React (v16.8.0-alpha.0)

Notes open for creative commons use @ developer blog: https://unfoldkyle.com, github: SmilingStallman, *email: kmiskell@protonmail.com*

React Key Concepts

-A JS UI library used to build and display HTML views. Renders interfaces in SPA and mobile apps. Useful for very dynamic sites (ex. Facebook).

-Library made of UI components created using pure javascript. Components highly contained. Components have both visual representation and dynamic logic. Components can interact with server. Component-based architecture (CBA).

-Note: it is common for people to shorthand web GUI as just UI

React Core Concepts & Uses
-Useful in building and managing complex dynamic UIs. "Building large applications with data that changes over time."

-UIs as functions. Call them with data. Get rendered in view with automatic updates. Renders components in memory and only updates elements with change state instead of re-rendering whole DOM. Much faster for dynamic sites. Powerful abstractions for same view on many browsers.

-Declarative over imperative. Imperative involves creating a lot of vars, then interacting with them. Declarative creates using functions, often calls on objects returns from another function call, etc. Shorter, more readable. Less split code/logic. Less local vars.

-Change in view = state change. React updates view. Uses virtual dom running in background to detect diffs in existing and new views.

-Unlike angular, and some others, is all pure JS with no templates in other lang. Split lang and files for single component. React can do all in one file in pure JS.

-React abstraction provides wrapper around events. Can render elements on sever.

-Speed. React virtual DOM exists in JS memory. Real Dom rendered in browser. Virtual dom then rendered by react. If virtual dom changes, diff with real dom will show, and react will update real DOM, but only elements of DOM that have changed.
-"Internal state" = virtual dom, "view" = real dom

-Large community with many already made components that you can use

-Note that since react is barebones, need to pack with routing and data modeling libraries like flux, react router, which can be done with *create-react-app*

-JSX - tiny syntax for writing React object in JS using < >, as in HTML, to define elements. JSX src code compiled into JS for browser.

Building a React Project

React Code 101

-*someArray.map(transformFunction)* - Returns an array of values after applying the transform function
-See also *Array.fillter(), Array.reduce(), Array.some(), Array.every(), Array.reverse()*

-If creating react app in most barebones way, will need three files: index.html, react.js, and react-dom.js. The JS files come from the react source module library.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="js/react.js"></script>                    //imports react library
    <script src="js/react-dom.js"></script>                //imports reactDom library
  </head>
  <body>
    <script type="text/javascript">
          let someElement = React.createElement(..);       //create react element
          ReactDOM.render(...)                             //render element(s) onto dom,
    </script>                                                 //ex. place on existing <div>
  </body>
</html>
```

-Note: standard to have create react elements and render methods in a separate file (ex. App.js) and render onto html files, which imports via <script src="...">

-Can install basic local http server with npm via *npm i -g http-server*, then navigate to folder containing site files, then run *http-server* to start server. ip to use for local host for browser access, etc. will show.

App Creation & Server with *create-react-app*
-install *create-react-app* with npm, then in terminal create app via: *create-react-app appName*, then start host via *npm start* from *appName* dir

-Once run *npm start*, will run the react server locally on your computer, after which the project can be inspected from the browser. Typical is *http://localhost:3000/*

Project Folder with create-react
-Basic flow: index.html and index.js are the key points which create-react builds on. index.js imports React core modules and App.js serves as the default component and holds a class which extends a component. It has a a *render()* method which returns react elements. index.js holds the ReactDom.render() method, which renders an instance of App via the App's render() method, on a specified location in the html dom (ex. a * *, which the index.html file contains

APP.JS (component with *render()* method) -->rendered by---index.js---->displayed on----index.html

-Folder contains:
node_modules package.json package-lock.json public README.md src

-node_modules - folder, holds library in node packages, including some defaults if used *create-react-app* to instantiate project
-public - folder, holds assets, html, favicon, images, etc.. Is hosted on *localhost:3000* for access to
-src - folder, holds js, css, etc.
-build - built when build project, for combo of needed src and public files

-index.html file is super basic DOM tree, (React will inject html later) with only two body element.
I) a *<noscript>* element to display an error message is browser doesn't have JS running
II) a *id="root" *, which will have components built onto it by REACT

-index.js - contains various *import* react modules, css, etc. Entry point into react.
-import css via *import './fileLocation.css';*
-Imports JS *import Hello from './components/hello';* //hello is a js file, but you leave .js off

-App.js - main view for intial project. Where you will implement REACT components in REACT rendering of view. Components generally in separate files, and included in app.js

-App.test.js - for holding program tests

-index.css and App.css

-*ReactDOM.render(<component />, document.getElementById('root'));* - renders component into browser. Does so realtime, so browser will auto-update as soon as any code changed.

-*npm start* - start the app development server. Can then acess in website, default via localhost:3000
-*npm build* - bundles the app into static files for production
-*npm test* - starts the test runner
-*npm eject* - removes npm tool from project and copies build dependencies, config files, and scripts into app dir. Once *eject*, cannot be undone.

-bootstrap - on OS level, the program that initializes the OS during startup. May also see someone use on more local terms, "ex. react is bootstrapped," ie the react environment is up and running on the local host

Intro to JSX

-Introduced largely as a way to create quicker, more reable react code, so you're not writing long nested *createElement* statements. Shorthand sugar to create react elements. A small language with XML-like syntax.

-Allows you to efficiently define and create react elements with complex html definitions, in readable

code. Allows you to mimic HTML coding from React, so don't have to ever really touch .html files
-JSX code compiled by various transpilers into standard JS for browser use

-In react, elements are instances of components
-With JSX, can create instance of component by calling *<ComponentName />*
-Can inspect elements in react.dom by *console.log(React.DOM)*

-JSX is supported by default in *create-react-app* as it includes Babel

-JSX react element creation ex.

```
<elementName key1='value1' key2='value2' ...>
    <Child1/>                  //create component of class Child1
     <Child2/>                 //create component of class Child2
</elementName>
```

Ex. *let h1Elm = <h1>Hello world</h1>;* //could also pass right side code to RDOM.render(), etc.
//object creation and instantiation

-Babel will transpile elements created with JSX into *React.createElement()* calls

-Can pass expression, props names, or binding inside { } and value of will be output, similiar to template literals
-ex. *{new Date().toLocaleString()}*

ex. *return ({helloReactComponent});*

Rendering Elements

-ReactDOM.render(<component />, document.getElementById('root'));
-Can also pass binding name of existing component instead of *<component />*

-React elements are immutable once created. If you want unique properties, set them at instantiation time.

Components & Props

Component Classes
-Can react component as classes via a class that *extends Component* class
-Make sure .js file for components imports *Component* via:
import React, { Component } from 'react';

-Only mandatory method for component is *render()*, which must return a single element, created from an html tagName or another as component. If need multi elements, wrap in * *, etc. container and return that

Ex. *class HelloWorld extends Component {*
render() {
return Hello World ;
}
}

ReactDom.render(React.createElement(HelloWorld, null, document.getElementById('contentDiv');
//renders onto content in html

-If *return* covers multiple lines and no text following *return* on first line, enclose lines in ( );

-Naming syntax: *let ComponentInstance = ...;*

Creating Components via Functions
-Can also define by returning a <someElement> React element from a function. Would then render function call, which renders component. Preferred over classes.

Methods Inside Components
-Can write component method in from
methodName() {...}

-Can call on class methods from with class via *this.methodName()*

Props
-Props are created for react elements via <*someElement keyName="value"*>. If *keyName* is standard HTML name, will be applied to element as if HTML attr val. If custom, not applied, but can access from within class via *this.props.keyName*. Props are immutable.

-To make dynamic objects, create the objects based on arguments passed in as properties (props) by JSX < > instantiation statements, then use those properties to create the component inside the component class by referencing the specified props args via *this.props.propertyName* values.

-Ex. create a class with an img and <a> inside a div, then pass in different img urls and links

-If want to display one of two options (ex. logout or login), ternary op inside <> component creation *propertyValue={ ... }* where one of two values is selected based on a *this.props* flag is a quick way to do so
-ex **
{(this.props.sessionFlag) ? '/logout' : '/login'} //renders different text tool
**

-If storing any custom html attributes, name attribute in *data*TheName format, to not accidentally override any native attributes by accident

-Reminder for ES6: rest parameter ..., ex. ...*someArray*, When last parameter in args list has a ... before

it, that parameter becomes an array of that name and holds any more args that are entered

CSS Styles

-To access and set CSS styles, use *this.props.style* or style style inside JSX definition via <*div style={...}*>, etc.

-Handy to create a single *const styles = {}* object at top of script, then give properties, where each property is an inner object containing styles for different elements in script, then apply styles via <*div style= {styles.someStyle}*>

-ex. *const styles = { inputBox: {//styling for input box},*
nextButton: {//styling for next button},
etc.};

-Note naming for css like this: *background-image* becomes *backgroundImage* when referencing in JavaScript/JSX

-To build JavaScript object inline do so in double curly braces {{...}}

-Example of creating element with inline style

```
<span style={ {border: '3px solid black',
              font-size: '2rem'
            } }
 </span>
```

Reserved Words - class, for

-As *class* and *for* are reserved words in JS, when defining an element using JSX and REACT and giving it a *class="someName"* attribute, use *className="..."* instead. Same with HTML labels that take a *for="someElement"* attribute. When setting attribute via JSX, use *htmlFor="..."* instead

Boolean Attribute Values

-For boolean attributes (ex. *loop* for <*video*>), define them as *loop={false}* or *loop={true}*. Can also omit *{...}* definition and then JS will take as true (ex. <*input disabled*>)

Comments in JSX & Random

*{/*JSX Comment*/}* - same as JS, but wrapped in {}

-Still use html codes (ex. #40;) when called for when creating inner html content (ex. inside <p>) with JSX

States & Lifecyles

-Allows you to store data for react components even with component instances having immutable props. Can automatically augment views based on data changes. Local to instances.

-React state - A mutable data store held by components. *this.state* object, which is held by components, and its attributes. Ex. *this.state.inputValue*. When a state changes, corresponding parts of view that require changes are updated in DOM by virtual dom.

-If want to pass to children, pass *this.state* as a props attr. ex. *<FormattedDate={this.state.date} />*.

-*{this.state.inputFile}* - will show on dom if put in *render()*

Instantiating States
-Can do via constructor inside component class:

```
constructor(props) {
   super(props)                  //necessary for it to have same properties as parent
   this.state = { keyA: "valueA",
                  keyB: expressioncall(),
                  keyC: [ arrayValues, ... ]
                  …
                }
}
```

-This is necessary because ECMAScript doesn't have support for class variables.…

Lifecyle Methods
-Important for free'ing up resources by destroyed components. Done in steps:
1) Pre-mount - creation of component and initial state in component constructor
2) Mount - Rendering of component onto DOM.
3) Unmount - Cleanup phase via methods called just before removed from DOM. Ex. *clearInterval()*

-Lifecyle methods - *componentDidMount(){...}* and *ComponentWillUnmount(){…}*.

-*componentDidMount()* - Runs automatically just after component rendered onto DOM. If have initialization methods for state, call them from here instead of in constructor. Ex. *setInterval()*

-*componentWillUnmount()* - Runs auto just before component removed from DOM. Use to reset state set via *componentDidMount()*. Ex. *clearInterval()*

-*setInterval(function, milliseconds)* - a *window* (global) function, that calls *function* every *milliseconds*.
-*clearInterval(function)* - Clears interval for interval for *function* set by *setInterval(function, ms)*
-Also *setTimeout(function, ms)* - same, but only executes once after *ms*
-Good way to update components on a timed basis

Updating States
-Avoid setting state directly via *this.state*
-Update states via *this.setState({keyA: "valueA" , …})*

-Note that calling *setState()* triggers *render()*

-*this.state* and *this.props* may be asynch, so when using *setState()* do not simply set state/props inside *setState* by accessing them via *this.state/this.props.* Instead, do as so:

```
this.setState((state, props) => ({
   counter: state.counter + props.increment
}));
//note that since calling from this scope, don't need to add this to state or props args
```

-Any method that updates state should be contained or called from only *componentDidMount()*

Stateless Components
-Have no state or lifecyle events/methods. Can take props and render a view, but that's it.
-Predictable. Easy to understand, maintain, & debug. Prefer stateless over stateful.

-Instead of creating components via *class Xyz extends ComponentNamei,* can create stateless components by creating components as a function *const Xyz = (args) => {<elements>},* then export function binding

-Instead of state, stateless components have two properties *propTypes* and *defaultProperties*

Stateful vs. Stateless - Why?
-Declarative over imperative. Better syntax with greater simplicity. Sometimes (ex. clock), not always possible, but prefer.

-If want to work towards stateless as possible in program that requires state, separate stateful from stateless components and import stateful to be used in stateless

Moving From Stateful to Stateless
-Where stateful components are generally implemented as classes, stateless are implements as functions, most often fat-arrow function, as described above

Handling Events

-As with css, events in React camelCase instead of hyphen-seperated
-Also, when setting prop as function call in react, set via *onClick={somefunction},* not *onClick="someFunction()"*

-To prevent default in event handling, call *event.preventDefault()* on event in event handling function

-Instead of using *addEventListener* to listen for event, common to set event handler when defining react node. Ex:

<*button onClick={this.handleClick}*> //when *onClick* occurs, *handleClick()* handles

-Important: class methods in JSX not bound by default to *this* of class, so make sure to bind if needed by calling *this.functionName = this.functionName.bind(this)* inside class constructor

-Anytime referring to a method name via *this.someMethod* instead of *this.someMethod()*, where are actually calling it, that method should be bound to proper *this* as described above, as per standard JS requirements for setting proper this