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

Fall Semester 2017 - Week 14 - React & React Native Notes

Dr Nick Hayward

Contents

Final Demo and Presentation

Presentation & demo: 8th December 2017 @ 2.45pm

Course total = 40%

- continue to develop your app concept and prototypes
  - *develop application using any of the technologies taught during the course*
  - *again, combine technologies to best fit your mobile app*
- if the app uses Apache Cordova
  - *implement a custom Cordova plugin for a native mobile OS*
  - *e.g. Android or iOS*
- produce a working app
  - *as far as possible try to create a fully working app*
  - *explain any parts of the app not working...*
- explain choice of technologies for mobile app development
  - *e.g. data stores, APIs, modules, &c.*
- explain design decisions
  - *outline what you chose and why?*
  - *what else did you consider, and then omit? (again, why?)*
- which concepts could you abstract for easy porting to other platform/OS?
- describe patterns used in design of UI and interaction

Final Report

Report due on 15th December 2017 by 2.45pm

- final report outline - coursework section of website
  - *PDF*
  - *group report*
  - *extra individual report*

Group Updates

- what is currently working?
- which data store?
- what is left to add or fix? features, UI elements, interactions...
- who is working on what? logic, design, testing, research...
- ...

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

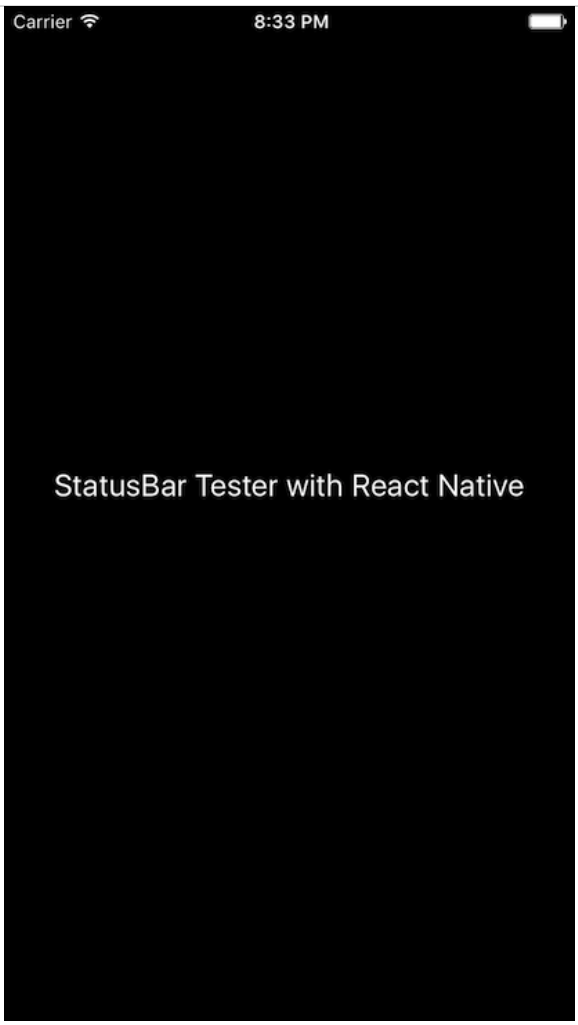
- add customisation to our app's *Status Bar*
  - *top bar with network icon, data, battery status, notification icons &c*
- various customisation options for each platform
  - *animate this bar*
  - *modify its colour*
  - *add custom style to match the current mode or status within our app*
- simple 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



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

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)
}
```

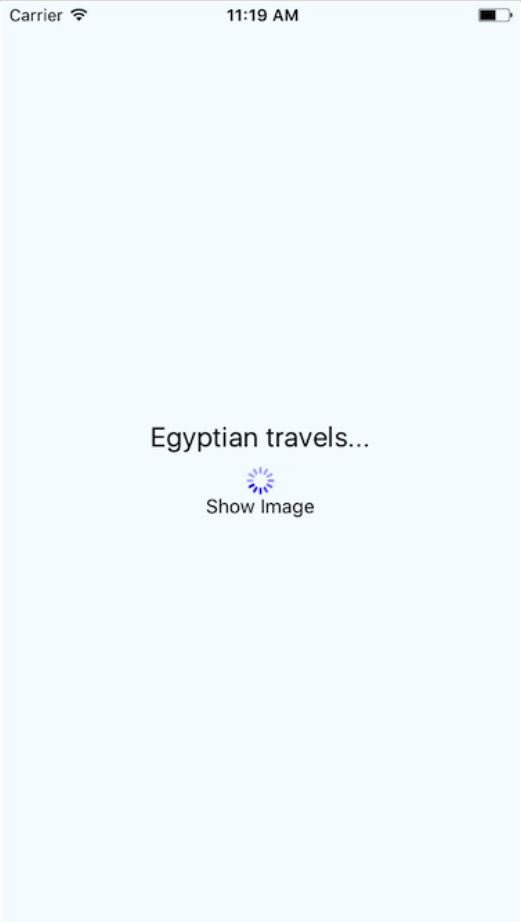
Image - React Native - Component Usage

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

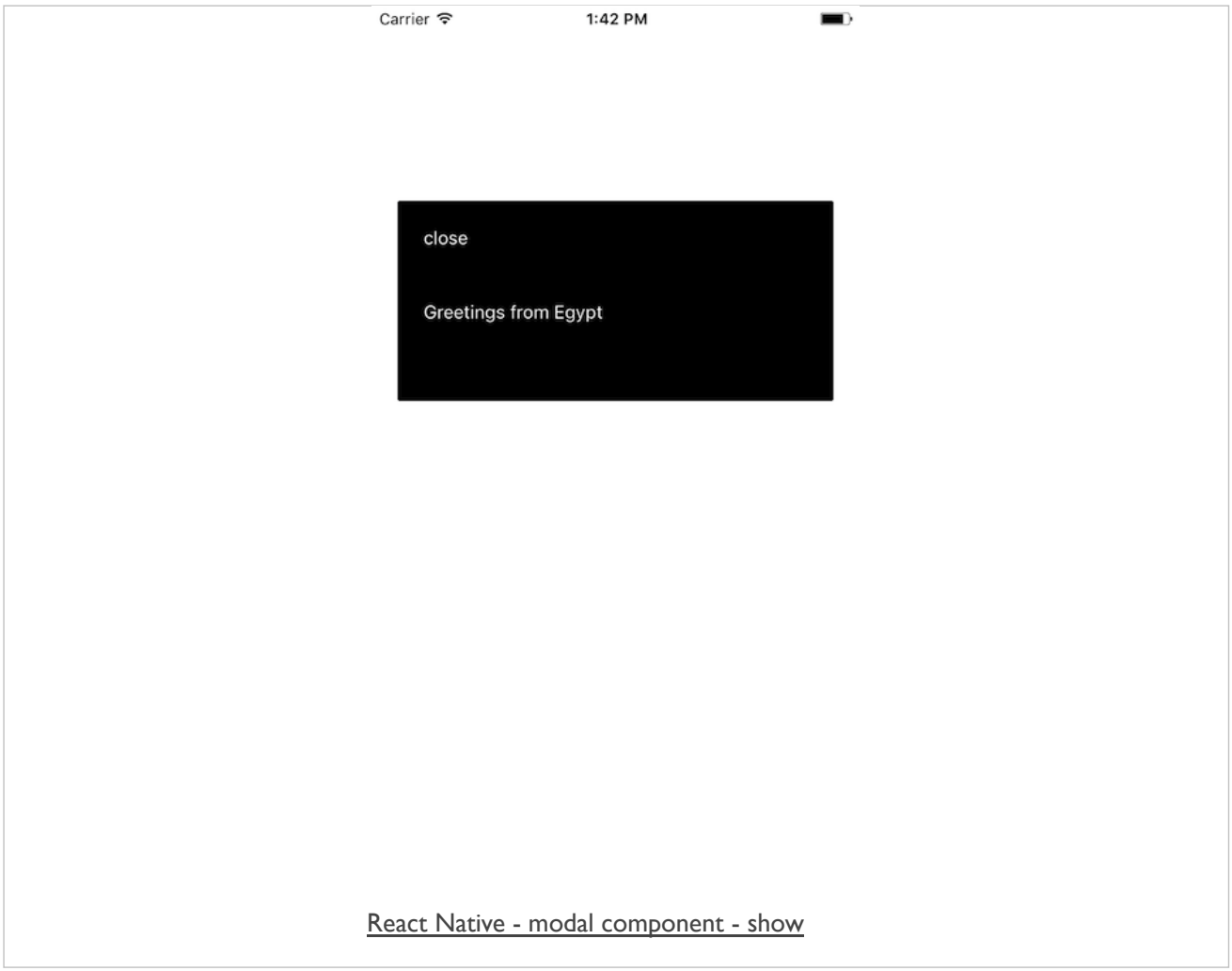
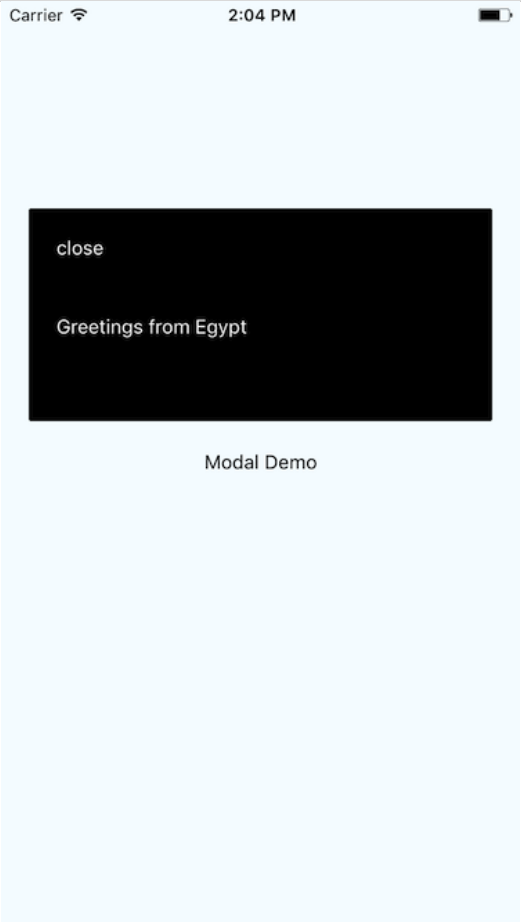


Image - React Native - Component Usage

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

React Native - component usage

lists - FlatList

- React Native provides suggested view components for lists
  - *two primary examples include `FlatList` and `SectionList`*
- `FlatList` is meant to be used for long lists of data
  - *in particular where data items may change during the lifecycle of an app*
- `FlatList` will only render elements currently shown on screen
  - *not all of the available elements at the same time*

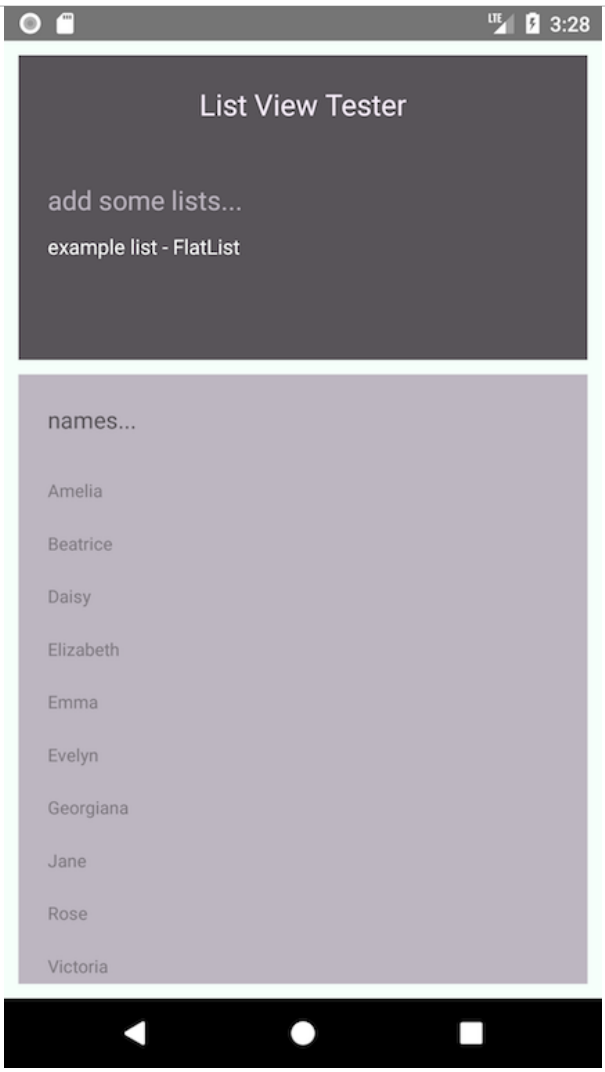
```
<FlatList
    data={[
        {key: 'Amelia'},
        {key: 'Beatrice'},
        {key: 'Daisy'},
    ]
    }
    renderItem={({item}) => <Text style={styles.listItem}>{item.key}</Text>}
/>
```

- component expects two *props*
  - *`data` for the list itself*
  - *`renderItem` to define the output structure for each list item*

```
renderItem={() => <Text></Text>}
```

Image - React Native - Component Usage

lists - FlatList



React Native - FlatList

React Native - component usage

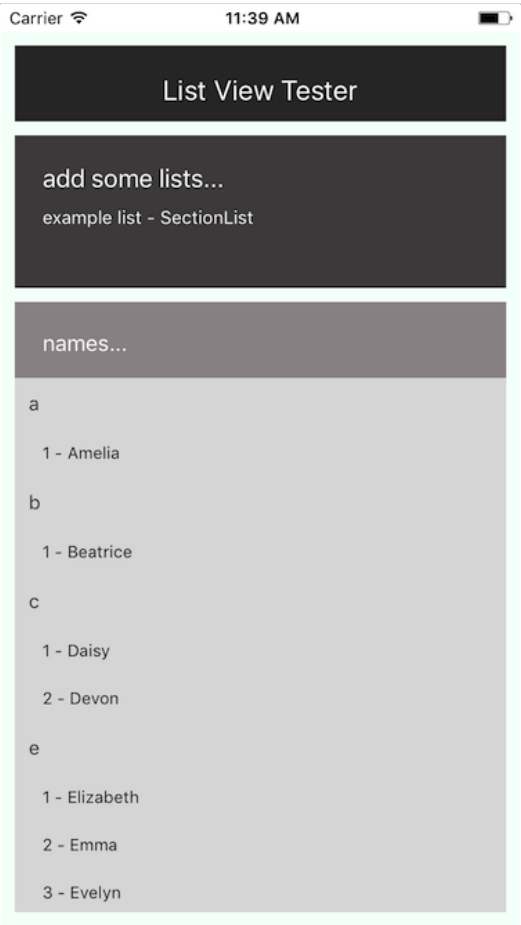
lists - SectionList

- may also create section breaks in a list of data. e.g.

```
<SectionList
  sections={[
        {title: 'a', data:[{key: 1, name: 'Amelia'}]},
        {title: 'b', data:[{key: 1, name: 'Beatrice'}]},
    {title: 'c', data: [{key: 1, name: 'Daisy'}, {key: 2, name: 'Devon'}]},
        {title: 'e', data: [{key: 1, name: 'Elizabeth'}, {key: 2, name: 'Emma'}, {key: 3, name: 'Evelyn'}]},
        {title: 'g', data:[{key: 1, name: 'Georgiana'}]},
        {title: 'j', data:[{key: 1, name: 'Jane'}]},
        {title: 'r', data:[{key: 1, name: 'Rose'}]},
        {title: 'v', data: [{key: 1, name: 'Victoria'}, {key: 2, name: 'Violet'}]},
        {title: 'y', data:[{key: 1, name: 'Yvaine'}]},
    ]}
    //keyExtractor={item => item}
  renderItem={({item}) => <Text style={styles.listItem}>{item.key} - {item.name}</Text>}
  renderSectionHeader={({section}) => <Text style={styles.heading4}>{section.title}</Text>}
/>
```

Image - React Native - Component Usage

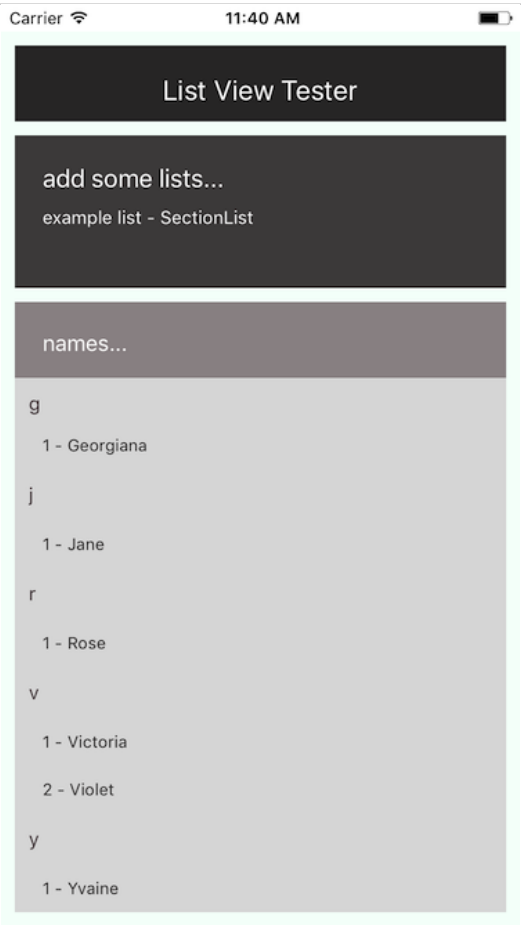
lists - SectiontList - top



React Native - SectionList - top

Image - React Native - Component Usage

lists - SectiontList - bottom



React Native - SectionList - bottom

React Native - component usage

ScrollView

- scrolling in React Native apps is achieved with a generic scrolling container
  - *`ScrollView`*
- specific view container can itself accept multiple child components and views
- scrollview container option to specify direction
  - *either horizontal or vertical*
- general usage
  - *add a `ScrollView` using the same general pattern as a standard `View` component*
  - *return a `ScrollView` as eiher the primary container for a component*
  - *or a child of a standard `View`*
  - *an app's screen may either scroll top to bottom*
  - *or simply present a component with scroll features*

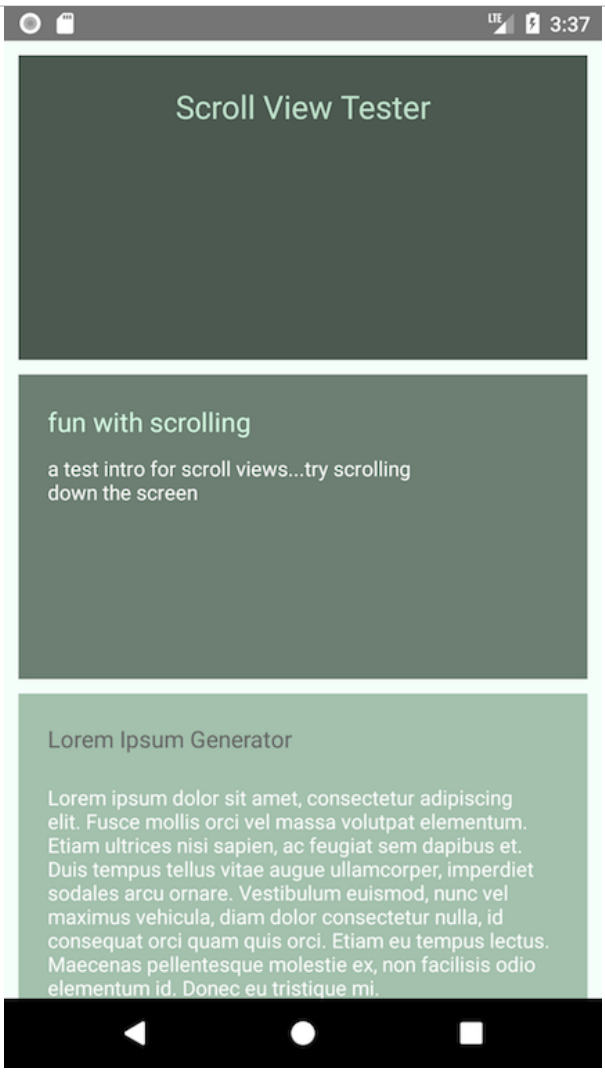
React Native - component usage

ScrollView - example

```
export default class ScrollTester extends Component {
  render() {
    return (
                <View style={styles.container}>
                    <View style={styles.headingBox}>
            <Text style={styles.heading1}>
                Scroll View Tester
            </Text>
            </View>
            <View style={styles.subHeadingBox}>
            <Text style={styles.heading2}>
                {intro.heading}
            </Text>
            <Text style={styles.content}>
                {intro.description}
            </Text>
            </View>
                    <ScrollView>
                    <View style={styles.contentBox}>
            <Text style={styles.heading3}>
                Lorem Ipsum Generator
            </Text>
            <Text style={styles.content}>
                            ...
                        </Text>
            </View>
                    </ScrollView>
        </View>
    );
  }
}
```

Image - React Native - Component Usage

lists - ScrollView



React Native - ScrollView

React Native - Component usage

text input

- a default component to handle user text input
- component `TextInput` is similar to a standard input field
  - *allowing a user to simply enter any required text content*
- to use `TextInput` with an app
  - *need to add the default module from React Native*
  - *add as part of the standard `import` statement*
- `TextInput` component includes a useful *prop*, `onChangeText`
  - *accepts callback function for each time text is changed in input field*
- also includes a complementary *prop*, `onSubmitEditing`
  - *handles text as it is submitted*
  - *again using a defined callback function*

React Native - Component usage

text input - props usage

- might accept user text input for a given value
  - *such as a name, place, &c.*
- then dynamically update the *view*
- e.g.

```
<TextInput
  style={styles.textInput}
  placeholder={this.state.quoteInput}
  onChangeText={(quoteText) => this.setState({quoteText})}
/>
```

React Native - Component usage

text input - props and state

- example relies upon calling and setting state for the app
  - *relative to `TextInput` and various `Text` components*
- simple constructor for this app
  - *pass required `props` and define intial values for `state`*

```
export default class TextUpdater extends Component {
  constructor(props) {
    super(props);
    this.state = {
      quoteInput: 'enter a favourite quotation...',
      quoteText: 'the unexamined life is not worth living...'
    };
  }
}
```

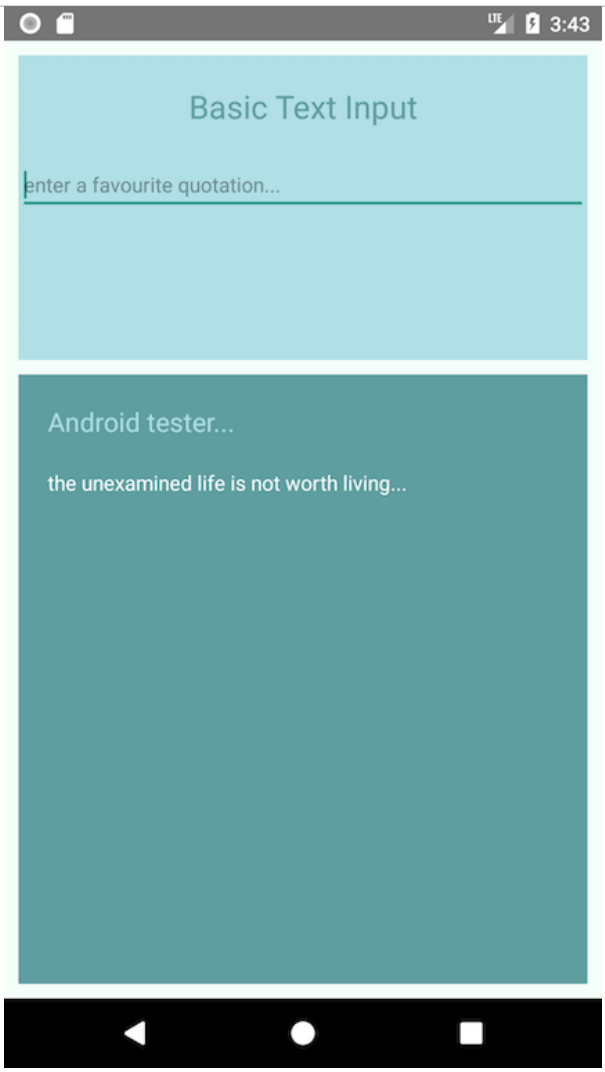
- then use the properties on `state`
  - *to set initial values for the text input field and the text output,*

```
<TextInput
  style={styles.textInput}
  placeholder={this.state.quoteInput}
  onChangeText={(quoteText) => this.setState({quoteText})}
/>
```

```
<Text style={styles.content}>
    {this.state.quoteText}
</Text>
```

Image - React Native - Component Usage

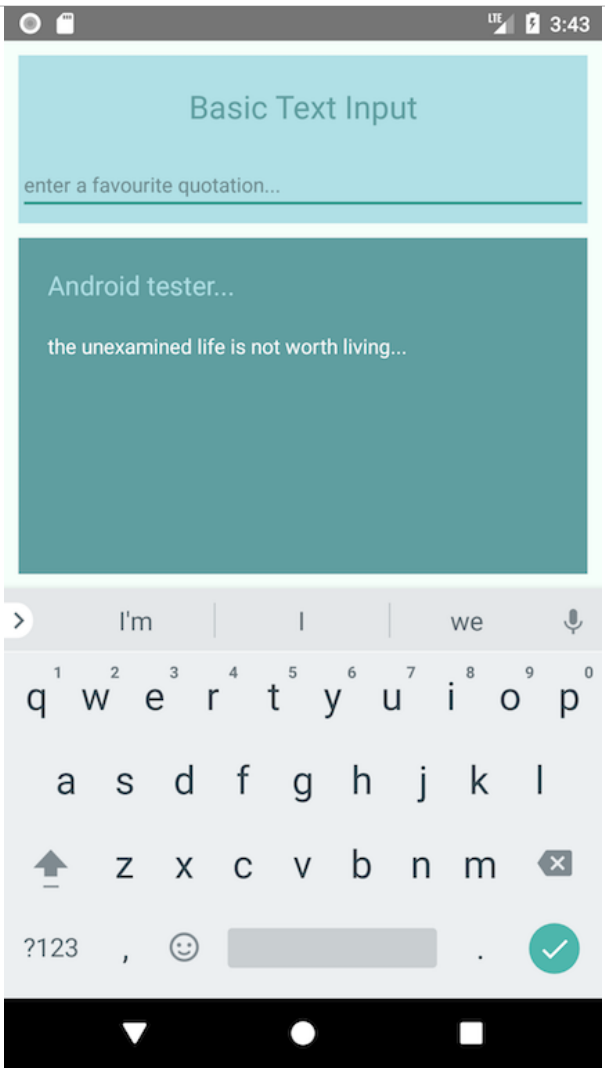
text input



React Native - text input

Image - React Native - Component Usage

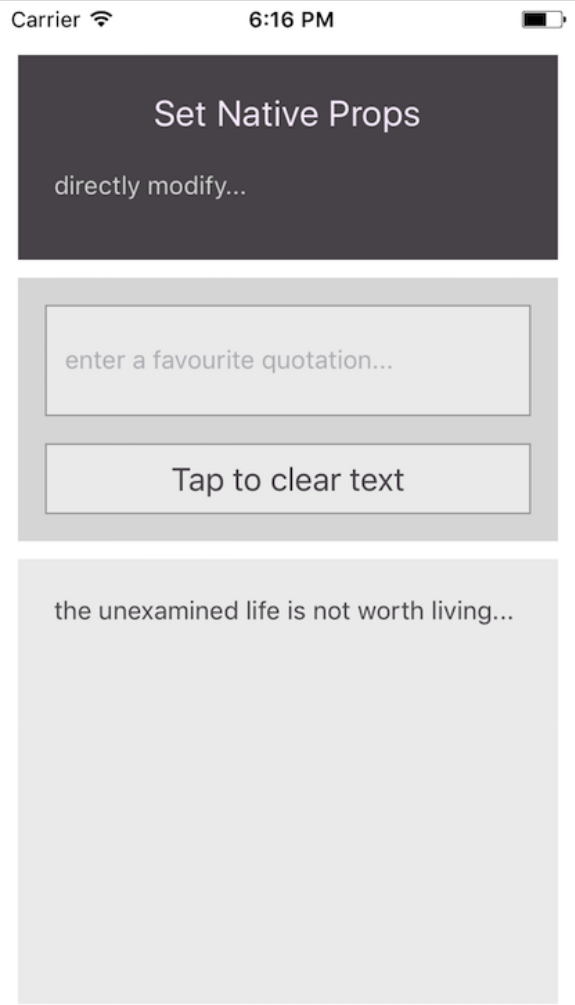
text input



React Native - text input

Image - React Native - Component Usage

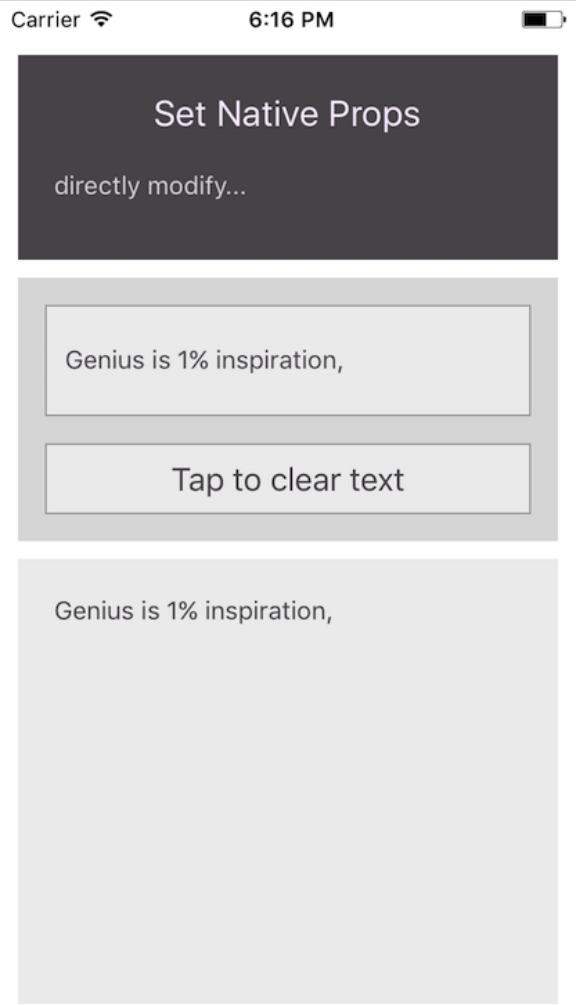
text input - use setNativeProps



React Native - text input

Image - React Native - Component Usage

text input - use setNativeProps



React Native - text input

React Native - component usage

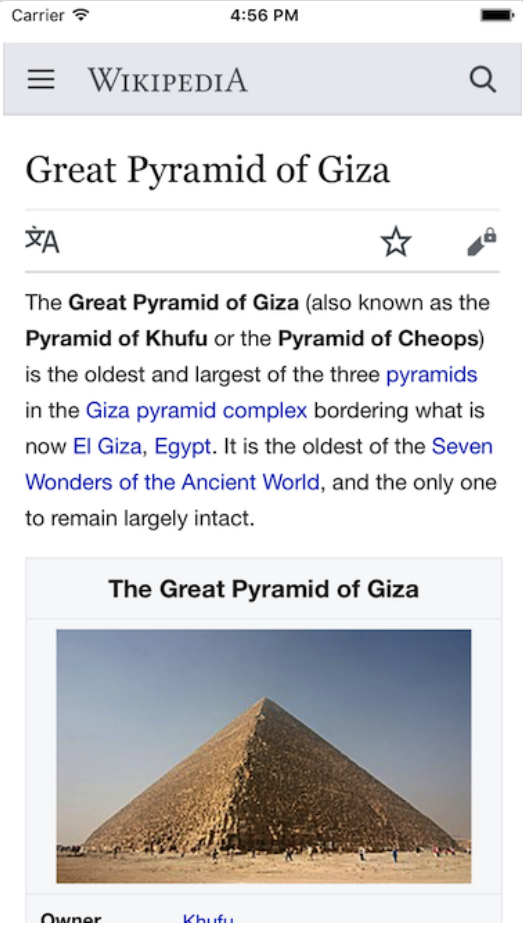
embed web content

- React Native offers a component solution for embedding web content
  - *embedded directly in a WebView*
  - *as a child to an existing view &c.*
- similar functionality to native WebView modules
- `WebView` component provides developers with a variety of props
  - *to help manipulate and structure a rendered web page*
- also use various available callbacks
  - *provide an option to register to specific events*
  - *e.g. error handling, message responses, navigation state change...*

```
<WebView style={styles.web}
  scalesPageToFit
  automaticallyAdjustContentInsets
  source={{
  uri: 'https://en.wikipedia.org/wiki/Great_Pyramid_of_Giza'
  }} />
```

Image - React Native - Component Usage

WebView - load external page &c.



React Native - WebView

React Native - component usage

iOS - segmentedControlIOS

- some components in React Native may be specific to a given mobile OS
  - *e.g. Segmented Control component is specific to iOS*
- offers a simple split option to switch between two groupings of content
- e.g. we might use this component as follows

```
<SegmentedControlIOS
  values={['Giza', 'Luxor']}
  selectedIndex={this.state.selectedIndex}
  onChange={(event) => {
    this.setState({selectedIndex: event.nativeEvent.selectedSegmentIndex});
  }}
/>
```

- instead of passing expected `onValueChange` props
  - *we can pass a callback prop for `onChange`*
- prop will receive an event argument
  - *e.g. from `nativeEvent` as shown in this example*
- also abstract this usage to pass in required values for each segment

React Native - navigation

intro to navigator

- React Native was initially released in 2015
  - *it came with a default navigator component to help structure internal navigation*
  - *structured stack control and management*
- community development and usage has moved towards various open project
- a popular option is the package **react-navigation**
  - *available from NPM*
- basic navigator components are stack-based
  - *similar to OnsenUI, jQuery Mobile navigation &c.*
- such components use a standard screen stack for navigating through an application
- as a user navigates to a new screen
  - *the navigator will push it onto the stack*
- as they navigate back
  - *a view &c. will simply be popped from the stack*

React Native - navigation

basic usage - part I

- create a new app with React Native,

```
react-native init BasicAppNavigation
```

- then install `react-navigation` community package

```
yarn add react-navigation
```

or

```
npm i react-navigation --save
```

React Native - navigation

basic usage - part 2

- React Navigation designed to meet many different navigation requirements
  - *it uses a concept of different Navigators to setup apps*
- start by importing package into `App.js`

```
import { StackNavigator } from 'react-navigation';
```

- then set the required file for our configuration of the routing

```
import routes from './config/routes';
```

React Native - navigation

basic usage - part 3

- in the `config` folder of our `src` directory
  - *add a `routes.js` file to store details of screens and routes*

```
import Home from '../screens/Home';
import CardScreen from '../screens/CardScreen';

const routes = {
  home: { screen: Home },
  card: { screen: CardScreen }
}

export default routes;
```

- import equired screens and their content and structure
- use screens as part of the routes for the app's navigation
- export the routes for use within our app

React Native - navigation

basic usage - part 4

- output a dynamic title for each screen navigation
  - *define a static property, `navigationOptions`*
  - *add to class for each screen component*

```
static navigationOptions = {
  title: "Home Screen"
}
```

- might also set this as dynamic to accept a props for each navigation request

```
static navigationOptions = ({ navigation }) => ({
  title: `Chosen cards - ${navigation.state.params.cards}`
})
```

React Native - navigation

basic usage - part 5

- add a component, such as a button, to allow us to call the `navigate` function

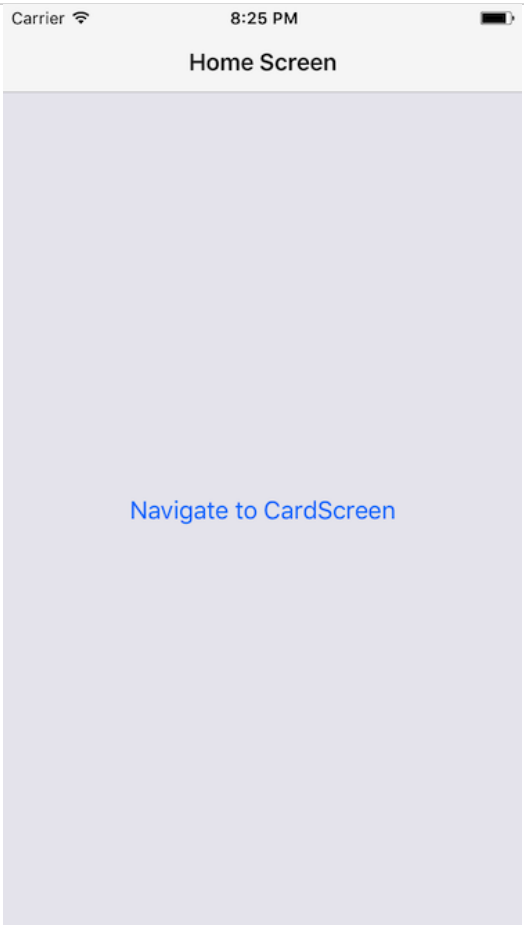
```
<Button onPress={() => this.props.navigation.navigate('card', { cards: 'Egypt' })}
  title="Navigate to CardScreen" />
```

- pass an argument for the required screen name
  - *defined in the config for the routes*
- we might pass a parameter for name of screen &c. to next screen
- e.g. accessed and used for title of screen

```
static navigationOptions = ({ navigation }) => ({
  title: `Chosen cards - ${navigation.state.params.cards}`
})
```

Image - React Native

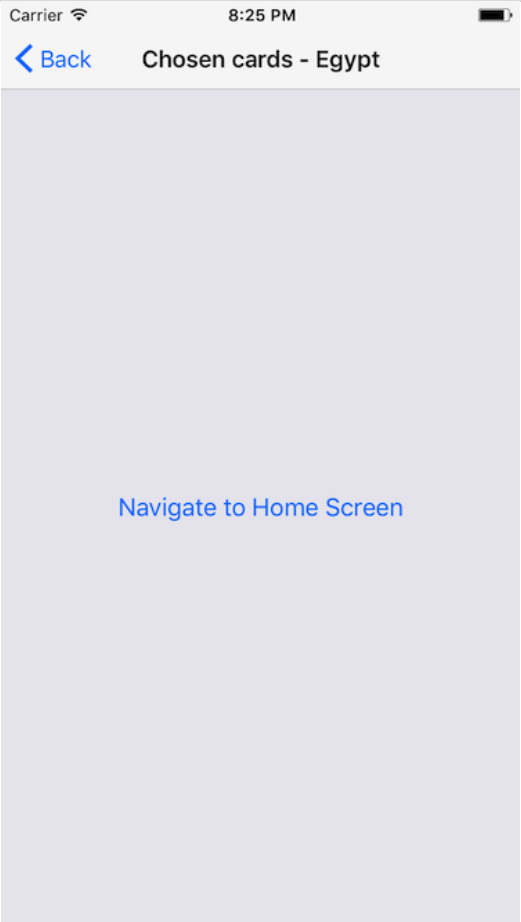
navigation - part 1



React Native - navigation

Image - React Native

navigation - part 2



React Native - navigation

React Native - Data

intro

- already seen data examples for Cordova
  - *including IndexedDB, Native Storage, various APIs...*
- React Native equally capable of accessing data stores
  - *a popular option for object based data storage is Firebase*
- useful to understand how React Native works
  - *with remote queries, fetching data, and authentication...*
- setup and add our own login and authentication for an app
- leverage an existing social provider
  - *e.g. Facebook, GitHub, Google, Microsoft, Twitter...*
- similar patterns and usage to web apps

React Native - Data - Firebase

NoSQL options

- other data store and management options now available to us as developers
- depending upon app requirements consider
  - *Firebase*
  - *RethinkDB*
- as a data store, Firebase offers a hosted NoSQL database
  - *data store is JSON-based*
  - *offering quick, easy development from webview to data store*
- syncs an app's data across multiple connected devices in milliseconds
  - *available for offline usage as well*
- provides an API for accessing these JSON data stores
  - *real-time for all connected users*
- Firebase as a hosted option more than just data stores and real-time API access
- Firebase has grown a lot over the last year
  - *many new features announced at Google I/O conference in May 2016*
  - *analytics, cloud-based messaging, app authentication*
  - *file storage, test options for Android*
  - *notifications, adverts...*

React Native - Data - Firebase

Firebase - intro

- React Native, of course, does not limit data stores or queries to just Firebase
- Firebase is hosted platform, aquired by Google
  - *provides options for data starage, authentication, real-time database querying...*
- *authentication* with Firebase provides various backend services and SDKs
  - *help developers manage authentication for an app*
  - *service supports many different providers, including Facebook, Google, Twitter &c.*
  - *using industry standard **OAuth 2.0** and **OpenID Connect** protocols*
- **Cloud Storage** used for uploading, storing, downloading files
  - *accessed by apps for file storage and usage...*
  - *features a useful safety check if and when a user's connection is broken or lost*
  - *files are usually stored in a Google Cloud Storage bucket*
  - *files accessible using either Firebase or Google Cloud*
  - *consider using Google Cloud platform for image filtering, processing, video editing...*
  - *modified files may then become available to Firebase again, and connected apps*
  - *e.g. Google's Cloud Platform*
- **Real-time Database** offers a hosted NoSQL data store
  - *ability to quickly and easily sync data*
  - *data synchronisation is active across multiple devices, in real-time*
  - *available as and when the data is updated in the cloud database*
- other services and tools available with Firebase
  - *analytics*
  - *advertising services such as adwords*
  - *crash reporting*
  - *notifications*
  - *various testing options...*

React Native - Data - Firebase

Firebase - basic setup

- start using Firebase by creating an account with the service
  - *using a standard Google account*
  - *Firebase*
- login to Firebase
  - *choose either Get Started material or navigate to Firebase console*
- at *Console* page, get started by creating a new project
  - *click on the option to* `Add project`
  - *enter the name of this new project*
  - *and select a region*
- then redirected to the *console dashboard* page for the new project
  - *access project settings, config, maintenance...*

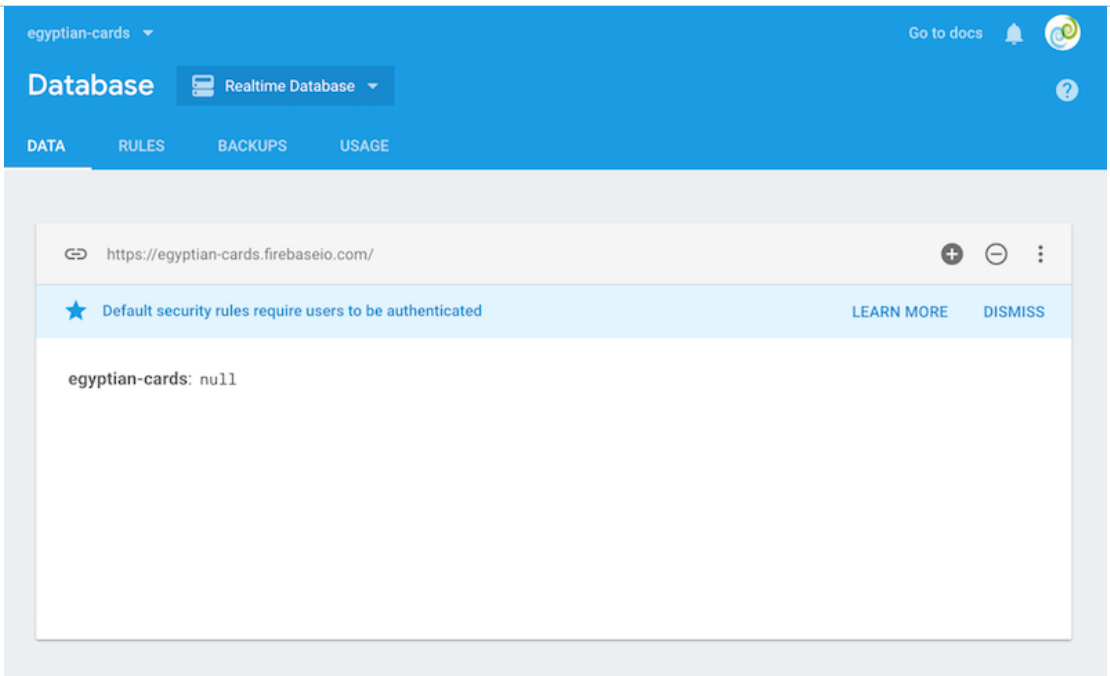
React Native - Data - Firebase

Firebase - create real-time database

- now setup a database with Firebase for a test React Native app
- start by selecting *Database* option from left sidebar on the Console Dashboard
  - *available under the DEVELOP option*
- then select *Get Started* for the real-time database
- presents an empty database with an appropriate name to match current project
- data will be stored in a JSON format in the real-time database
- working with Firebase is usually simple and straightforward for most apps
- get started quickly direct from the Firebase console
  - *or import some existing JSON...*

Image - Firebase

create a database



Firebase - create a database

React Native - Data - Firebase

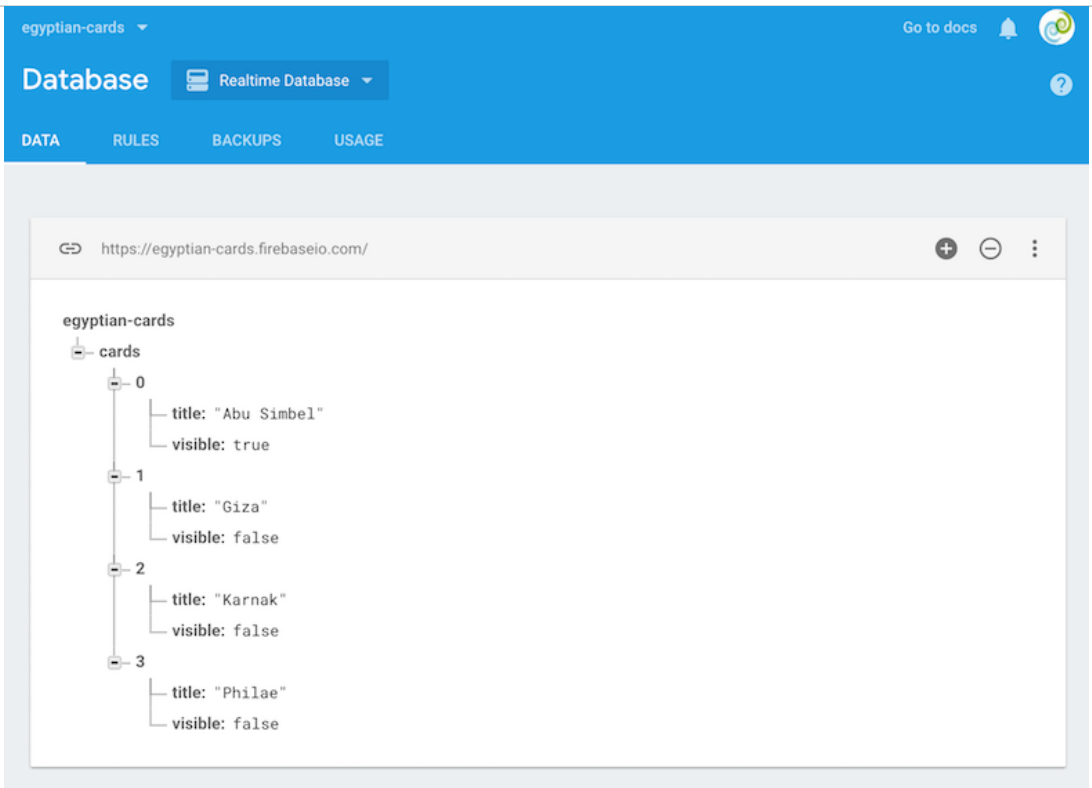
Firebase - import JSON data

- start with some simple data to help testing Firebase with a React Native app
- import JSON into our test database
  - *then query the data &c. from the app*

```
{
  "cards": [
    {
      "visible": true,
      "title": "Abu Simbel",
      "card": "temple complex built by Ramesses II"
    },
    {
      "visible": false,
      "title": "Amarna",
      "card": "capital city built by Akhenaten"
    },
    {
      "visible": false,
      "title": "Giza",
      "card": "Khufu's pyramid on the Giza plateau outside Cairo"
    },
    {
      "visible": false,
      "title": "Philae",
      "card": "temple complex built during the Ptolemaic period"
    }
  ]
}
```

Image - Firebase

JSON import



Firebase - import JSON file

React Native - Data - Firebase

Firebase - permissions

- initial notification in Firebase console after creating a new database
  - *Default security rules require users to be authenticated*
- permissions with Firebase database
  - *select RULES tab for current database*
- lots of options for database rules
  - *Firebase - database rules*
- e.g. for testing initial React Native we might remove authentication rules
- change rules as follows

from

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

to

```
{
  "rules": {
    ".read": "true",
    ".write": "true"
  }
}
```

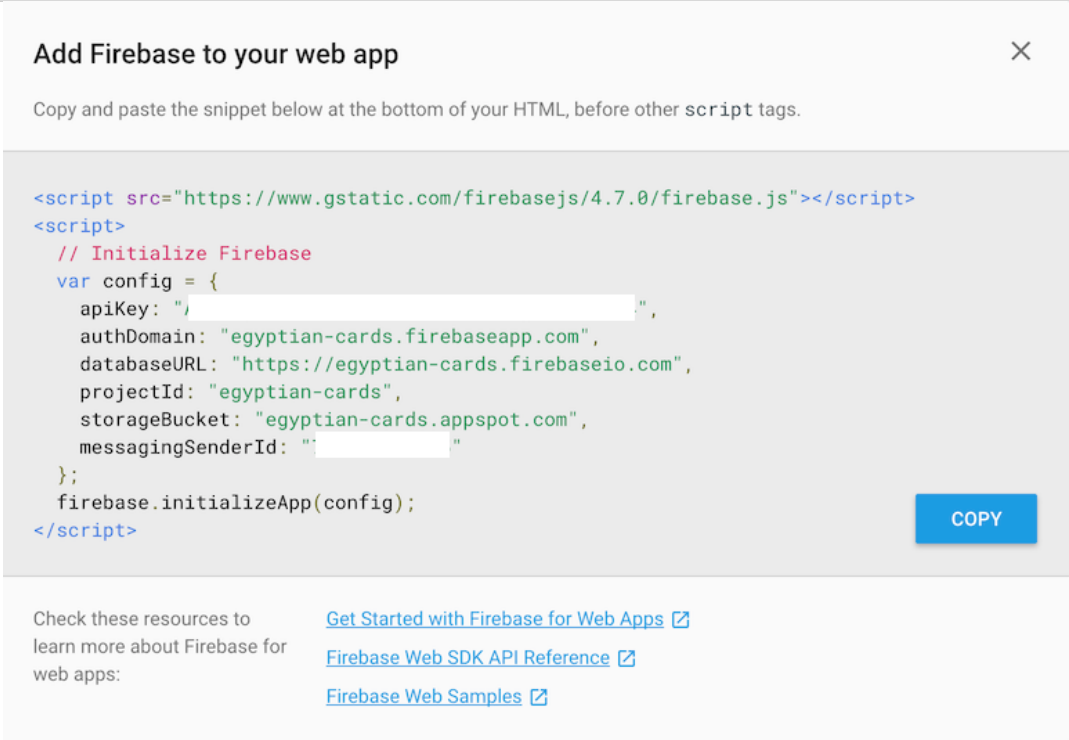
React Native - Data - Firebase

add Firebase to React Native - part 1

- we can now test our new Firebase database with a React Native app
- need to start by getting some useful information from Firebase
  - *select the Project Overview link in the left sidebar*
  - *then click on the icon for Add Firebase to your web app*
- we can take advantage of the provided JavaScript SDK with React Native
- Firebase console will show us a modal with initialisation settings
  - *config settings for adding Firebase usage to our app*

Image - Firebase

initialisation config settings



Firebase - config settings

React Native - Data - Firebase

add Firebase to React Native - part 2

- start by copying these config values for use with our React Native app
- Firebase runs on a JavaScript thread
  - *certain complex applications, e.g. detailed animations &c.*
  - *may be adversely affected by this structure...*
- might consider using a community package called `react-native-Firebase`
  - *package acts as a wrapper around the Firebase SDK for Android and iOS*
  - *React Native Firebase*
- for most React Native apps we simply integrate Firebase JavaScript SDK
  - *install using NPM or Yarn*

```
npm install firebase --save
```

or

```
yarn add firebase
```

React Native - Data - Firebase

add Firebase to React Native - part 3

- after installing Firebase support for our app
  - *add a new file, `firebase.js`, to a `services` folder in the `src` directory*
- `firebase.js` - specify an initialisation function for working with Firebase services
- working with the initialisation config data provided by Firebase
  - *for the JavaScript SDK for our app*
- need to import the firebase module
  - *then setup a function to handle the initialisation config*

```
import * as firebase from "firebase";

export const initialize = () => firebase.initializeApp({
    apiKey: "__your-api-key__",
    authDomain: "egyptian-cards.firebaseapp.com",
    databaseURL: "https://egyptian-cards.firebaseio.com",
    projectId: "egyptian-cards",
    storageBucket: "egyptian-cards.appspot.com",
    messagingSenderId: "__your-sender-id__"
})
```

React Native - Data - Firebase

add Firebase to React Native - part 4

- need to export the `initialize` function from `firebase.js`
  - *use in a central config file for API usage*
- create a new file for API config management in the `src/services` directory
- config file helps manage multiple services and APIs within a project's structure
- import the `initialize` function for Firebase

```
import { initialize } from './firebase';
```

- then export the functionality for Firebase

```
export const initApi = () => initialize();
```

React Native - Data - Firebase

add Firebase to React Native - part 5

- need to setup Firebase usage in our application root, `App.js`
- use the `componentWillMount` lifecycle hook to call the `initApi()` function
- ensure Firebase is ready and available for our app

```
export default class extends Component {
    componentWillMount() {
        initApi();
    }

    render() {
        return (
          ...
        )
    }
}
```

React Native - Data - Firebase

add Firebase to React Native - part 6

- after setup and initialisation, we can start to consider working with our Firebase database
- benefits of Firebase is that the SDK allows our apps and database to be in sync
  - *as and when updates are registered*
- we need to setup database listeners to ensure the state of our app is updated
  - *whenever a database is modified on Firebase...*
- add such listeners to our `firebase.js` file

```
// setup listener for firebase updates
export const setListener = (endpoint, updaterFn) => {
    firebase.database().ref(endpoint).on('value', updaterFn);
    return () => firebase.database().ref(endpoint).off();
}
```

- using this function to perform two key tasks
- after passing arguments for `endpoint` and `updateFn`
  - *get reference to endpoint for our Firebase database*

```
firebase.database().ref(endpoint)
```

- we can send other required endpoints for our app and Firebase database
  - *such as `cards` in our current example*
- then call the `on()` function allowing us to pass `udpaterFn`
  - *passed as we call the `setListener` function in our app*
- then return a function to allow us to remove the attached listener later in our app

React Native - Data - Firebase

add Firebase to React Native - part 7

- start to use such listeners and functionality in our app
- create a `getCards()` function in `api.js` file
  - *use the `setListener` we created in `firebase.js`*

```
// get cards from current firebase database
export const getCards = (updaterFn) => setListener('cards', updaterFn);
```

- then import this function for a given screen in our app, such as the Card screen,

```
import { getCards } from '../services/api';
```

- then set our state to use this function, and the cards from the database

```
componentDidMount() {
  this.unsubscribeGetCards = getCards((snapshot) => {
    this.setState({
      messages: Object.values(snapshot.val())
    })
  })
}
```

React Native - Data - Firebase

add Firebase to React Native - part 8

- in `componentDidMount()` lifecycle hook
  - *use `Object.values` on Firebase `snapshot.val()`*
  - *`FlatList` component we're using for rendering expects an array*
  - *Firebase returns an object for the values*
- `getCards` is calling `setListener`
  - *returns a function for a remove listener*

```
firebase.database().ref(endpoint).off();
```

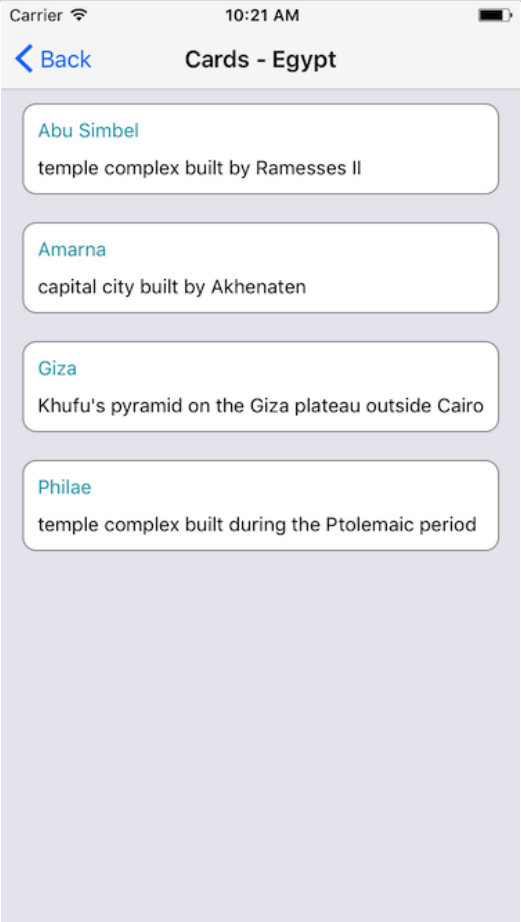
- set the result for `getCards` to `this.unsubscribeGetCards`
- then later call it as necessary in the lifecycle hook for `componentWillUnmount`
- might also add a single call, instead of constantly checking for updates

```
firebase.database().ref(endpoint).once('value')
```

- returns a *promise*
  - *we can use in a standard manner, or chain with `then()`...*

Image - Firebase

render data from database



Firebase - render data

React Native - fetching data

HTML5 Fetch API - intro

- React Native also provides support for the developing HTML5 Fetch API
- also use other JS libraries such as *axios* or standard *XMLHttpRequest*
  - *no CORS (cross-origin resource sharing) issues with React Native*
- use for network based queries, API requests, and so on...
- start with a simple query structure with fetch

```
fetch('https://your-server/api/getnotes.json')
```

- Fetch API with return a promise
  - *we can then chain to* `then()`
  - *or perhaps use with async or await using ES6 JavaScript*
- might also add a second paremeter to this fetch query

```
fetch('https://your-server/api/getnotes.json', {
  method: 'POST',
  headers: {
    ...
  },
  body: JSON.stringify({
    ...
  })
})
```

React Native - fetching data

HTML5 Fetch API - working with the data

- response from a Fetch request will return a *Blob*
- response contains metadata
- access return data using a promise chain &c.

```
fetch('https://your-server/api/getnotes.json')
  .then(result => result.json())
  .then(yourData => this.setState({
    yourData
    })
  )
  .catch(error => {
    console.error(error);
  });
```

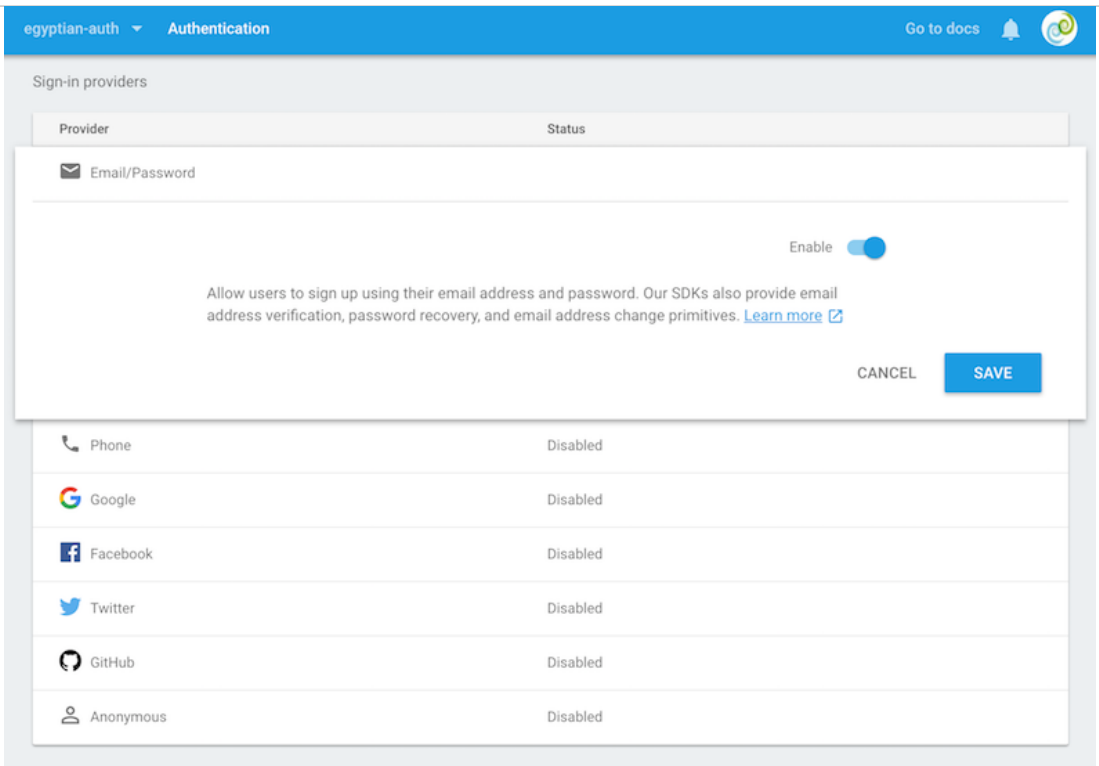
React Native - Authentication

Firebase - setup authentication

- part of using authentication with Firebase
  - *need to explicitly configure this option in the Console Dashboard*
- need to setup the sign-in method for a particular database
- select various options and providers, including
  - *email and password*
  - *phone*
  - *Google*
  - *Facebook*
  - *Twitter*
  - *GitHub*
  - *and Anonymous*

Image - Firebase

authentication options



Firebase - auth options

React JavaScript Library

Additional reading, material, and samples

- design thoughts
- event handling
- more composing components
- DOM manipulation
- forms
- intro to flux
- animations
- lots of samples...

References

- Axios JS library
- Firebase
- Firebase - database rules
- Google's Cloud Platform
- MDN - super
- React
- React Native
- React DevTools
- React Navigation
- React Native - Layout Props
- React Native - StatusBar
- XMLHttpRequest
- Yarn - Firebase