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

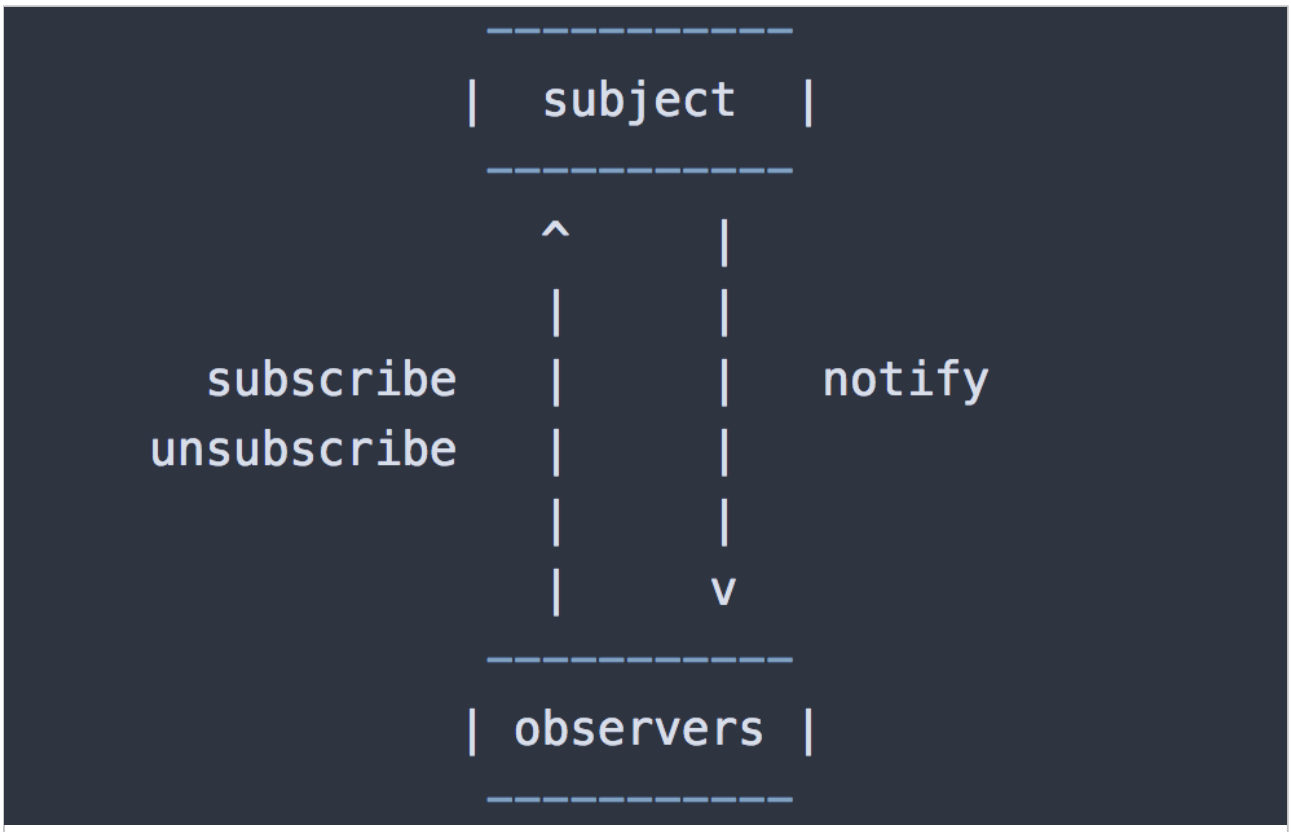
Fall Semester 2019 - Week 8

Dr Nick Hayward

Design Patterns - Observer - intro

- *observer* pattern is used to help define a *one to many* dependency between objects
- as **subject** (object) changes state
  - *any dependent **observers** (object/s) are then notified automatically*
  - *and then may update accordingly*
- managing changes in state to keep app in sync
- creating bindings that are event driven
  - *instead of standard push/pull*
- standard usage for this pattern with bindings
  - *one to many*
  - *one way*
  - *commonly event driven*

Image - Observer Pattern



Observer Pattern

Design Patterns - Observer - notifications

- observer pattern creates a model of event subscription with notifications
- benefit of this pattern
  - *tends to promote loose coupling in component design and development*
- pattern is used a lot in JavaScript based applications
  - *user events are a common example of this usage*
- pattern may also be referenced as *Pub/Sub*
  - *there are differences between these patterns - be careful...*

Design Patterns - Observer - Usage

The observer pattern includes two primary objects,

- **subject**
  - *provides interface for observers to subscribe and unsubscribe*
  - *sends notifications to observers for changes in state*
  - *maintains record of subscribed observers*
  - *e.g. a click in the UI*
- **observer**
  - *includes a function to respond to subject notifications*
  - *e.g. a handler for the click*

Design Patterns - Observer - Example

```
// constructor for subject
function Subject () {
  // keep track of observers
  this.observers = [];
}

// add subscribe to constructor prototype
Subject.prototype.subscribe = function(fn) {
  this.observers.push(fn);
};

// add unsubscribe to constructor prototype
Subject.prototype.unsubscribe = function(fn) {
  // ...
};

// add broadcast to constructor prototype
Subject.prototype.broadcast = function(status) {
  // each subscriber function called in response to state change...
  this.observers.forEach((subscriber) => subscriber(status));
};

// instantiate subject object
const domSubject = new Subject();

// subscribe & define function to call when broadcast message is sent
domSubject.subscribe((status) => {
  // check dom load
  let domCheck = status === true ? `dom loaded = ${status}` : `dom still loading.
  // log dom check
  console.log(domCheck)
});

document.addEventListener('DOMContentLoaded', () => domSubject.broadcast(true));
```

Design Patterns - Observer - Example

- Observer - Broadcast, Subscribe, & Unsubscribe

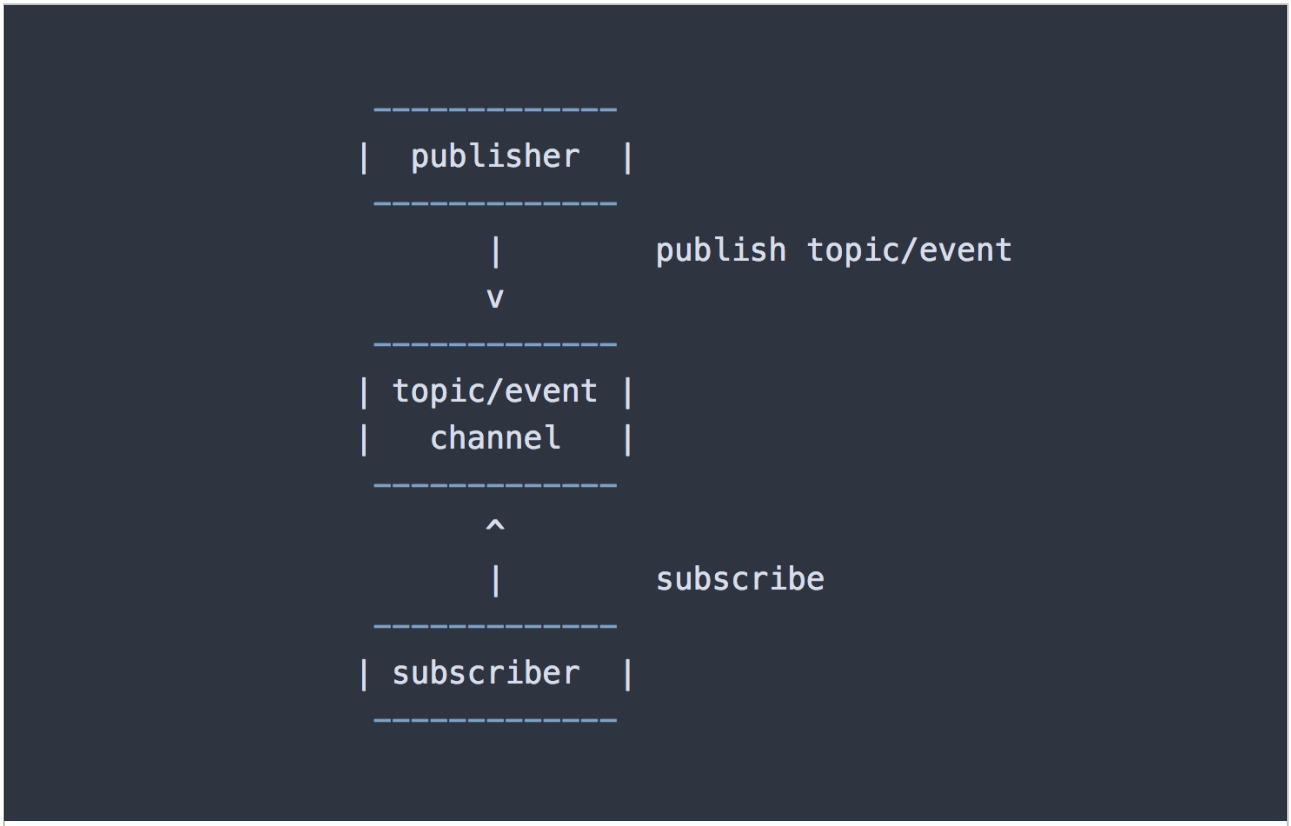
Design Patterns - Pub/Sub - intro

- variation of standard *observer* pattern is *publication and subscription*
  - *commonly known as PubSub pattern*
- popular usage in JavaScript
- *PubSub* pattern publishes a *topic* or *event* channel
- publication acts as a *mediator* or *event system* between
  - *subscriber objects wishing to receive notifications*
  - *and publisher object announcing an event*
- easy to define specific events with *event system*
- events may then pass custom arguments to a subscriber
- trying to avoid potential dependencies between objects
  - *subscriber objects and the publisher object*

Design Patterns - Pub/Sub - abstraction

- inherent to this pattern is the simple abstraction of responsibility
- publishers are unaware of nature or type of subscribers for messages
- subscribers are unaware of the specifics for a given publisher
- subscribers simply identify their interest in a given topic or event
  - *then receive notifications of updates for a given subscribed channel*
- primary difference with *observer* pattern
  - *PubSub abstracts the role of the subscriber*
- *subscriber* simply needs to handle data broadcasts by a *publisher*
- creating an abstracted *event system* between objects
  - *abstraction of concerns between publisher and subscriber*

Image - Publish/Subscribe Pattern



PubSub Pattern

Design Patterns - Pub/Sub - benefits

- *observer* and *PubSub* patterns help developers
  - *better understanding of relationships within an app's logic and structure*
- need to identify aspects of our app that contain direct relationships
- many direct relationships may be replaced with patterns
  - *subjects and observers*
  - *publishers and observers*
- tightly coupled code can quickly create issues
  - *maintenance, scale, modification, clarity of code and logic...*
  - *semmingly minor changes may often create a cascade or waterfall effect in code*
- a known side effect of tightly couple code
  - *frequent need to mock usage &c. in testing*
  - *time consuming and error prone as app scales...*
- *PubSub* helps create smaller, loosely coupled blocks
  - *helps improve management of an app*
  - *promotes code reuse*

Design Patterns - Pub/Sub - basic example - part 1 - event system

```
// constructor for pubsub object
function PubSub () {
this.pubsub = {};
}

// publish - expects topic/event & data to send
PubSub.prototype.publish = function (topic, data) {
  // check topic exists
  if (!this.pubsub[topic]){
    console.log(`publish - no topic...`);
    return false;
  }
  // loop through pubsub for specified topic - call subscriber functions...
  this.pubsub[topic].forEach(function(subscriber) {
      subscriber(data || {});
    });
};

// subscribe - expects topic/event & function to call for publish notification
PubSub.prototype.subscribe = function (topic, fn) {
  // check topic exists
  if (!this.pubsub[topic]) {
    // create topic
    this.pubsub[topic] = [];
    console.log(`pubsub topic initialised...`);
  }
  else {
    // log output for existing topic match
    console.log(`topic already initialised...`);
  }
  // push subscriber function to specified topic
  this.pubsub[topic].push(fn);
};
```

Design Patterns - Pub/Sub - basic example - part 2 - usage

```
// basic log output
var logger = data => { console.log( `logged: ${data}` ); };

// test function for subscriber
var domUpdater = function (data) {
  document.getElementById('output').innerHTML = data;
}

// instantiate object for PubSub
const pubSub = new PubSub();

// subscriber tests
pubSub.subscribe( 'test_topic', logger );
pubSub.subscribe( 'test_topic2', domUpdater );
pubSub.subscribe( 'test_topic', logger );

// publisher tests
pubSub.publish('test_topic', 'hello subscribers of test topic...');
pubSub.publish('test_topic2', 'update notification for test topic2...');
```

- Demo - Pub/Sub

Mobile Design & Development - Patterns

Fun Exercise

Four groups, one app per group:

- Fast Food - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/fastfood/
- Ingredients - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/ingredients/
- Street Food - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/street-food/
- Supermarkets - http://linode4.cs.luc.edu/teaching/cs/demos/422/gifs/supermarkets/

For your assigned app, consider the following

- where may you use either the Observer or Pub/Sub pattern in the app?
  - *consider from a developer's perspective*
- which parts of either pattern, Observer or Pub/Sub, creates a unified UX?
  - *consider UX in the app, and then compare with use of chosen pattern...*

~ 10 minutes

React JavaScript Library

overview

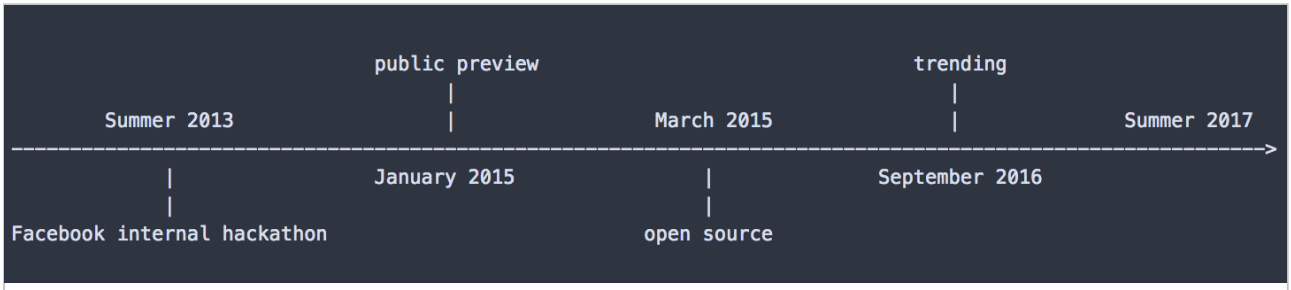
- **React** began life as a port of a custom PHP framework called XHP
  - *developed internally at Facebook*
- XHP, as a PHP framework, was designed to render the full page for each request
- **React** developed from this concept
  - *creating a client-side implementation of loading the full page*
- **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*
  - ...

React JavaScript Library

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

React JavaScript Library

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)*

React JavaScript Library

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*

React Native

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

React Native

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 1

- clear the boilerplate 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>
    );
  }
}
```

React Native

app - getting started - part 2

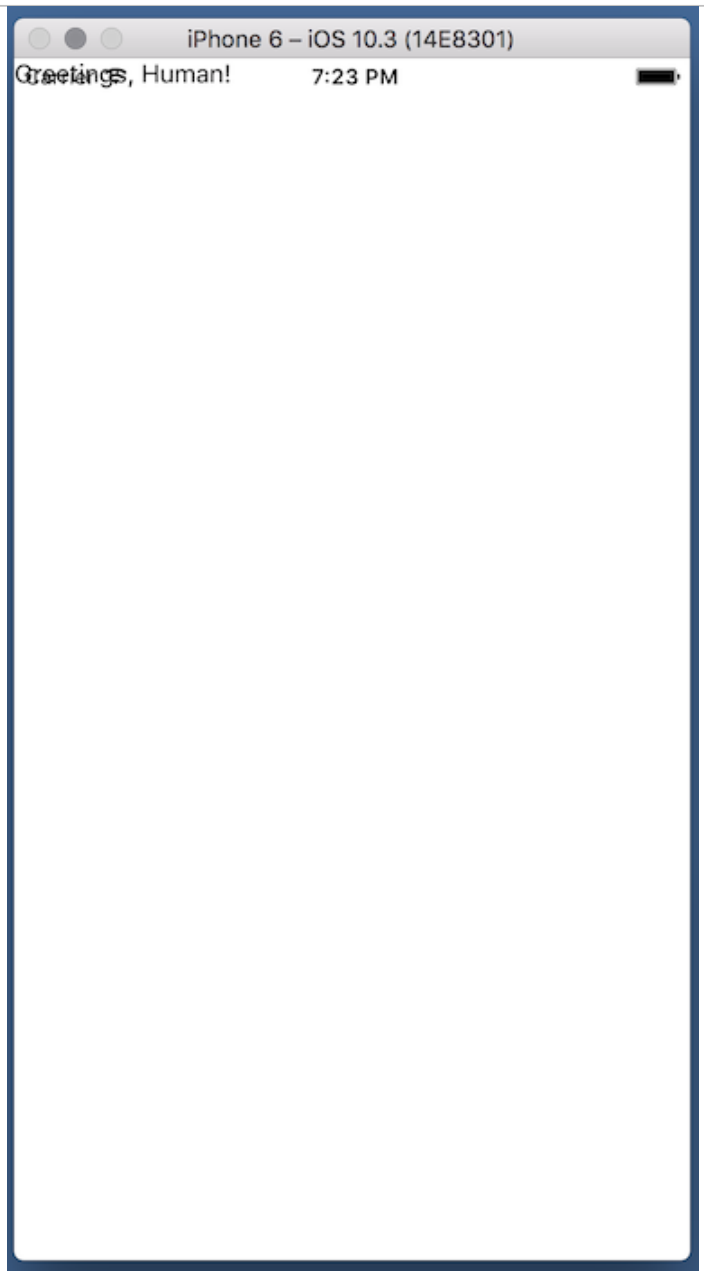
- use this new component within our app
- register it in the default `index.js` file, e.g.

```
// 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 p
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*

References

- Cordova
  - *OnsenUI - JavaScript Reference*
  - *Whitelist plugin*
- React Native
  - *React*
  - *React Native*
  - *React DevTools*