# React (16.12.0)

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

**Learning Resources**
-Primary: React Official Docs: https://reactjs.org/docs
-Secondary (practical ex.): *Complete React Developer in 2020 (w/ Redux, Hooks, GraphQL)*

## React Key Concepts

### React Intro

-Single Page Application – Code only loaded once, stay on page throughout interaction of, and JS updates page as needed. Since JS handling all updates, and running on host, less need for server communications

-Angular (by Google) replaced JQuery as standard in ~2010 and created better organization through MVC

-React created by FB in 2013 in response to ads app framework becoming too hard to manage

-React was created as an alternative to imperative programming, where directly changing individual parts of your program in response to user events

-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).

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

### Declarative Paradigm

-Imperative programming – emphasizes *how* to do something with things like for, while loops, if, else, classes, objects with statements that explicitly manage state (ex. The DOM)

-Declarative programming – focused more on *what* you want to do, without explicitly stating how to do it. In JavaScript, done largely through map, reduce, filter, etc.. More abstract and less flexible than imperative, but simplifies and shortens code.

-SQL is an example of declarative, as you *declare*, what you want it to to (ex. *SELECT*) without specifying the logic for *SELECT*.

-Declarative vs Imperative further reading:
https://learntocodetogether.com/imperative-vs-declarative-programming-what-is-the-difference/
https://codeburst.io/declarative-vs-imperative-programming-a8a7c93d9ad2

-React is declarative because it tells what the DOM should look like, without explicitly defining all logic for this. Example. Tell React to run all images in an array through a map, then render images for each item present in map > 200kb in size. While you tell react *what* you want to happen, you like React handle the *how* of rendering this.

-React DOM based on state of data

**React Key Concepts**
I) Don't touch the DOM. React will handle rendering.

II) Component based. Site broken down into small components, while compile larger site, and can be re-used throughout site or on other projects
-Components often built through functions/classes, where function takes in state of site, then builds component based on with JSX

III) Uses unidirectional flow. VirtualDOM which is essentially a blueprint of the actual DOM. As the virtualDOM updates from state change, etc., actual DOM is also updated. One way, as data only flows down (state changes (ex. user action) → virtualDOM updates state affected components → DOM updated by VDOM). Makes for easier debugging as not full of side effects, etc.. Summary: Action → State → Views

IV) React is not complex. It simply gives you UI focus, without being highly verbose, unlike the Angular framework, which is more heavy. Easy to move from React browser, to React Native, desktop apps with React, etc..

**Strong React Development**
I) Deciding on how to break down app and what components to use
II) Deciding where the state is and what component(s) it lives in
II) Deciding on what changes when state changes

# Installation & create-react-app

**Setup**
I) Node (check installed properly w/ 'npm -v' and 'node -v')
II) create-react-app – builds up all webpack, Babel, etc. config to save time. React is included in create-react-app.

-To build react app, navigate to folder you want app in, then run *create-react-app appName*

-Every time run *create-react-app,* will update to latest version, which also uses latest React version

-In addition to building basic react app, also includes *start, build, test,* and *eject* scripts, to run React app in Node on localhost
-To run, call *npm commandname reactAppName* from folder app root folder.
    -ex. *npm start lains-react-app*

**create-react-app folders & Files**
I) src – all react app code goes here
II) public – built react code, pulled from src folder by React, ends up here, to be used by browser. Assets, html, favicon, images, etc..

-node_modules - folder, holds library in node packages, including some defaults if used create-reactapp to instantiate project

-App.js - imports essential react components (*import React from 'react'*), css, logo, etc. and contains the code for what to render (JSX built from various components, etc.)

-index.js – also imports basic React components. Main purpose is to render App component to VirtualDOM via call to *ReactDom.render(<App />, document.getElementByID('root'));*

-index.html – contains super barebones html code, including * *, where app is rendered to

-package.json – includes all the various extra components installed for the react app, such as create-react-app

**Building React Apps**

-To deploy app run: *npm run build* in react app folder, which builds and optimizes code for run in browser. This code is then stored in a '*build'* folder inside the app folder, which is what should be hosted on the webserver.

**Babel & Webpack**
-Babel is transpiles JSX into JS to be read by browser, ensuring cross-browser comparability in process
-Webpack takes JS files of app and combines into less files, for greater optimization

# JSX

**Intro**
-Similar to template lang, that allows you to write HTML style code in JS, which then transpiles into html. Lets you write HTML easily from within JS.

-After compilation, JSX become JS function calls and eval to JS objects, thus Can store JSX in variables and return JSX code

-ex. *const element = <h1>Hello, my name is { user.name }</h1>*

-To embed JS in "html" wrap in curly braces
-Can also embed variables, function calls, etc.
-If embedded JSX spans multi lines, wrap in ( )

-JSX elements can have children, just like HTML elements

-If empty tag, can close immediately with: />
    -ex. *<Homepage />*

**Attributes**
-Uses camelcase for attribute names
-Wrap string attribute values in quotes, JSX attribute in { }

**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 l*oop ={ true}*. Can also omit *{...}* definition and then JS will take as true (ex. *<input disabled>*)

# Components

-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).

-Components can be seen as rooms in a house. Each room has a basic design. If you want 4 bedrooms, render 4 *<Bedroom />* components. If you want unique bedrooms, build the base component, then pass in specific data to it during creation of. A site is made of many components, each representing a specific part of the site.

-Components can be functions or classes. Classes can have state without using hooks.

-Name components CamelCase, with first word capitalized

-All components must: *import React from 'react';* at top of JS file

-Functional components:

```
const ComponentName = props.name => return <h1>Hello, {props.name}</h1>;
```

-Component is returned from function

-Class components:

```
class Welcome extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}</h1>;
    }
}
```

-Class components contain a call to *render(),* which returns the JSX for the component

-All components must have a unique "key." This is often numerical. It is set by the *key* attribute for the component. If not set during creation, page will throw warning.

## Rendering Components

-With React, html page is very barebones, mostly just *<head>*'s *<meta>* and *<link>* elements, as well as a very basic *<body>* structure, often containing just one html element with id *root*

-JS for app then contains *render* method, which takes a react component

-React elements, unlike DOM elements, are quick to create and update

-React has virtual DOM, which watches all react elements. When a state change occurs in one, it compares to the actual DOM and updates as needed, in the most efficient manner (sometimes in batches), which updates view. All this is done client-side, as React is JS.

-Basic render: *ReactDOM.render(someElement, document.getElementById('root'));*
-Renders *someElement* component onto the *root* html node

-React elements are immutable. Every time the state of an element changes, it is re-rendered. React, unlike the DOM, only updates what needs to be updated through state changes.

## Props

-props are attributes passed into the component during time of creation, which can be referenced inside the definition of the component, to create unique components, through ways like running *map* on array

of inner objects of the same “kind,” outputting one component per map iteration.

-Props ex.

```
const Person = props => <p>My name is {props.name}. I am {props.age} years old.</p>

<div>
  <Person name='Kyle' age='32' />          //two <p> created, each with specified name/age
  <Person name='Tom' age='40' />
</div>
```

-To shorten reference to props, use destructuring inside component. Ex.:

```
const {name, age} = props;
const Person = props => <p>My name is {name}. I am {age} years old.</p>
```

-If have a class component, must reference props via *this.props*

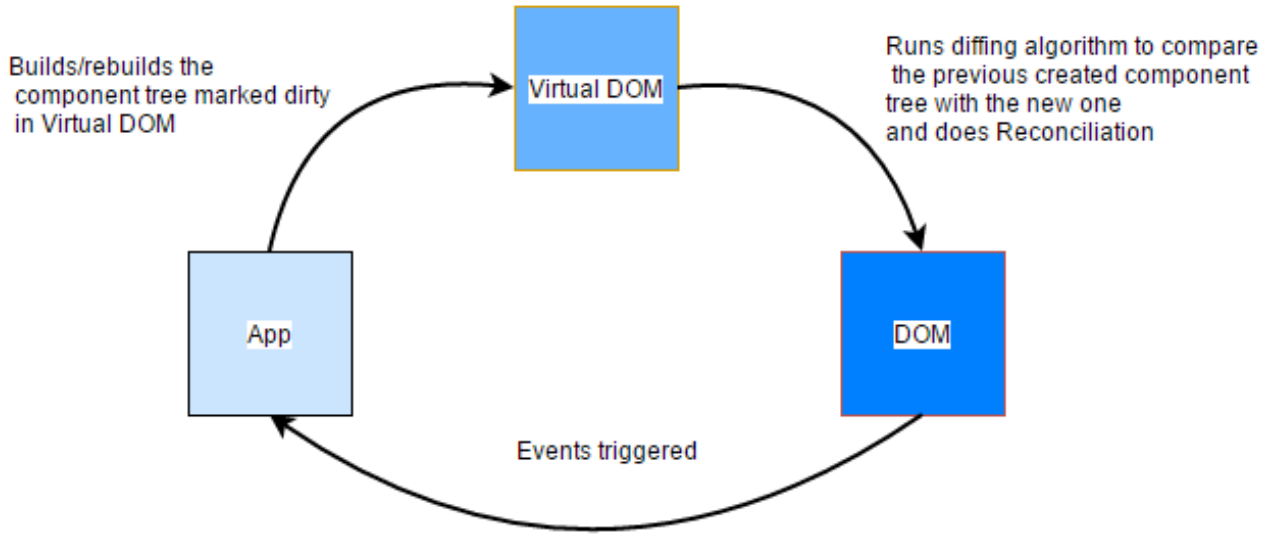
## Virtual DOM in Depth

-more info: https://programmingwithmosh.com/react/react-virtual-dom-explained/

-When using React on desktop development, are actually using *react* and *react-dom*

-*react* is the source of react components, state, props, etc.

-ex. *React.createElement('h1', {}. 'React Header');*




- *react-dom* is the glue between the VDOM and the DOM

-ex *ReactDOM.render(React.createElement(App), document.getElementByID('root'));*

-separated once React Native came out, to allow a different source than *react-dom* to handle rendering

-The VDOM holds a representation of the DOM. Whenever state changes, the VDOM updates. Once VDOM updated, *react-dom* does a *diff* on previous VDOM, then decides the fastest, most efficient way

to re-render, and only updates the state changed elements, instead of unchanged children too.

-Uses observable pattern to listen for state changes

-*render()* is the function controlling this. Every time the *state* or a *prop* changes inside *render*, react detects this, and re-enders the component with a VDOM update

-Updates to DOM are sent in batches, instead of VDOM updating DOM with every single change

# State

-State is available to class components. Unlike *props*, it is mutable. This allows you to create components that change with user interaction, etc.

-Class components wanting to use state must first call the parent (*React.component)* constructor, inside their own constructor, which then gives access to *state* and methods used to interact with state*:*

```
class Checkbox extends React.Component {
        constructor() {
                super();
                this.state = { isOn: true };
        }
}
```

-State is simply a JavaScript object. As typical, properties can hold anything from strings to inner objects, to arrays.

Example of use of state
1) Wish to render a series of <*Card* /> components, each with unique background images and titles.

2) In *constructor(){ ….. }* for card container Component, declare
*this.state = { cards: [array of objects] }*;
Each object in the array has a *imageUrl* and *title* property.

3) In *render(….)* return a series of <*Card* />s, which are created by a *map* call on *this.state.cards*, passing into the *map* callback destructed *{ title, imageUrl }*, and returning a
*<Card title={ title} image={ imageUrl} />*

4) Thus, for each *card* object in the array of cards, a unique <*Card* /> is created, based on the logic inside the definition of the <*Card* /> component, as referenced through *props.title* and *props.imageURL,* from that attributes retrieved from the *state* of the <*Card* /> creating component.

-To change state, call *this.setState({newStateObject})*

-Only change state by calling *this.setState()*, as calling *this.setState()* tells React to re-render via new call to *render()*, which has VirtualDOM update DOM

-Object passed into *setState()* can take in new properties and overwrite old properties. Passing in partial object only changes existing/adds new, without erasing unspecified old state properties (ex. State is initialized with *propertyA* and *PropertyB* values. Later, change state with modified *propertyB* value only. Overwrites *propertyB*, but does not change existing *propertyA value*).

-If want to set property name dynamically, “template literal completes the string”

-Since state is asynch, if ever using existing *this.state* or *this.props* in setting of *state*, this can cause

problems, as asynch calls do not necessarily run in order

-As *setState()* is asynch, meaning state might not be updated right away. *SetState* second arg is thus a callback, to be run after *setState* promise returns. If need to check state value in handler, etc. that sets state (ex. counter increment), do this inside callback.

-As rule to make sure proper *state* is being referenced, *setState()* also takes in a function in form:

*this.setState( (someState, someProps) => { //state object }) );*

-This is typically used to pass in the *state* and *props* as they exist at the time the function is called

-Example:

*this.setState( (curState, curProps) => { counter: curState.counter + curProps.increment } );*

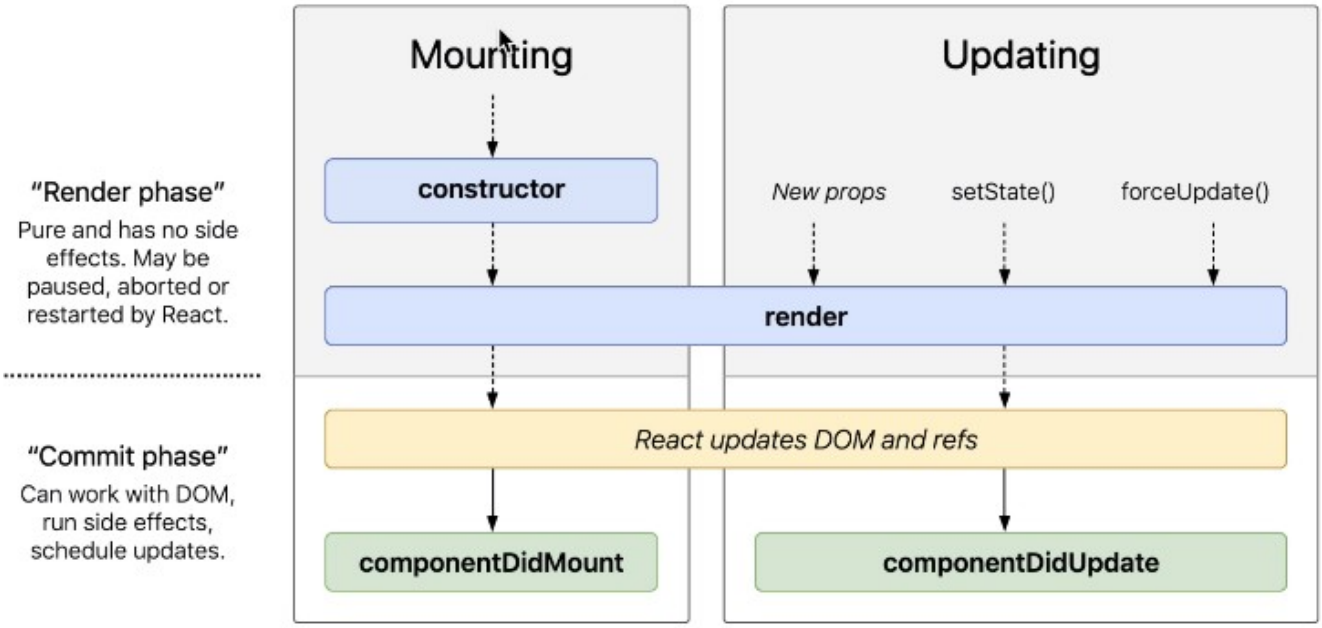
-State is only available from within the class that defines it. It can be passed down to children, which then reference it through *props*, but it cannot be referenced via a parent. This is why React has unidirectional (top-down) data flow.

-Using the ++, --, also will not work in setstate on existing states so use +1 instead with object wrapped in *( )* :

*this.setState(state => ({buttonClicked: state.buttonClicked + 1}));*

# Lifecyle Methods

-Methods that run at various stages during the component lifecycle




-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()

-*componentDidMount()* - put statements in it you want to run after the component has mounted and the page has loaded to component mount point

```
componentDidMount(){
        //statements here
}
```

-*componentDidMount()* ex. of use. Have ticking clock. Wish to update every second with a *setInterval* call that changes state each second. This goes inside *componentDidUpdate(){…..}*.

-*componentDidUpdate(){…...}* - called every time props or state updated, or *forcedUpdate()* called. ie every time component re-renders. Put statements here to run after update.

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

-*componentWillUnmount()* ex. of use. From above clock example, in *componentWillUnmount(){….}*, make call to *clearInterval* on variable holding the call to *setInterval* to free up resources.

# Event Handling

-Similar to event handling in HTML (ex. *on-click*), except camelCase (first word lowercase) and pass in the name of the handling function to

-ex. `<button onClick={activateLasers}>Activate Lasers</button>`

-If wish to prevent default, must explicitly state this inside handling function by calling:
`event.preventDefault();`

-Note that React events are "synthetic events" created by React

-Typical event handling definition is to put event handling on component during definition (ex. by adding an *onClick*), instead of using *addEventListener*

-Make sure to use *this.handlingFunction* when when setting handling function to function defined in class component

-Common to pass in arguments to handling functions, which are then used to change state, etc.

**This and Arrow Functions**
-The this inside the definition of the arrow function refers to the this of where the function was defined.

-By defining functions as variables (functional programming), this is can be made use of

-Useful for when writing class methods without needing to do a bunch of binding.

-General rule: use arrow functions to define any class methods not contained with render() or React lifespan methods.

-Example:

```
handleClick = (e) => { …..};
<button onClick={this.handleClick}>Click me</button>
```

# Conditional Rendering

-Typically used to only render some components, based on state of application

-Since variables can store functions, can pass in a function as props to be used as event handler

-When conditional rendering from inside class, typically:
1) class has method (ex. handling methods) for what to render, based on state/props
2) inside *render()*, create *let* for conditional element
3) Then have conditional code that checks current state and updates conditional element, based on state changes

-ex.

```
class ConditionalRender extends React.Component{
  constructor(){
    super();

    this.state = { loggedIn: false };
  }

  logout() { this.setState({loggedIn: false}) };
  login() { this.setState({loggedIn: true}) };

  render(){
    let button;

    if(this.state.loggedIn)
      button = <button onClick={() => this.logout()}>aaaaa</button>;
    else
      button = <button onClick={() => this.login()}>Log In</button>;

    return <div>{button}</div>;
  }
}
```

**Inline If with Logical &&**
-In JS, *true && expression === false* and *false && expression === true*

-Can this do: *{ some-condition && <tag></tag> }* where is *some-condition === true*, element will render, but will not render if *false*

**Inline If-Else with Ternary**
-ex.

```
    <div>
      {isLoggedIn ? (
        <LogoutButton onClick={this.handleLogoutClick} />
      ) : (
        <LoginButton onClick={this.handleLoginClick} />
      )}
    </div>
```

**Prevent from Rendering**
-Typically using a boolean state value that is checked before render, to determine if render or not

-ex.
```
<WarningBanner warn={this.state.showWarning} />
<button onClick={this.handleToggleClick}>
 {this.state.showWarning ? 'Hide' : 'Show'}
</button>
```

# Conditional Rendering

**Rendering Multiple Components**
-Often done with *map()*, where pass in existing collection to, and for each iteration, render element

-ex.
```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
        <li>{number}</li>                    //renders a <li> for each num in numbers
);
```

-Since JS variables, can reference vars containing React elements, to embed them in nested react.

-ex. (continuing above ex.):
```
<ul>{listItems}</ul>
```

**Keys**
-When creating list, each list item needs a *key='uniqueId'* attr assigned to it
-Used to uniquely ID each elm in list

-Speeds up React updates of the DOM, by letting React reference elements by keys, as can move update faster if list order changes

-Key is typically pulled from data using to populate *<li>* values

-If data used to create object does not have a key, or one able to create as a composite key, pass in *index* to map callback as second arg, to reference index for item as key
-Avoid doing this if possible, as may cause performance issues

**Keys Outside of Lists**
-Keys only needed when using lists, so define keys in same function that builds list
-If pull item out of list, key not important unless added back to list

-ex. If have function that defines structure for *<li>* element, then build *<ul>* with those *<li>*, add *key* to *<li>* in function that ads those *<li>* to list, not in function that creates *<li>*

**Key Scope**
-Keys only need to be unique amongst children, not globally

**Embedding *map()* in JSX**
-As typical, embed via JS wrapped in *{ }*

-ex.
```
<ul>
    {numbers.map((number) => <ListItem key={number.toString()} value={number} />)}
</ul>
```

# Forms

**Controlled Components**
-In standard HTML, form elements maintain a state in the DOM to hold their value
  -ex. *$('#someInputBox').val() returns current text in <input>*

-In React, state for form elements managed explicitly by user with *state*

-Text *<input>*s: *this.state* holds a *value* property for each element. Set *onChange* handling method for form element that takes in *event* as arg. Handling method then calls *this.setState({value: event.target.value});*

-Since each *state* change in form will be handled by handling function, can also use handling function to manipulate data in real time (ex. call *toUpperCase()* on all text input)

-Still need to call *event.preventDefault()* in forms *onSubmit* handling function if wish to prevent page reload after submission

-Reference:
  -<checkbox>: *target.checked*
  -<textarea>, <input type='text'>, <select>: *target.value*

-Review: Can skip *for* in *<label>* and just wrap instead

```
<label>
    New Password:
   <input type='password />
</label>
```

***<textarea>***
-HTML: *<textarea>Default text</textarea>*

-In React, set the contents of the text area same as with *<input>*: by giving it a *value* attribute to changed in *state* via an *onChange* handling method that sets *{value: event.target.value}*

***<select>***
-In standard HTML, set default *<option>* with presence of boolean true *selected* attr

-In react, default stored in *value={this.state.value}* attr, with *onChange* that updates *state.value* with *event.target.value*

***<input type='file'>***
**-**Read only and thus uncontrolled element, which does not need a *value* attribute

**Multiple Inputs, One Handler**
-Since a variety of input's set their *state* via a *value* attr which changes via *onChange*, can create handler that works for a variety of inputs, dynamically, via:

1) Giving identifier of *value* property name in *state* same name as *name* attribute of element
2) Updating *state* for *value* via: *this.setState( {[event.target.name]: event.target.value} );*

**Formik**
-Look into for if using forms a lot in react. Library that includes validation, keeps track of visited fields, handles form submission, etc., based on controlled components