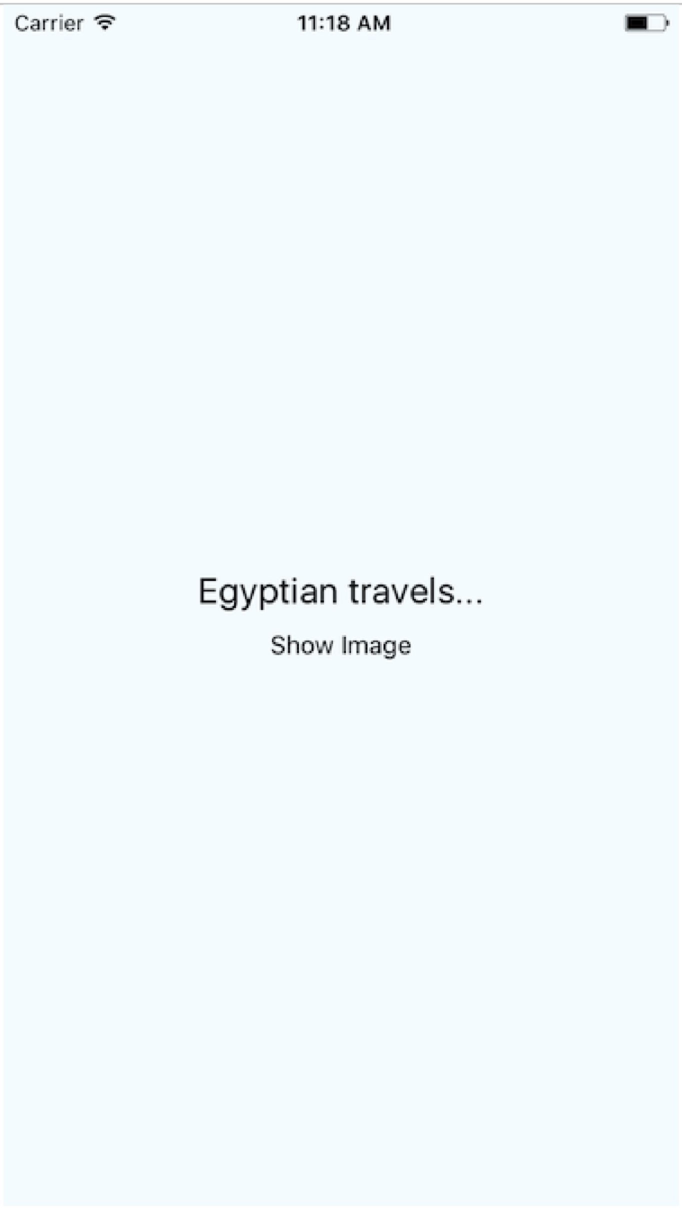
```
// instantiate object
constructor(props) {
  super(props);
  // bind function
  this.showImage = this.showImage.bind(this);
}
```

- `showImage` function can now be added

```
showImage() {
  this.setState({
    loading: true
  });
  setTimeout(() => {
    this.setState({
      showImage: true,
      loading: false
    })
  }, 2500)
}
```

Image - React Native - Component Usage

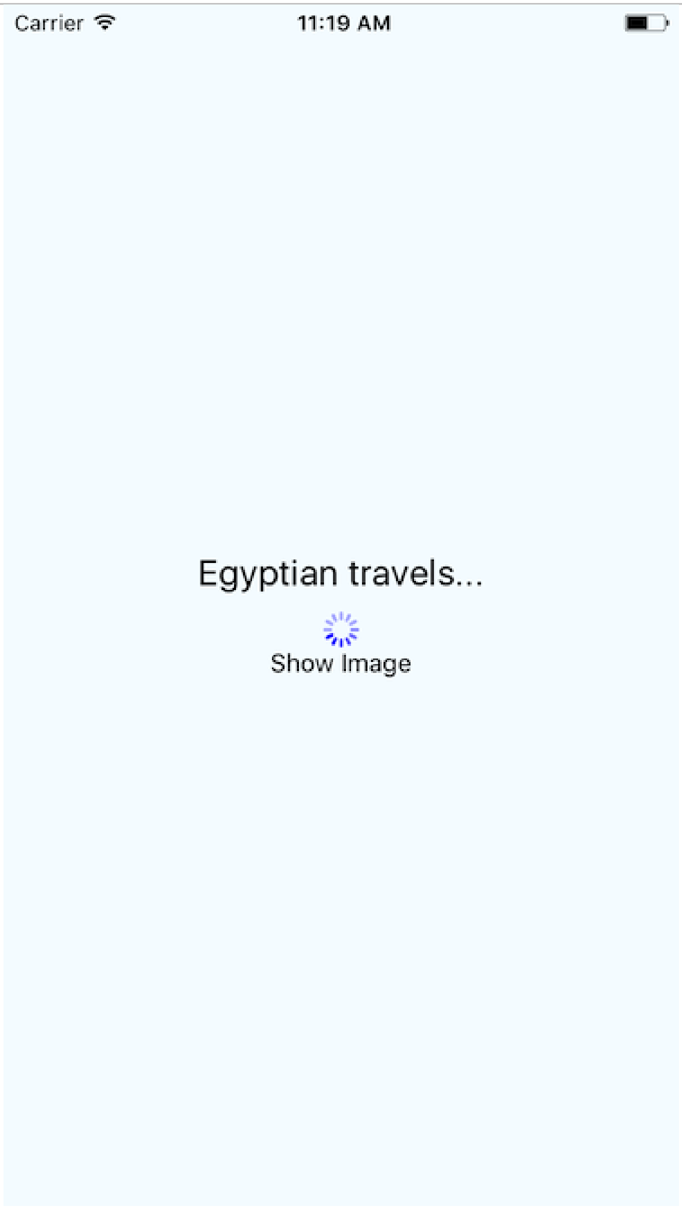
ActivityIndicator component - part 1



React Native - ActivityIndicator component

Image - React Native - Component Usage

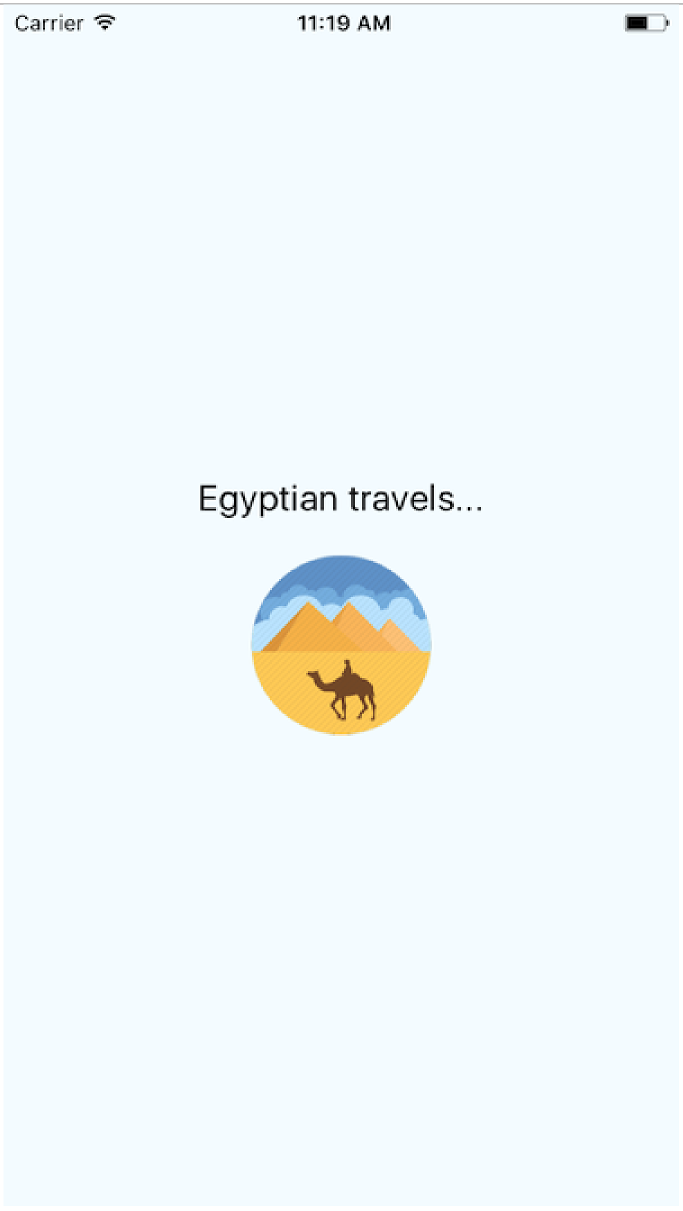
ActivityIndicator component - part 2



React Native - ActivityIndicator component

Image - React Native - Component Usage

ActivityIndicator component - part 3



React Native - ActivityIndicator component

React Native - component usage

custom modal

- React Native also supports a `Modal` component by default
- use it for success messages, feedback or prompts to a user, &c.
- also nest various child components to create the necessary output
- `Modal` component will accept the following props
  - *animationType*
  - *Transparent*
  - *Visible*
  - *onShow*
- also some custom props for each mobile platform
  - *e.g.* `presentationStyle` *for iOS*

React Native - component usage

custom modal - example

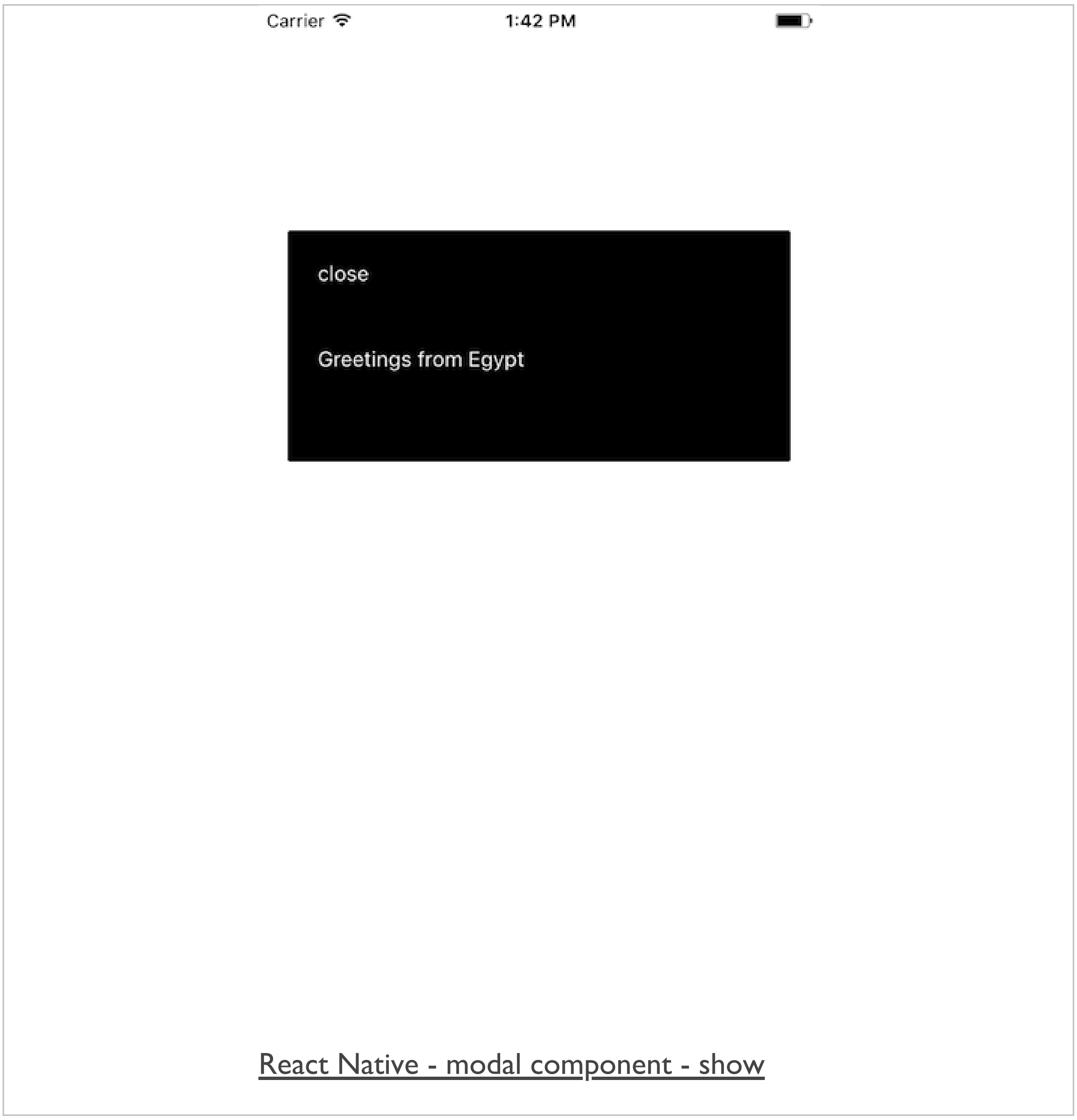
```
...
state = {
  modalVisible: true,
}

setModalVisible(visible) {
  this.setState({modalVisible: visible});
}

<Modal
  animationType="slide"
  transparent={false}
  visible={this.state.modalVisible}
  >
  <View style={styles.modal}>
    <TouchableHighlight onPress={() => {
    this.setModalVisible(!this.state.modalVisible)
    }}>
      <Text style={styles.modalClose}>close</Text>
    </TouchableHighlight>
    <Text style={styles.modalText}>Greetings from Egypt</Text>
  </View>
</Modal>
```

Image - React Native - Component Usage

custom modal component - part 1

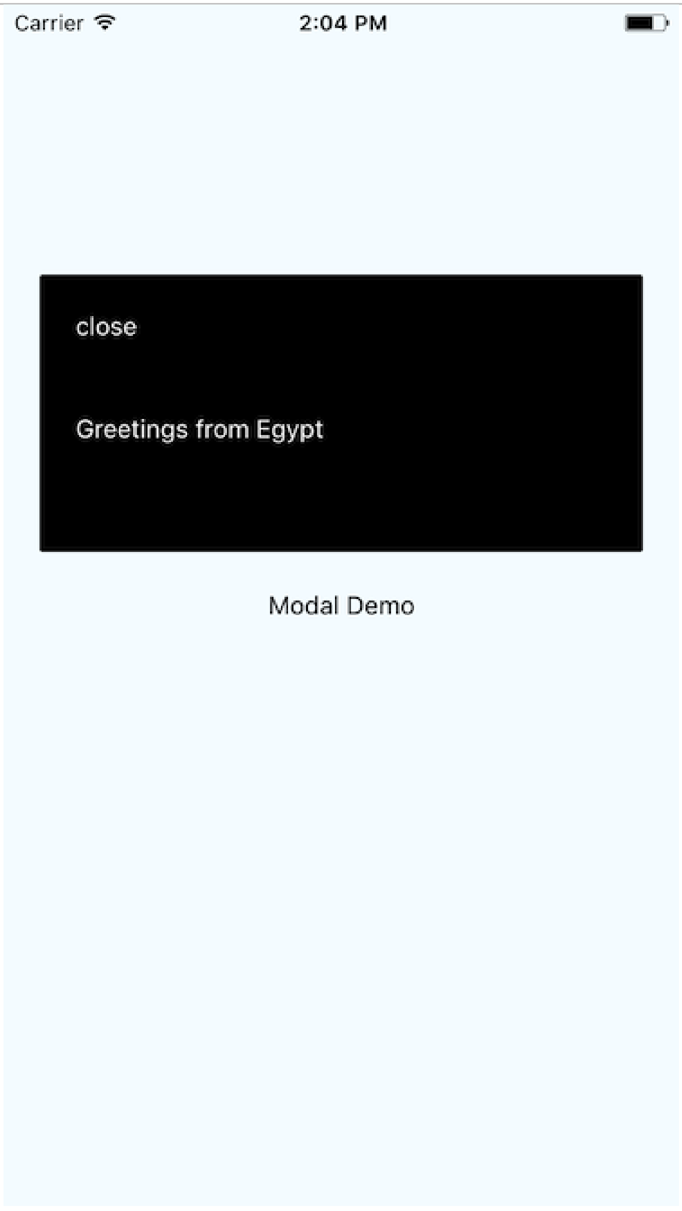


Greetings from Egypt

React Native - modal component - show

Image - React Native - Component Usage

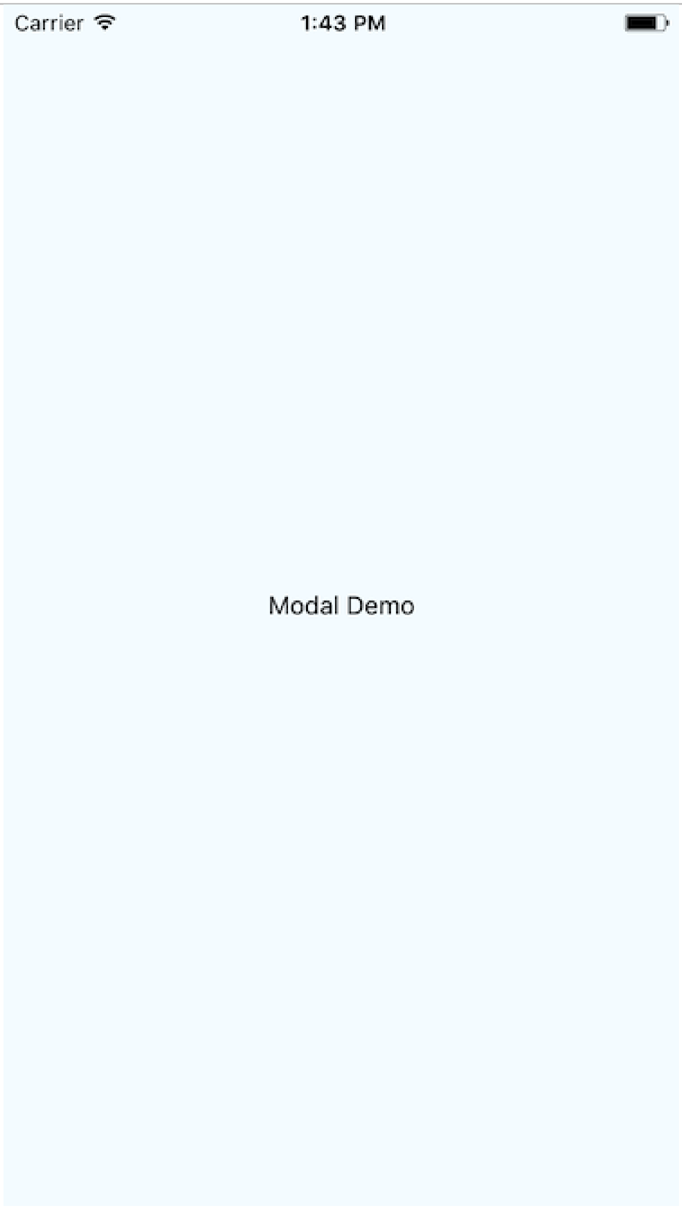
custom modal component - part 2



React Native - modal component - show with transparency

Image - React Native - Component Usage

custom modal component - part 3



React Native - modal component - closed

Mobile Design & Development - UI Components & Usage

Fun Exercise

Four groups, two apps

- Fashion - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/fashion/
- Travel Notes - http://linode4.cs.luc.edu/teaching/cs/demos/422/videos/travelnotes/

For each app, consider the following

- define UI components for the app?
- which components may be reused to create different effects?
- which components could be abstracted to extend a parent component?
- how is the UI influenced by the use of such components?

~ 10 minutes

References

- React DevTools
- React Native - Layout Props
- React Native - StatusBar