React Documentation

Create React environment

Install using npx:
1) Download node.js
2) In command line, type 'npx create-react-app *project_name*' to create '*project_name*' folder
3) In command line, type 'cd *project_name*', then type 'npm start' to start react server

Install using npm:
1) Download npde.js
2) In command line, type 'npm install create-react-app -g'
3) In command line, type 'creat-react-app<*project_name*>
4) In command line, type 'cd *project_name*', then type 'npm start' to start react server

Difference between npx and npm:
'npx' installs react environment locally
'npm' installs react environment globally

What are created in the folder?

package.json - contains dependencies and scripts information for the project
package-lock.json - ensures consistent installation of dependencies
node_modules (folder) - all the dependencies installed
public (folder):
- manifest.json - related to progressive web apps
- index.html - the only html file in the application for single page application (id = root)

src (folder): most of the work done in this folder
- index.js - the starting point of a react application. The App component is rendered inside 'root' DOM
- App.js - contains the App component in index.js. It represents the view we see in browser
- App.css - generated when App.js is generated. Provide styling and corresponding to App.js
- App.test.js - generated when App.js is generated. Provide unit tests
- index.css - entry point css file for HTML. Corresponding to index.html
- serviceWorker.js - related to progressive web apps

When run npm start, what programs are running?

1. index.html runs and opens up the browser
2. since index.html contains 'root' DOM node, the program enters index.js
3. react renders 'App' component into 'root' DOM node
4. the 'App' component contains HTML code, which is displayed in the browser

Components

- App.js contains the biggest component in react program. It contains other components.
- Components is usually placed in a javascript file
- Can be reuse

Component types

- Functional Component
    - Basic javascript function, return HTML tag
- Class Component
    - Class extending component class
    - Must contain render method which returns HTML

How to create components?

1) Functional Component
    1. create a javascript file named '*comp1.js*'
    2. add " *import React from 'react'* " to the front (so we can use jsx syntax)
    3. create javascript functions
    4. at the end, export this component: '*export default comp1*', where comp1 is the name of the function

    *export default can only export one function. To export multiple functions, use 'named export'

    *named export:
        1. 'export const f_name = () => ...'. For each function, put export in front of them

2) Class Component
    1. create a javascript file named 'comp2.js'
    2. add " *import React, { Component } from 'react'* " to the front (so we can use jsx syntax)
    3. create class functions
    4. at the end, export component (same as above)

How to create javascript functions?

1. Old method:

```
function Great(){
return <h1>Hello World!</h1>
}
```

2. New method (es6 arrow function syntax):

```
const Great = () => <h1>Hello World!</h1>
```

How to create class functions?

1. example:

```
class Welcome extends Component {
render(){
    return <h1>Class Component</h1>
    }
}
```

How to use created components in App.js?

- *If export default (only 1 function)*

1. go to App.js
2. add " *import comp1 from 'path/comp1'* " to the front
3. add '*<comp1></comp1>*' to the desired location

- *If using named export (multiple functions)*

1. go to App.js
2. add " *import { f_name } from 'path/comp1'* " in the front
3. add '*<f_name></f_name>*' to the desired location

When to use functional/class component?

Functional component
PROS:
- simple functions
- do not use 'this' keyword
- mainly responsible for UI

Class component
PROS:
- rich features
- maintain private data
- complex UI logic
- lifecycle hooks

Component Lifecycle Methods
4 Stages:
Mounting: when an instance of a component is being created and inserted into the DOM
Updating: when a component is being re-rendered as a result of changes to either props or states
Unmounting: when a component is being removed from the DOM
Error Handling: when there is an error during rendering, in a lifecycle method, or constructor of any child component

Mounting Phase:
- **constructor (props)**:
    - get called when new component is created
    - initializing state
    - binding the event handlers
    - need to call super(props): we have to call super to pass in *props*
- **static getDerivedStateFromProps(props, state)**:
    - use when the state of the component is dependent on changes in props over time, to set state
    - cannot use 'this' keyword. Need to return an object that represent the state
- **render ():**
    - Only required method
    - read props/state and return JSX
    - DO NOT change state or interact with DOM
- **componentDidMount():**
    - invoked immediately after **a component and all its children components** have been rendered

Updating Phase:

- **static getDerivedStateFromProps(props, state)**
- **shouldComponentUpdate(nextProps, nextStates):** determines if the component should re-render or not
    - Perform Optimization
- **render()**
- **getSnapshotBeforeUpdate(prevProps, prevState):** called right before the changes from the vitual DOM are to be reflected in the DOM
    - Used to capture some information from the DOM
    - return null or value
    - return value will pass to the next method
- **componentDidUpdate(prevProps, prevState, snapshot):** called after the render is finished in the re-render cycles

Unmounting Phase:

- **componentWillUnmount():** Method is invoked immediately before a component is unmounted
    - cancel network requests, removing event handlers, cancel subscriptions and timers
    - DO NOT call setState method

Error Handling Phase:

- **static getDerivedStateFromError(error):**
- **componentDidCatch(error, info):**

USED WHEN THERE'S ERROR DURING RENDERING

JSX

JSX: Extension of javascript lanaguage syntax

Why do we need JSX? Because it makes our code simpler and concise

How do we use JSX? By Importing React library and follow the syntax

Key points:

- **JSX assign css class to tag:** <div className = "hello"> </div> (Replace class by className)
- camelCase property naming convention
- '*for*' keyword replaced by '*htmlFor*'

Properties

- Properties are used for component dynamic. It contains attributes/values of the object.
- It is like a function. We **pass through** parameters to the function, and assign function name in App.js
- Props are immutable. We **CANNOT** change it through component itself.

Functional component:

Pure Text Solution:

1. create a/multiple '*props*' in component class parentheses (Ex. const func = (props) => ...)
2. go in App.js, add attributes inside the component tags (Ex. <func name="Kevin"></func>
3. treat 'props' as an object, we place '{ props.name }' in the desired location

* Note, we treat everything javascript scripts when use { }

HTML Solution: (Introducing children)
1. create a/multiple '*props*' in component class parentheses (Ex. const func = (props) => ...)
2. go in App.js, add regular html code **IN BETWEEN** component tags (Ex. <func> ***HTML CODE*** </func>
3. go back to component class, add '{ props.children }' in the return statement

* keep in mind, return can only include one html code. Thus, we need '*div*' tag to include all html codes

Class component:

Pure Text Solution:
1. go in App.js, add attributes inside the component tags (Ex. <func name="Kevin"></func>
2. treat 'props' as an object, we place '{ this.props.name }' in the desired location

HTML Solution:
1. go in App.js, add regular html code **IN BETWEEN** component tags (Ex. <func> ***HTML CODE*** </func>
2. go back to component class, add '{ this.props.children }' in the return statement

State

- Both state and properties influence UI control
- State is managed **within component**
- Variable declared in function (component) body
- State **CAN BE** changed

How to create state in class component?
1. create a new js component file
2.add a constructor before render():
```
constructor () {
super()
this.state = {
message = 'Hello'
}
}
```

*This constructor constructs default state value, etc.

3. If you want to change the message from Hello to Goodbye, we create changeMessage function:
```
changeMessage(){
    this.setState({
    message = 'Goodbye'
    })
}
```

4. same as class component before (functions)
5. add a button, when click, change from Hello to Goodbye
```
<button onClick = { () => this.changeMessage() }>Click!</button>
```

What is setState?
- Since we cannot modify state directly, we use 'setState' to modify state

How do we use setState?
Step 3 above.

How to pass in function in setState? (update state based on previous value)
We pass parameter in setState by

```
increment () {
this.setState((prevState) => ({
count: prevState.count + 1
}),() => {
console.log('Count is', this.state.count)
})
}
```

*As we can see, we pass in prevState inside our parameter (it is consider a function). We use it to store previous state.

setState Rules:
- **ALWAYS** use setState instead of modifying state directly
- If code must be executed after the state is updated, put the code in the **callback function**
  Callback function:

```
increment () {
this.setState((prevState) => ({
count: prevState.count + 1
}),() => {
console.log('Count is', this.state.count)
})
}
```

- When want to update state based on the previous value, pass in function and use it

Deconstructuring props and states:
- We deconstructure props and states because it is easier to use in component

  Functional Component:
  Ex. (You can use **name** directly without changing App.js file)

```
const Hello = (props) => {
const {name} = props
return (
<div className="Jello">
<h1>Hello { name } (jsx version)! </h1>
</div>
)
}
```

  Class Component:
  add ' *const {name} = this.props* ' in *render() { ... }*

Event Handling:

Button: (Handle Event)

Functional Component:

1. create a function called clickHandler:

```
function clickHandler() {
        console.log('clicked')
}
```

2. add a button
3. add '*onClick = {clickHandler}*' inside button tag (<*button onClick= {clickHandler}></button*>)

Class Component:

1. create a function called clickHandler:

```
clickHandler() {
        console.log('clicked')
}
```

2. add a button
3. add '*onClick = {this.clickHandler}*' inside button tag (<button onclick= {this.clickHandler}></button>)

How about button clicked to change state? (Official way)

1. create a function called clickHandler:
2. add a button
3. go back to state constructor, add '*this.clickHandler = this.clickHandler.bind(this)*' after this.state
4. add '*onClick = {this.clickHandler}*' inside button tag (<button onclick= {this.clickHandler}></button>)

Parents and Children? Please see in code.

Conditional Rendering

1. **if/else (Too simple)**

2. **Conditional:**

```
render() {
return this.state.isLoggedIn ?
(<div>Welcome Kev!</div> ):
(<div>Weclome Guest!</div>)
}
```

Render lists of items from an array

If we have an array of persons, including person's name, id, age, etc:
- Create two separate javascript files, one stores all names, the other one displays them

Steps:
1. create NameList.js
2. inside NameList, create an array: 'name=[{id: 1,name: 'Kev'},{id: 2,name: 'Tom'}]'
3. inside NameList, create map function:
   *const nameList = name.map(person => <**Person** key={person.id} person={person}></Person>)*
4. *return {nameList}*
5. add 'import Person from './PersonDisplay'
5. create **Person**.js
6. inside Person, create a function and passin value 'person'
7. *return {<h2>{person.name</h2>}*

Key Points:
- Always include an ID number in the array to avoid duplication
- ID number must be unique
- Key uses ID number

CSS in Component:

1. Regular CSS stylesheets
Steps:
1. create a component file called 'Stylesheet.js'
2. create a css fille called 'myStyles.css'
3. create a .changeColor {} in myStyles.css
4. import css file into component file: 'import ./myStyles.css'
5. in return, add 'className = 'changeColor' to HTML tag
6. add Stylesheet to App.js
7. Advanced feature: if you want App.js to control whether the component uses the style or not, add 'changeColor = { false/true }' in App.js
8. add props into function parameter
9. go back to Stylesheet.js, create a variable called **style**, and make if statement:
   if (props.changeColor == true), style = changeColor
   else style = ' '
10. in HTML tag, in return, add 'className = {style} ' to HTML tag

What if we want to add more styles?
In HTML tag, in return, change '*className = {`$ {style } **anotherClass**`}*'

2. Inline Styling (No need to create separate stylesheets)

***Keep in mind: all attributes follow camelCase rules and all values are strings!!**

Steps:

1. create classes before function:

```
const style1 = {
        fontSize: '72px',
        color: 'blue'
        }
```

2. apply it to HTML tag:

<h1 ***style****={style1}></h1>*

3. CSS Modules

4. CSS in JS Libraries (Bootstrap) (https://react-bootstrap.github.io/)

Steps:

1. run in commandline, 'npm install react-bootstrap bootstrap')
2. import each bootstrap if you need it in component (check link tutorial)
3. import bootstrap css file:

    ' ***import*** **'bootstrap/dist/css/bootstrap.min.css'** '

Forms

Details check hello-world folder on desktop

' *type* = "*submit*" ' is an attribute on button, which provides "submit when pressing enter key"

React Fragments

Fragments are very useful when we disrupt the natural flow of html tags.
When we return html codes, we have to use <div> to enclose all codes.
For example, a table contains lots of code together.
The div tag will disrupt the table.
Thus we use fragments. It operates the same as div tag.

Usage: ***<React.Fragment></React.Fragment>*** or **<></>** (but can't use keys)

Key Points:
- You can use Key in react fragment
- can use the shorthand representation, but can't use keys

Pure Components

- pure component does not re-render every 2 second
- it preforms shallow comparison on props and state
- component not re-render = performance boost
- ensure all children components are also pure component
- never mutate the state. Always return a new object that reflects the new state
- always use regular class components unless bad performance
- only works with class components, not functional components

React Memo

- same property as pure components, but it used for functional components
- '***export default React.memo(MemoComp)***'

React Refs

- Ref focus text input field when we refresh the webpage
- more detail please visit ref code in hello world folder
- Ref forwarding

React Portal

- Since we always place code in App.js (id=root), we can change where we want to place codes
- It is useful to create pop-ups for websites

Higher Order Components (HOC)

- share the same code (functionality) that can be used in many components
- a function takes a component as an argument and return a new component
- const Ironman = withSuit(TonyStark)
- enhance a component to a better component
- it can also maintain state

****When creating HOC, PLEASE remember to pass down rest of the props!!! (Ex. from App.js) include this code in the return statement:**

' *{... this.props}* '

Render Props

- A technique for **sharing code** between React Components using a **prop** whose value is a **function**!

Context

- provide a way to pass data through the component tree **without** having to pass props down manually at every level
- UserProvider and UserConsumer

HTTP Request

- Use axios to implement get and post
- npm install axios