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

Fall Semester 2017 - Week 11 - React & React Native

Dr Nick Hayward

React JavaScript Library

overview

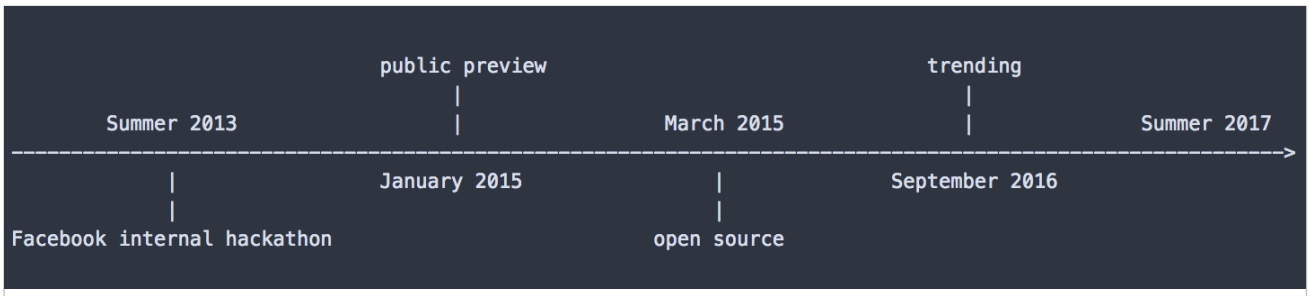
- **React** began life as a port of a custom PHP framework called XHP
  - *developed internally at Facebook*
- XHP, as a PHP framework, was designed to render the full page for each request
- **React** developed from this concept
  - *creating a client-side implementation of loading the full page*
- **React** can, therefore, be perceived as a type of *state machine*
  - *control and manage inherent complexity of state as it changes over time*
- able to achieve this by concentrating on a narrow scope for development,
  - *maintaining and updating the DOM*
  - *responding to events*
- **React** is best perceived as a *view* library
  - *no definite requirements or restrictions on storage, data structure, routing...*
- allows developers freedom
  - *incorporate **React** code into a broad scope of applications and frameworks*

React Native

overview

- familiar to React developers
- React Native offers a native mobile experience
  - *using React JS patterns and structures*
- developers can create native components for Android and iOS
- basics of React development are still required for React Native development, e.g.
  - *components*
  - *JSX*
  - *props*
  - *state*
  - ...
- create modular components with JavaScript
  - *without associated HTML and CSS*

Image - React Native Timeline



Timeline of React Native...

- React Native

React Native

native concept

- enables the transformation of JavaScript to required native modules,
  - *i.e. for Android and iOS.*
- as we compile a React Native app, we are now dealing with a native app
  - *a performant, natively compiled app*
- performance may become identical to those developed using the native SDK
  - *i.e. Java or Kotlin for Android*
  - *Objective-C and Swift for iOS*
- another benefit of working with React Native
  - *its ability to wrap many core APIs for iOS and Android*
- React Native provides an API as a simple bridge to its own modules
- possible to integrate React Native into an existing native mobile application

React JavaScript Library

why use React?

- React is often considered the *V* in the traditional *MVC*
- [React(http://facebook.github.io/react/docs/why-react.html) was designed to solve one problem

> *building large applications with data that changes over time*

- React can best be considered as addressing the core concerns
  - *simple, declarative, components*
- simple - define how your app should look at any given point in time
  - *React handles all UI changes and updates in response to data changes*
- declarative - as data changes, React effectively refreshes your app
  - *sufficiently aware to only update those parts that have changed*
- components - fundamental principle of React is building re-usable components
  - *components are encapsulated in their design and concepts*
  - *they make it simple for code re-use, testing...*
  - *in particular, the separation of design and app concerns in general*
- React leverages its built-in, powerful rendering system to produce
  - *quick, responsive rendering of DOM in response to received state changes*
- uses a virtual DOM
  - *enables React to maintain and update the DOM without the lag of reading it as well*

React Native

why use React Native?

- React introduced many interesting and exciting options for developing UIs
- React Native adopts many of these concepts to help ease the development of mobile applications, e.g.
  - *improved state management*
  - *uni-directions data flow*
  - *component based UI design and construction*
  - *associated ease of inheritance and abstraction*
  - ...
- React Native = code in JavaScript, and then compile to full native code
- JavaScript logic of app becomes native code for respective mobile OS
- quick and easy developer tools
  - *e.g. live reloading of app during development*
  - *hot loading of modules*
  - *developer tools for interactions and mapping*
  - ...

state changes

- as **React** is informed of a state change, it re-runs render functions
- enables it to determine a new representation of the page in its virtual DOM
- then automatically translated into the necessary changes for the new DOM
  - *reflected in the new rendering of the view*
- may, at first glance, appear inherently slow
  - *React uses an efficient algorithm*
  - *checks and determines differences*
  - *differences between current page in the virtual DOM and the new virtual one*
- from these differences it makes the minimal set of necessary updates to the rendered DOM
- creates speed benefits and gains
  - *minimises usual reflows and DOM manipulations*
- also minimises effect of cascading updates caused by frequent DOM changes and updates

component lifecycle

- in the lifecycle of a component
  - *its props or state might change along with any accompanying DOM representation*
- in effect, a component is a known state machine
  - *it will always return the same output for a given input*
- following this logic, React provides components with certain *lifecycle* hooks
  - *instantiation - mounting*
  - *lifetime - updating*
  - *teardown - unmounting*
- we may consider these hooks
  - *first through the instantiation of the component*
  - *then its active lifetime*
  - *finally its teardown*

component lifecycle - intro

- React components include a minimal lifecycle API
- provides the developer with enough without being overwhelming
  - *at least in theory*
- React provides what are known as *will* and *did* methods
  - *will - called right before something happens*
  - *did - called right after something happens*
- relative to the lifecycle, we can consider the following groupings of methods
  - *Instantiation (mounting)*
  - *Lifetime (updating)*
  - *Teardown (unmounting)*
  - *Anti-pattern (calculated values)*

component lifecycle - method groupings - Instantiation (mounting)

- includes methods called upon instantiation for the selected component class
- eg: `getDefaultProps` or `getInitialState`
  - *use such methods to set default values for new instances*
  - *initialise a custom state of each instance...*
- also have the important `render` method
  - *builds our application's virtual DOM*
  - *the only required method for a component*
- `render` method has rules it needs to follow
  - *such as accessible data*
  - *return values*
- `render` method must also remain *pure*
  - *cannot change the state or modify the DOM output*
  - *returned result is the virtual DOM*
  - *compared against actual DOM*
  - *helps determine if changes are required for the application*

component lifecycle - method groupings - Lifetime (updating)

- component has now been rendered to the user for viewing and interaction
- as a user interacts with the component
  - *they are changing the state of that component or application*
  - *allows us as developers to act on the relevant points in the component tree*
- State changes for the application
  - *those affecting the component*
  - *may result in update methods being called*
- we're telling the component how and when to update

component lifecycle - method groupings - Teardown (unmounting)

- as React is finished with a component
  - *it must be unmounted from the DOM and destroyed*
- there is a single hook for this moment
  - *provides opportunity to perform necessary cleanup and teardown*
- `componentWillUnmount`
  - *removes component from component hierarchy*
  - *this method cleans up the application before component removal*
  - *undo custom work performed during component's instantiation*

component lifecycle - method groupings - Anti-pattern (calculated values)

- React is particularly concerned with maintaining a single source of truth
- one point where `props` and `state` are derived, set...
- consider calculated values derived from `props`
  - *considered an anti-pattern to store these calculated values as state*
- if we needed to convert a `props` date to a string for rendering
  - *this is not state*
  - *it should simply be calculated at the time of `render`*

React JavaScript Library

a few benefits

- one of the main benefits of this virtual approach
  - *avoidance of micro-managing any updates to the DOM*
- a developer simply informs React of any changes
  - *such as user input*
- React is able to process those passed changes and updates
- React has inherent benefit of delegating all events to a single event handler
  - *naturally gives React an associated performance boost*

first app - basic-app

- basic app for React Native will follow a known, prescribed pattern
- use React Native CLI tool to generate a shell app for developing an app
- in a development directory, e.g. `/Development/react-native/`
  - *issue the following command to generate project files for an app*

```
react-native init BasicApp
```

- command will call the React Native CLI
  - *then initialises a new project named* `BasicApp`
  - *installed to a directory named* `BasicApp` *in CWD*
- command also outputs useful instructions for running an app on iOS and Android

React Native

how to start an app - iOS on OS X

- CWD to React Native app
- issue the following command in the terminal, e.g.

```
react-native run-ios
```

- command will build the project
- launch the iOS simulator
- then show the app in a simulator window

how to start an app - Android on OS X

- assuming Android has been setup and configured correctly
- running an app with Android follows the same pattern as iOS, e.g.

```
react-native run-android
```

- initial run will scan local machine for *symlinks*
- starts JS server for development and testing
- then it will need to download and config Gradle for local Android setup
- it starts to build and install the app in the CWD

React Native

`basic app` - intro

- now start to develop a basic app with React Native
- might add a basic screen, show a list of items from JSON, and render some images
- consider how the fundamental structures and patterns work in React Native

app - basic app directory structure

- basic structure is as follows,

```
|-- BasicApp
    |__ __tests__
    |__ android
    |__ ios
    |__ node_modules
    |__ App.js
    |__ app.json
    |__ index.js
    |__ package-lock.json
    |__ package.json
    |__ ...
```

- main directories and files created as we initialise a new project
- necessary files to build an app with React Native for iOS and Android
  - *located in their respective directories, `iOS` and `Android`*
  - *these are native project directories*
  - *can be imported as native apps into Android Studio and Xcode*
- **n.b.** not necessary to modify these files for majority of apps
- `app.json` file includes brief metadata for a generated app
  - *e.g. name, display name, and so on...*
- `package.json` file is a standard file for Node development
  - *contains metadata for the React Native app...*

React Native

app - getting started - part I

- clear the biolerplate code from the `App.js` file
- add a basic component for a home screen message, e.g.

```
// import React, Component module as Component from base React
import React, { Component } from 'react';
// import Text as Text from React Native
import { Text } from 'react-native';

// default export - BasicApp - used when no explicit import reference...
export default class BasicApp extends Component {
  render() {
    return (
      <Text>Greetings, Human!</Text>
    );
  }
}
```

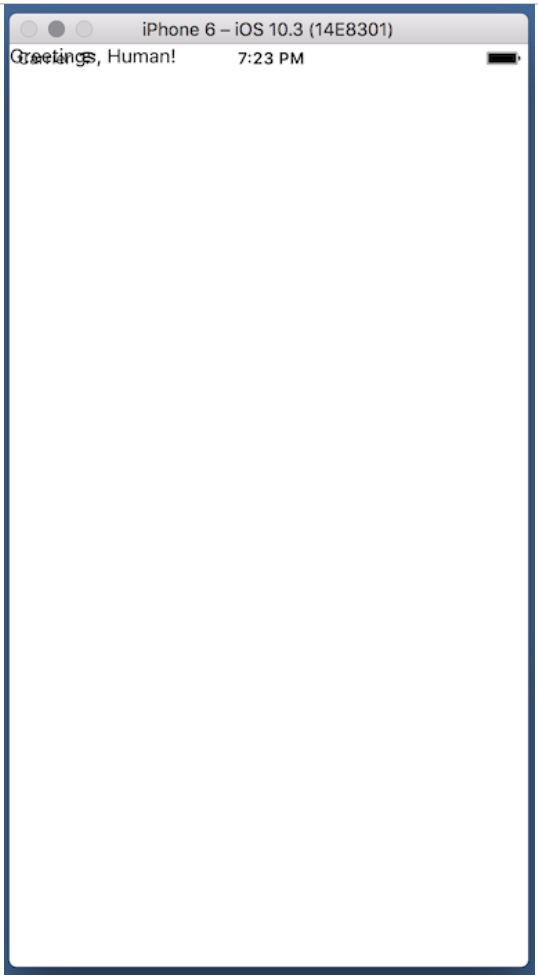
app - getting started - part 2

- use this new component within our app
- register it in the default `index.js` file, e.g.

```
// import AppRegistry as AppRegistry
import { AppRegistry } from 'react-native';
// import App from App.js (.js implied...)
import App from './App';

// register new component as Basic App - pass default from App.js
AppRegistry.registerComponent('BasicApp', () => App);
```

Image - React Native - Basic App

first example



BasicApp in iOS Simulator

React Native - Props

intro

- `props` in React and React Native are parameters
  - *we may pass them as a component is created...*
- such props enable most components to be customised as they're created
- use `props` to pass variables within a component &c.
- often use `props` to pass values and variables between components
- in custom components usage of `props` helps abstract component structure
  - *helps reuse within an app...*

React Native - Props

`props` usage - part I

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

// custom abstracted component - expects props for text `output`
class OutputText extends Component {
  render() {
    return (
      // render passed props `output` value
      <Text>{this.props.output}</Text>
    );
  }
}

// default component - use View container render OutputText message with passed props...
export default class WelcomeMessage extends Component {
  render() {
    return (
      // View container - render Text output from OutputText component
      <View style={{alignItems: 'center'}}>
        // JSX embed OutputText component - pass value for props `output`
        <OutputText output='welcome to the basic tester...' />
      </View>
    );
  }
}
```

React Native - Props

`props` usage - part 2

- we define the required imports for React and React Native
  - *including existing components we need for this basic app*
- `AppRegistry` - entry point for JavaScript to enable a React Native app to run...
  - *added as part of `init` command for React Native apps*
- `Text` - used to display text within an app
- `View` - a UI container for displaying content
  - *basic requirement for UI development with React Native*
  - *supports layout structures with flexbox, style, touch, accessibility...*
- then define our required custom components
  - *one abstracted for broader re-use*
  - *the other for use in the current specific app*
- `OutputText` is the abstracted component
  - *accepts `props` as part of the output for a standard `Text` component*
- as `render()` function is called for this component
  - *it returns text output with the value of the passed props*
- `WelcomeMessage` is a custom component
  - *also set as the default export for the module*
- if the export is not explicitly set
  - *`WelcomeMessage` component will be called at execution*
  - *this component returns a standard `View` container*
  - *with its own defined `style` props*

React Native - Layout and Styles

flex and CSS inspired

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

React Native - Layout and Styles - add some flex

intro

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

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

React Native - Layout and Styles - add some flex

`flexDirection`

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

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

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

React Native - Layout and Styles - add some flex

`justifyContent`

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

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

React Native - Layout and Styles - add some flex

`alignItems`

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

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

more layout options

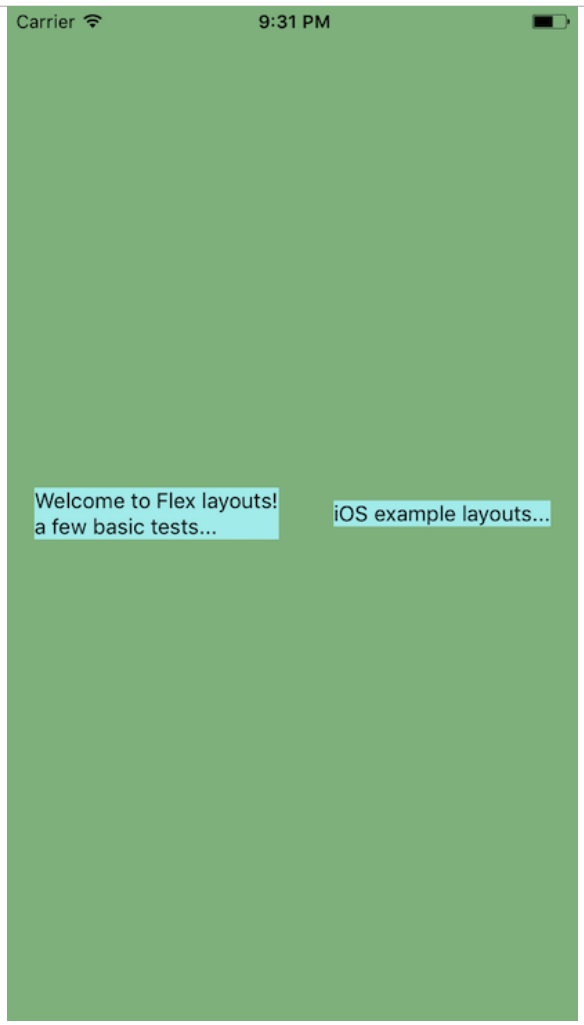
- further options may be specified as props
  - *add to a given component or stylesheet...*
- full details can be found at the following URL,
  - *Layout Props*

React Native - Layout and Styles - add some flex

basic flex usage - part I

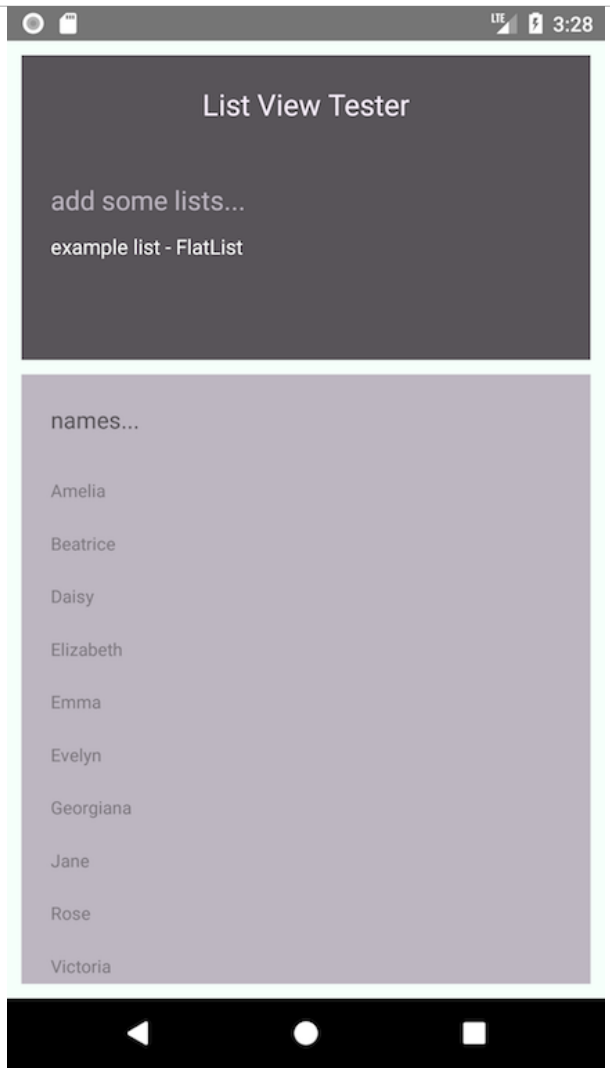
```
...
export default class BasicFlexApp extends Component {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.col}>
          <Text>
            Welcome to Flex layouts!
          </Text>
          <Text>
            a few basic tests...
          </Text>
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

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

Image - React Native - Flex Basics



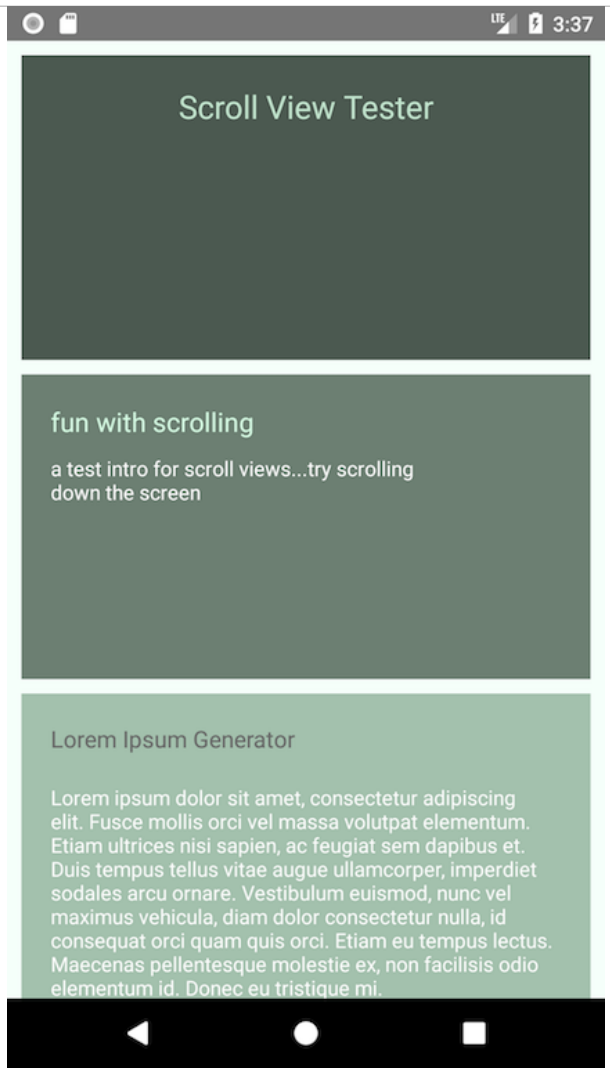
React Native Flex Basics

Image - React Native - Flex Basics - List View



React Native List View

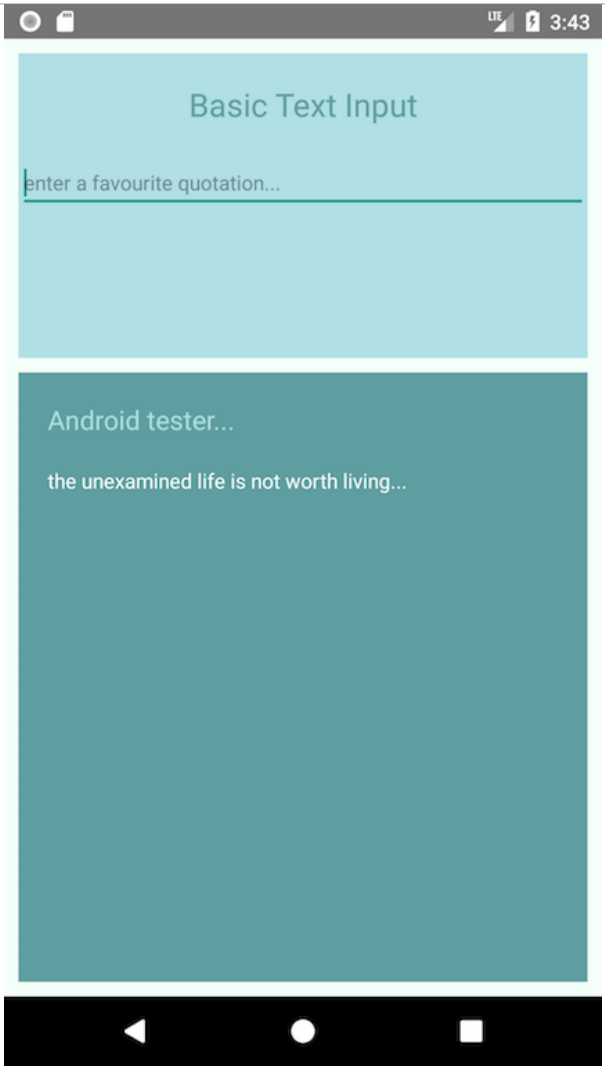
Image - React Native - Flex Basics - Scroll View



React Native Scroll View

Image - React Native - Styles

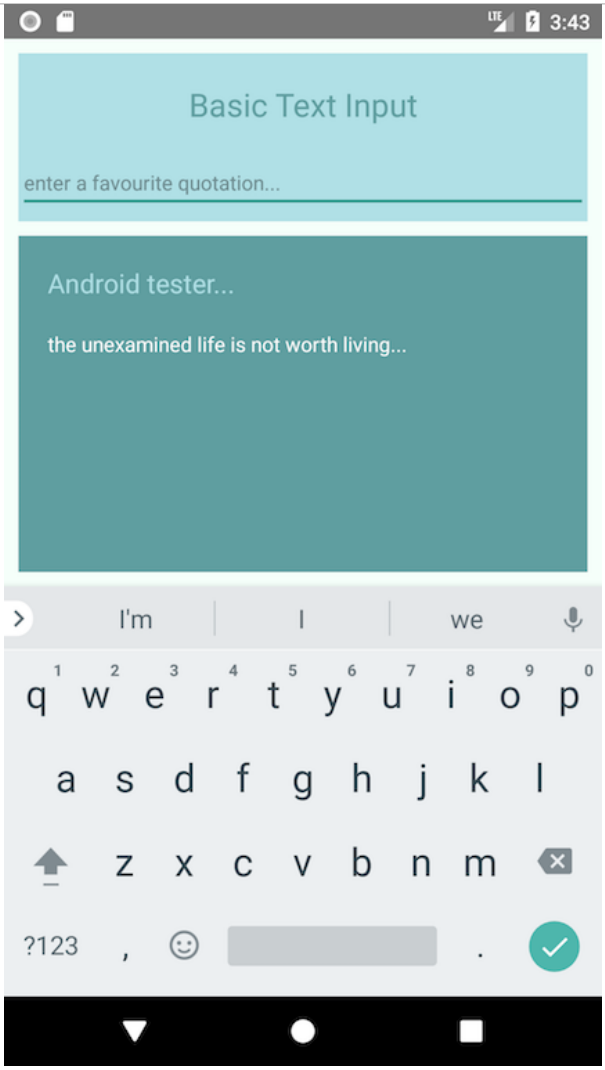
text input



React Native Styles - Text Input

Image - React Native - Styles

text input with keyboard



React Native Styles - Text Input

React Native - Layout and Styles

basic styling

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

```
fontWeight
fontSize
backgroundColor
```

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

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

React Native - Layout and Styles

`style` usage

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

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

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

React Native - Layout and Styles

Platform specific styles

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

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

React Native - Layout and Styles

Style inheritance - part 1

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

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

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

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

Style inheritance - part 2

- usage may then be as follows,

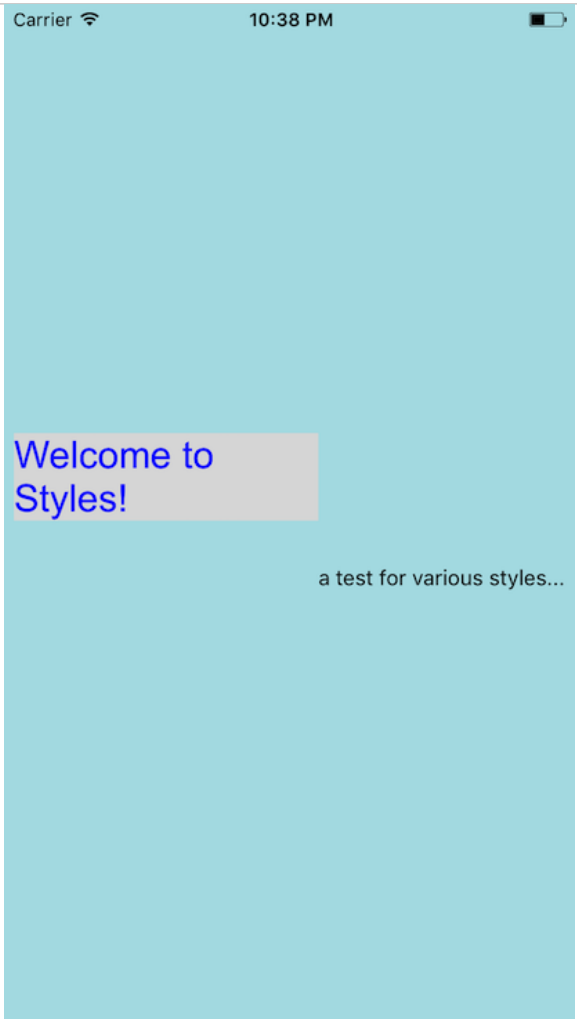
```
<MyAppText style={styles.textdefault}>
  some app text...
  <Text style={styles.welcome}>Welcome to Styles!</Text>
</MyAppText>
```

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

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

Image - React Native - Styles

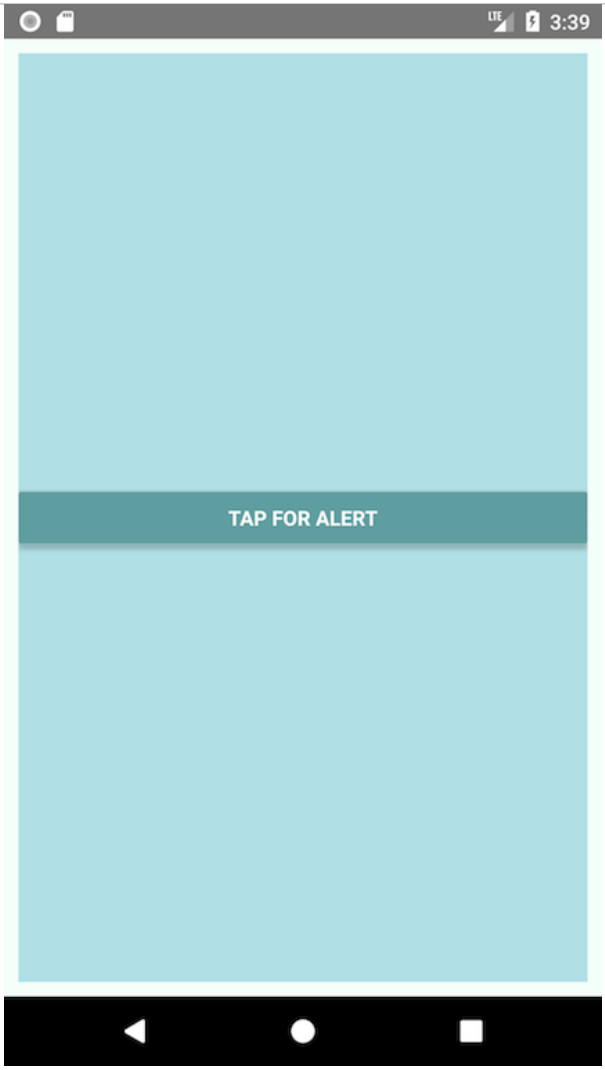
basic styles



React Native Styles - Inherit

Image - React Native - Styles

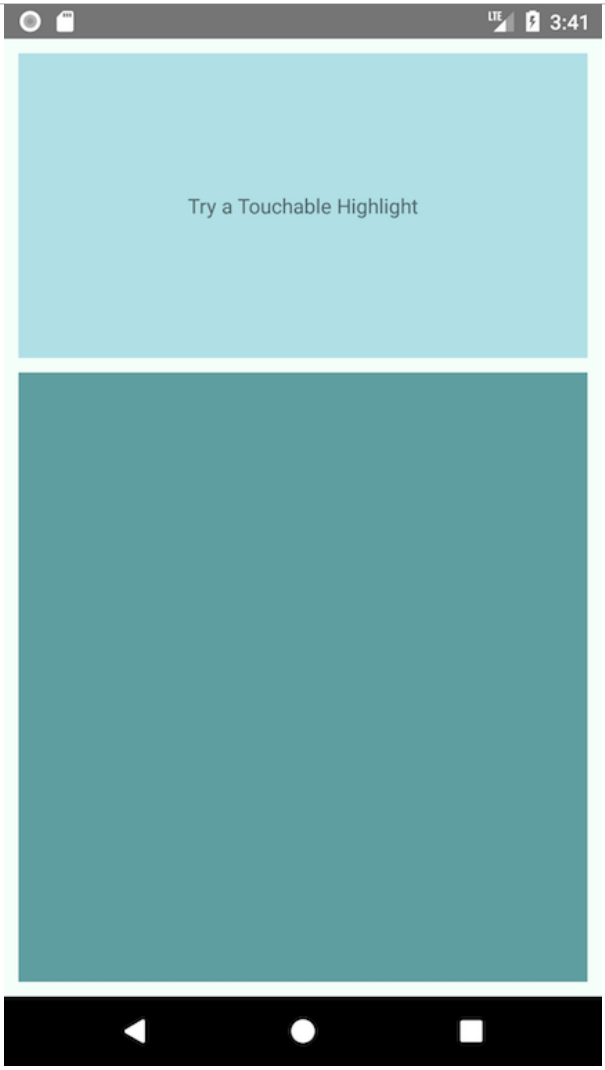
basic buttons



React Native Styles - Buttons

Image - React Native - Styles

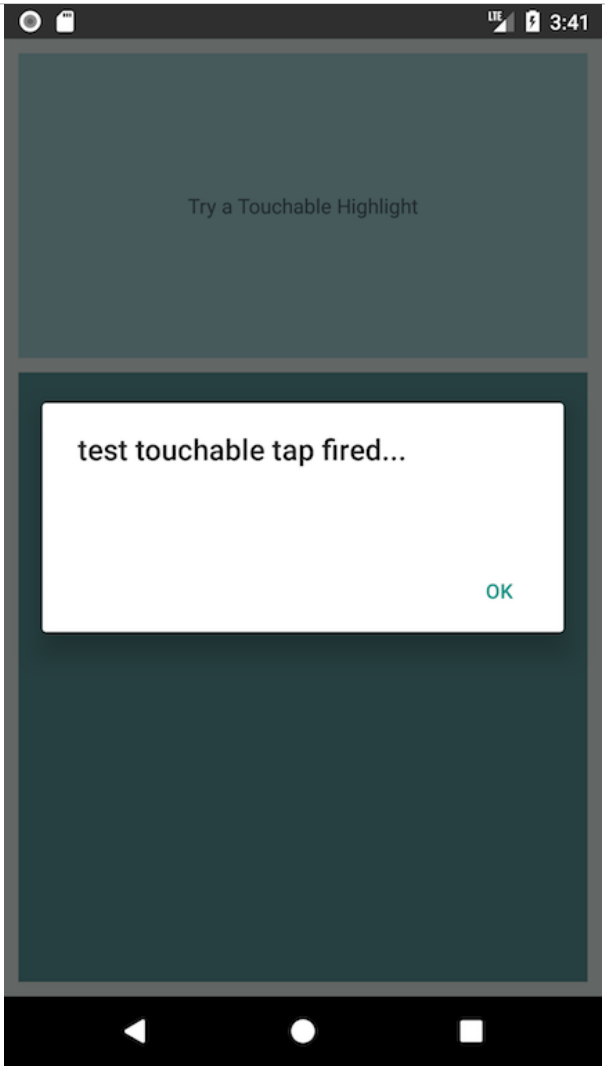
basic touchable



React Native Styles - Touchable

Image - React Native - Styles

basic touchable with alert



React Native Styles - Touchable

React Native - State

intro

- React and React Native manage data using either `props` or `state`
- `props` are set by the parent, and remain immutable for a component's lifetime
- if we need to modify data whilst an app is running, we can use `state`
- React has a distinct pattern to `state` usage
  - *`state` should be initialised in the constructor for a component &c.*
  - *`setState` may then be used to modify and update `state`*

React Native - State

general usage

- use `state` to manage data within an app
  - *from basic UI updates to data from a remote DB or API*
- as the data is updated
  - *we can modify `state` within our app*
- `state` may be managed within a React Native app
  - *or by using containers such as Redux, MobX...*
- *Redux* and *MobX* are predominantly used with React based apps
  - *standalone libraries for state management*
- by introducing a container such as *Redux*
  - *circumvent direct management of `state` with `setState`*
  - *`state` updates rely upon Redux management.*

React Native - State

state usage - example

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

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

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

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

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

React Native - State

state usage - example outline - part 1

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

React Native - State

state usage - example outline - part 2

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

Image - React Native - State

basic usage

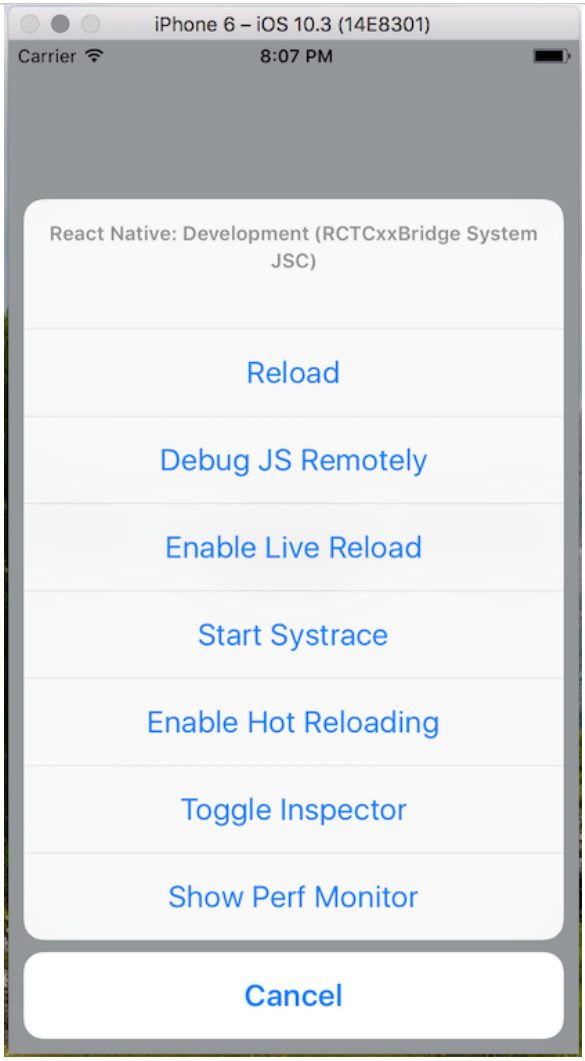
React Native - Debugging an app

Chrome DevTools

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

Image - React Native - Chrome DevTools

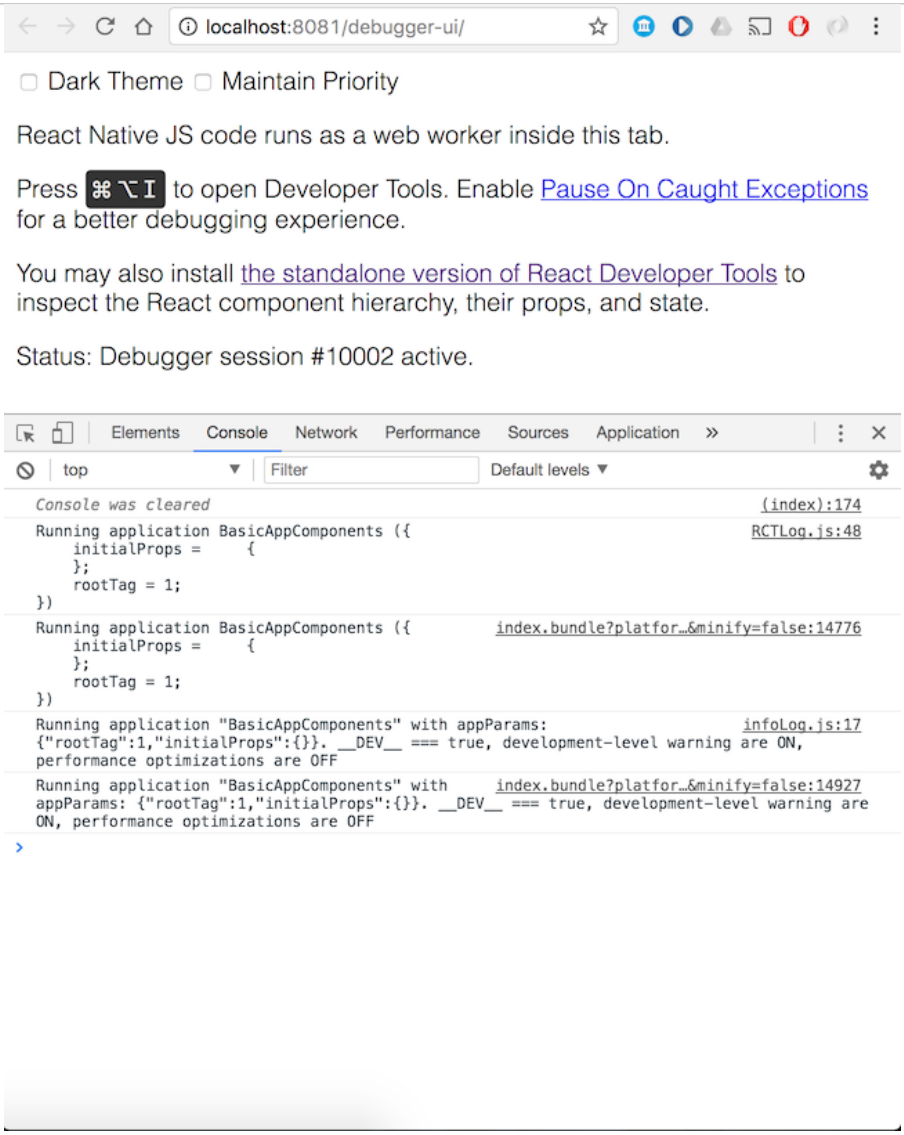
iOS simulator options



react Native Options in iOS Simulator

Image - React Native - Chrome DevTools

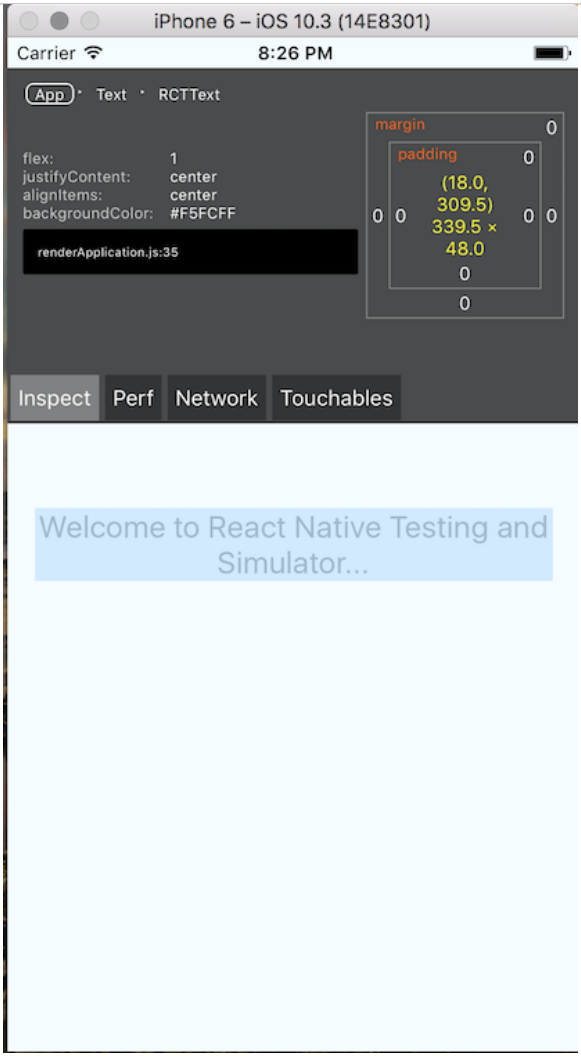
developer tools



React Native Debugging in Chrome DevTools

Image - React Native - Debug Options

inspector



React Native Inspector

React JavaScript Library

a React approach to development - part 1

- **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 with then propagate this change to the UI
- it's the change in state that causes components to be updated

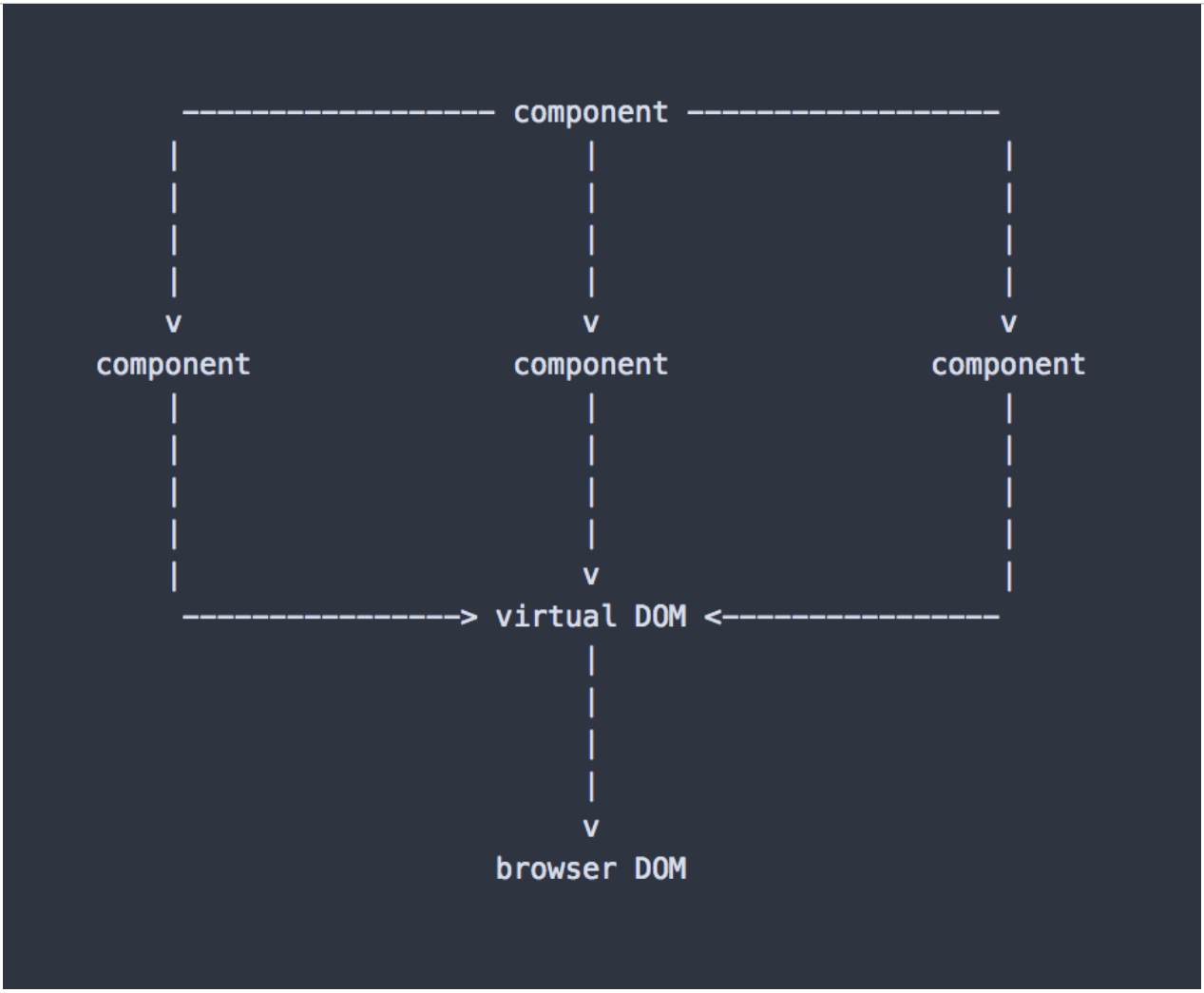
React JavaScript Library

a React approach to development - part 2

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

React JavaScript Library

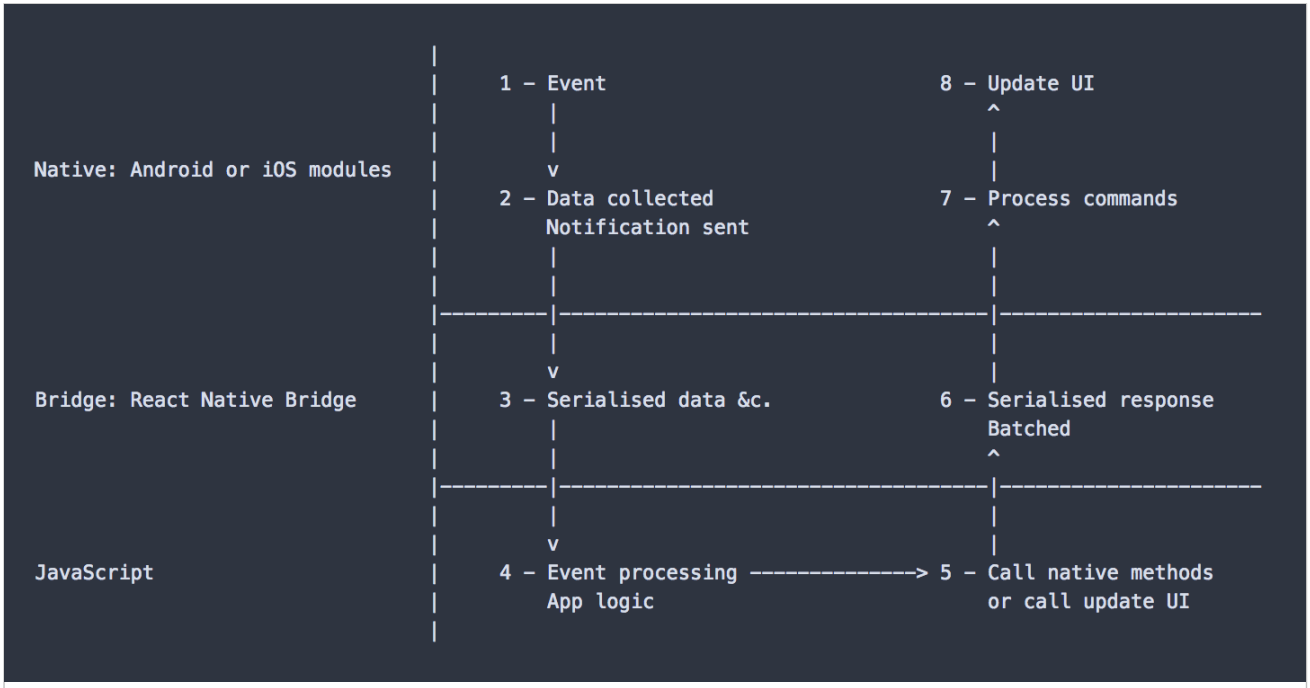
data flow



React Data Flow

Image - React Native - Structure

structural considerations



React Native Structure

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 perfomance

References

- MDN - super
- React
- React Native
- React DevTools
- React Native - Layout Props