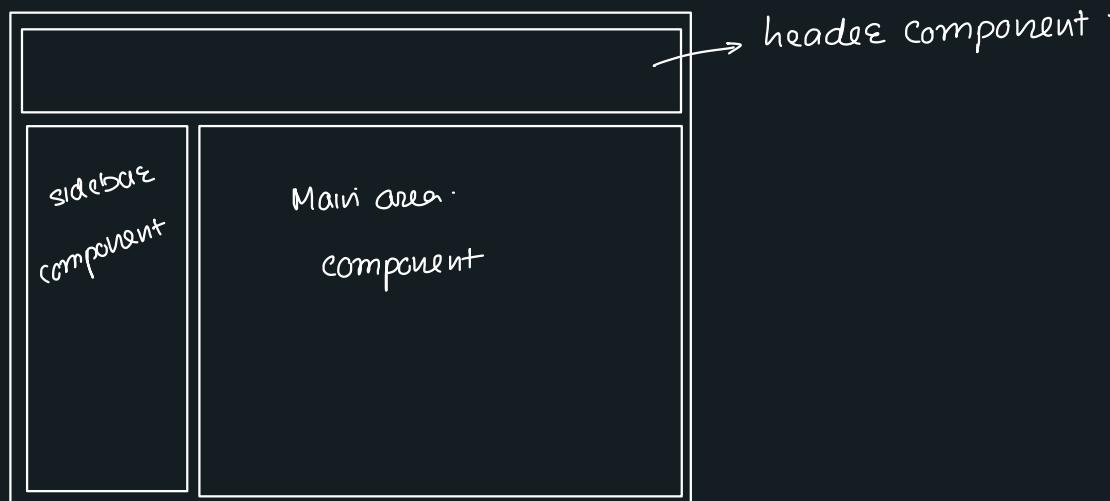


React JS

What is React?

a Javascript library for building User Interfaces
↓
Components

any webpage can be seen as composed of components



Why to think in terms of component?

When we use component based approach we don't build a single big app

Rather we build all these components individually and glue them together

which then appears as a single application

Advantages of using components :

① Easy to work in teams

②) Keep code Manageable and Maintainable

③) Component Reusability

Why React ?

- UI state becomes difficult to manage with Vanilla JavaScript.
- Focus on Business logic and not on development.
- Huge Ecosystem, High performance, Active community.
- Backed by Facebook

React Alternatives

- Angular
- Vue
- Backbone
- jQuery (Not so much)

Two Kinds of Applications

With all these frameworks we can build two types of applications

- Single Page Application
- Multi Page Application

SPA: we only get back one HTML page from server. Content is rendered on client using framework

Entire page is managed by React.

Typically only one ReactDOM.render() call.

Popular way of building webapp these days

MPA: Multiple HTML pages are sent by server on different routes.

although these pages can contain React components along with normal HTML CSS. But the page is not controlled by React.

There is one ReactDOM.render() call per widget

Next Generation JavaScript :-

• Understanding let and const

Two keywords for creating variables.

constant → variables whose value never changes (single assignment)

let → variables that allows changes in value.

• 2) ARROW FUNCTIONS

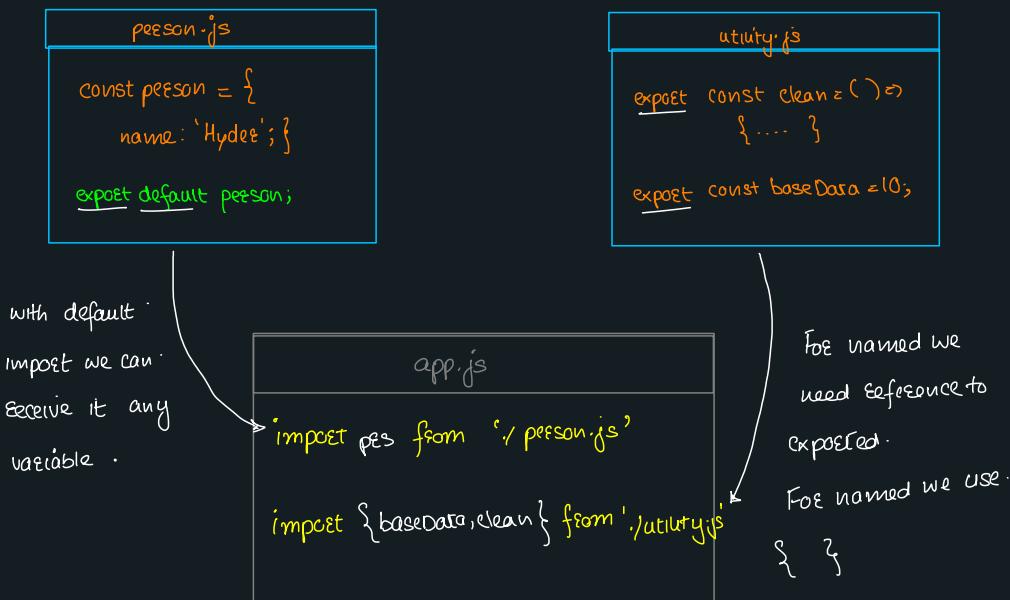
```
function HelloWorld(name){  
    console.log(name)  
}
```

```
const HelloWorld = name =>  
    console.log(name)
```

• Exports and Imports (Modules):

These are two types of exports

- ① Default Exports
- ② Named Exports



with default export we can choose any name while importing eg.

`import pes from './person.js'`

→ NOTE: with named exports we strictly have to use exact names as declared in the file. eg

`import {clean} from './utility.js'`

However we can use Aliases as follows

`import { clean as cln } from './utility.js'`

Classes: are blueprints for objects. Classes contain properties

and methods.

NOTE: methods are created.

only using method
name.

```
class Person {  
    name: 'Hydee' → property: value.  
    method → call = () => { ... }  
}
```

USAGE: const Hydee = new Person()

Classes, properties and Methods:

There is a more modern syntax that spares use of constructor

function

ES6

```
class XYZ {  
    constructor() {  
        this.propertyName = 'value'  
        ...  
    }  
    myMethod() { ... }  
}
```

ES7.

```
class XYZ {  
    propertyName = value;  
    myMethod = () => { ... }  
}
```

new syntax uses Arrow function. Advantage

of using Arrow Function is that it
eliminates problems with this keyword.

• Spread and Rest Operators:

actually both spread and rest are same operators. 3 dots "..." It is named differently based on situation it is used.

Spread → used to split up an array or object properties

eg const newArray = { ... oldArray, 1, 2 }

const newObj = { ... oldObj, newProp: 5 }

Rest Operator → used to merge a list of function arguments into an array.

```
function showArgs(...args) {  
    console.log(args)  
}
```

showArgs(1, 2, 3, 4, 5)
→ output: [1, 2, 3, 4, 5]

• Destructuring: allows to extract array elements or object properties and store them in variables

Array Destructuring

```
let [a, b] = [10, 20]
```

// a = 10

// b = 20

Object Destructuring

let {
 ↓
 name
} = {
 ↓
 name: "Hyde",
 age: 23
}
variable name must match obj property.
// name = Hyde

• References and Primitive Types

numbers strings booleans are primitive

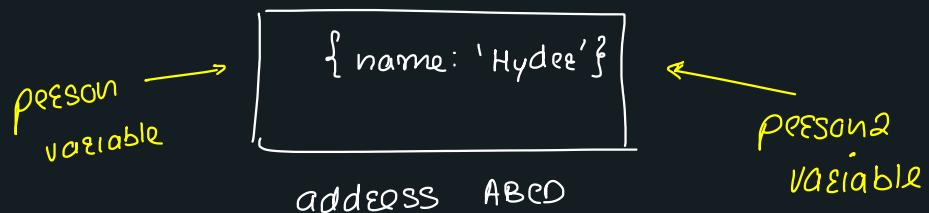
objects and Arrays are non primitive

For non primitive types assignment means referencing

eg const person = { name : " Hydee" };



const person2 = person; By this assignment person2 will also refer to same memory address



Therefore any change made will reflect by both variables

Since both of them are pointing to same memory location.

Thus objects should not be copied using assignment but using spread operator.

$\therefore \text{newObject} = \{\dots \text{oldObject}\}$

④ Array Functions

④ map → building array method that returns a new array

let numbers = [1, 2, 3]

let doubled = numbers.map(num => num * 2);

console.log(doubled)

doubled // [2, 4, 6]

Basics : Core React Concepts

Building a Workflow :

- Dependency Management Tool : npm or yarn
- Bundler : Webpack → write Modular code & bundle it together
- Compiler : Babel + presets [change NextGen JS to ES5]
- Development Server

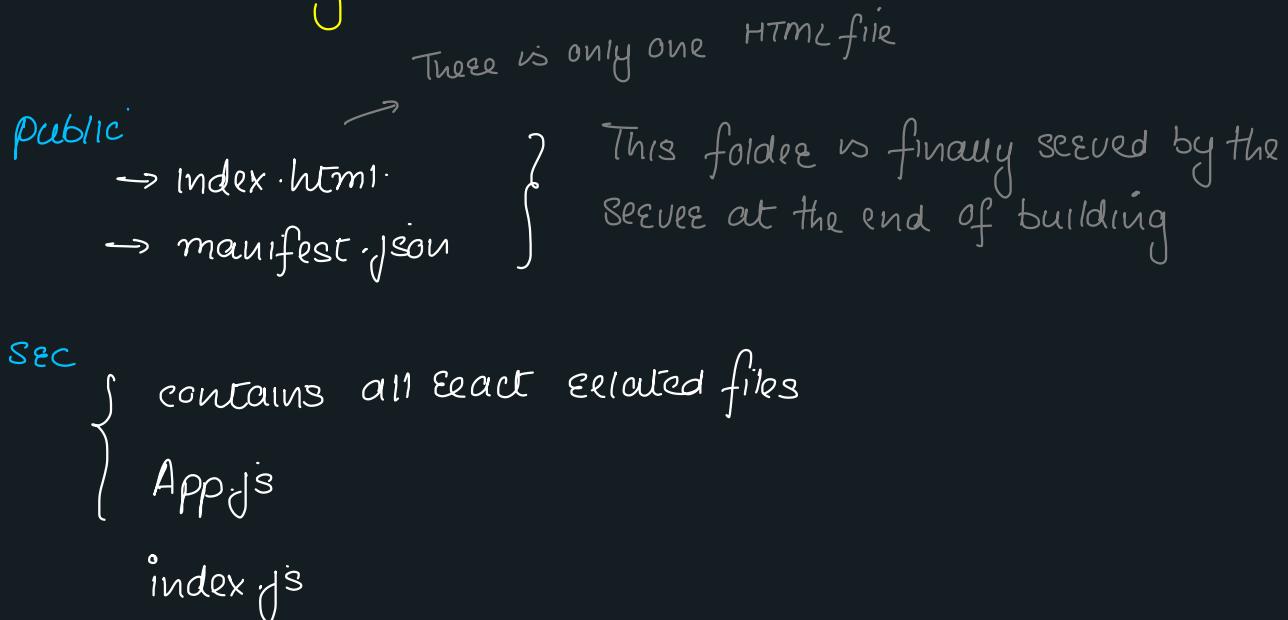
• Using create-react-app:

It is a tool maintained by Facebook and community that automates entire workflow discussed above.

- Install NodeJS

- `npx create-react-app myapp`
 - ↳ creates files & structure directly in the directory
 - ↳ creates a new directory called myapp and then creates structure

④ Understanding React Folder Structure:



⑤ Component Basics

```
import React, { Component } from 'react'
import './App.css'

class App extends Component {
  render() {
    return (
      

# Hello World


    )
  }
}

export default App;
```

JSX code
Syntax sugar
converted to
JS automatically

every class based component needs to have
at least a render method. The render
method should return either HTML or
null.

→ This is exported so that it can be used throughout the project

• Understanding JSX:

```
<div className='App'>  
  <h1> Hi there </h1>  
</div>
```

```
React.createElement('div', {className: 'App'},  
  React.createElement('h1', null, 'Hi there'))
```

JSX

compiled to

Normal Javascript

There are certain restrictions while using JSX eg we cannot use reserved keyword of class is reserved in JavaScript thus we use className.

- It should return a single root element.

• Creating a Functional Component:

```
import React from 'react'; ① import React
```

```
const Person = () => { ② } ③ component logic  
  return (  
    <div>  
      <h1> Hello World </h1>  
    </div>  
  )
```

```
export default Person; ④ export component
```

allows us to pass data from parent (wrapping) element

- Working with props: to child (embeded) component.

```
const Person = (props) => {
```

```
    return (
```

```
        <p> My name is {props.name} </p>.  
    );
```

```
}
```

```
<Person name='Hydee' />
```

NOTE: for class based components
props is accessed via this.props

understanding children:

props object has a special property called as children

```
<Person>  
    <h1> Hello World </h1> } children  
</Person>
```

Understanding and using STATE:

class has properties. In any component which extends Components we can use special property called state as following

```
class App extends Component {  
    state = {  
        name: 'Hydee', age: 23 }
```

```

  render() {
    return (
      <div>
        <p> My name is {this.state.name} </p>
      </div>
    )
  }
}

```

accessing state

Note: if state changes it will lead React to re-render entire component.

```

① import React from "react";
class Person extends React.Component {
  state = {
    name: "Hyder",
  };
  clickHandler = () => {
    this.setState({ name: "Me" });
  };
  render() {
    return (
      <div>
        <p> My name is {this.state.name}</p>
        <button onClick={this.clickHandler}> Change Text </button>
      </div>
    );
  }
}
export default Person;

```

event handlee

event *reference* *to handlee*

- for handling events we only pass reference to handlee function
- Handlee functions are usually ARROW Functions to avoid this based issues.

- State Management in Functional Components using Hooks:

prior to version 16.8 of React, Class based components were only possible way of having state inside React Components.

In case of functional components we use Hooks to manage state.

```
import React, { useState } from 'react';
```

```
const Person = props => {
```

```
const [personState, setPersonState] = useState({ name: "Hyder" })
```

What it actually does is that it creates a variable called as personState. Initially personState is initialized to value passed to useState Hook as argument. That is, personState = {name: "Hyder"}

for changing the state we use setPersonState function

```
import React, { useState } from "react";
```

① Importing React and useState Hook.

```
export default function Person2() {
```

```
let [name, setName] = useState("Hyder");
```

② useState Hook

name variable is initialized to value 'Hyder'

```
const clickHandler = () => {
```

```
setName("Someone");
```

③ Event Handler

```
};
```

```
return (
```

```
<div>
```

```
  <p>{name}</p>
```

```
  <button onClick={clickHandler}>Change</button>
```

```
</div>
```

```
);
```

```
}
```

→ using setName function we can modify of state name equal to argument passed.

④ handle Response

Stateful vs Stateless Components:

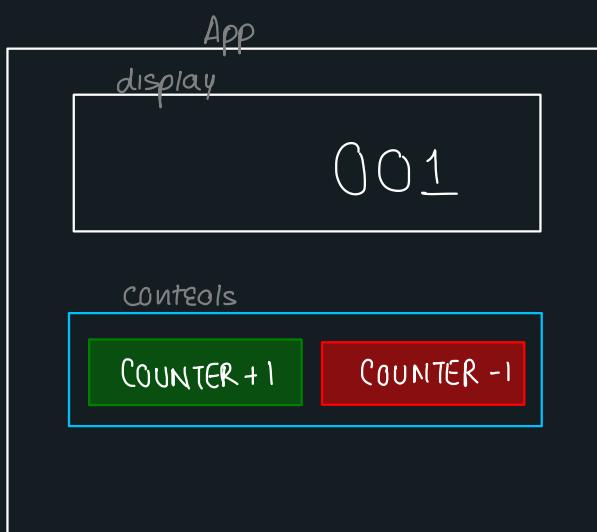
Components either class based or functional which have internal state are called smart components / containers.

While as components with no internal state are called DUMB or presentational components.

We should have more stateless components & few statefull components

• Passing Method References between Components:

Suppose our React app follows below described structure



App component contains state which keeps track of current counter value. This state is passed to display component as prop.

However counter handles for changing the state of current counter value needs to be defined in App and reference to these handles to be passed to controls component via props

```

src > App.js > App > countUpHandler
1 import React, { useState } from "react";
2 import logo from "./logo.svg";
3 import "./App.css";
4
5 import Display from "./components/display";
6 import Controls from "./components/controls";
7
8 function App() {
9   let [count, setCount] = useState(0);
10
11   let countUpHandler = () => [
12     setCount(count + 1);
13   ];
14
15   let countDownHandler = () => [
16     setCount(count - 1);
17   ];
18
19   return (
20     <div className='App'>
21       <Display count={count} />
22       <Controls
23         up={() => [
24           countUpHandler();
25         ]}
26         down={() => [
27           countDownHandler();
28         ]}
29       />
30     </div>
31   );
32
33 export default App;
34

```

```

src > components > display.js > ...
1 import React from "react";
2
3 export default function display(props) {
4   return (
5     <div>
6       <input className='display' disabled='true' value={props.count} />
7     </div>
8   );
9 }
10

```

```

src > components > controls.js > controls
1 import React from "react";
2
3 export default function controls(props) {
4   return (
5     <div className='controls'>
6       <button onClick={props.up} className='counter_up'>
7         Counter + 1
8       </button>
9       <button onClick={props.down} className='counter_down'>
10        Counter - 1
11      </button>
12    </div>
13  );
14 }
15

```

① handlee fn is defined inside stateful component.

② it is wrapped in an arrow fn and passed as prop

③ event is connected to prop reference.

```

src > components > controls.js > controls
1 import React from "react";
2
3 export default function controls(props) {
4   return [
5     <div className='controls'>
6       <button onClick={props.up} className='counter_up'>
7         Counter + 1
8       </button>
9       <button onClick={props.down} className='counter_down'>
10        Counter - 1
11     </button>
12
13     <input
14       onKeyPress={(e) => [
15         if (e.key === "Enter") {
16           let value = parseInt(e.target.value);
17           props.textInc(value);
18         }
19       ]}
20     />
21   </div>
22 ];
23 }
24

```

```

4
5 import Display from "./components/display";
6 import Controls from "./components/controls";
7
8 function App() {
9   let [count, setCount] = useState(0);
10
11   let countUpHandler = () => [
12     setCount(count + 1);
13   ];
14
15   let countDownHandler = () => [
16     setCount(count - 1);
17   ];
18
19   let textIncHandler = (value) => [
20     setCount(count + value);
21   ];
22
23   return (
24     <div className='App'>
25       <Display count={count} />
26       <Controls
27         up={() => [
28           countUpHandler();
29         ]}
30         textInc={(value) => [
31           textIncHandler(value);
32         ]}
33         down={() => [
34           countDownHandler();
35         ]}
36       />
37     </div>
38   );
39 }
40
41 export default App;

```

• Adding Styling

There are two ways to style components.

- Global Styling with CSS files . All these CSS files are globally scoped . However we do need to import it in JS so that Webpack includes it in the bundle .

• Inline Styles :

```
1 import React from "react";
2
3 export default function controls(props) {
4   const styles = {
5     counterUp: {
6       backgroundColor: "green",
7       color: "white",
8     },
9
10    counterDown: {
11      backgroundColor: "red",
12      color: "white",
13    },
14  };
15
16  return (
17    <div className='controls'>
18      <button
19        style={styles.counterUp}
20        onClick={props.up}
21        className='counter_up'>
22          Counter + 1
23        </button>
24      <button
25        style={styles.counterDown}
26        onClick={props.down}
27        className='counter_down'>
28          Counter - 1
29        </button>
30
31      <input
32        onKeyPress={(e) => {
33          if (e.key === "Enter") {
```

{ ① Inline styles }

② using Inline styles .

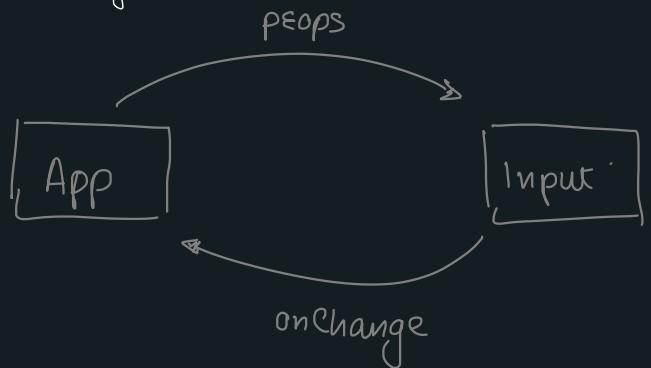
JSX provides a special property called as style . This can be used to apply above defined styles to an element in JSX

• Two way binding

```
function App() {
  let [name, setName] = useState('Hyder')
  const handleChange = (newValue) => {
    setName(newValue)
  }
  return (
    <div>
      <input value={name} onChange={handleChange}>
    </div>
  )
}
```

```
function Input(props) {
  return (
    <input value={props.value} onChange={(e) => props.handleChange(e.target.value)}>
  )
}
```

whenever input changes onChange event will fire which intent will change state of App. Since state of App component changes, props passed to Input component will also change.



• Rendering Content Conditionally

```
class App extends React.Component {
  state = { show: false }

  render() {
    return (
      <div>
```

```
{ this.state.show ? <p> Hello World </p> : null }
```

```
<p> Does work </p>
```

```
</div>
```

```
)
```

```
}
```

{ } contains JS expressions which are evaluated to a single value. However we cannot use if else directly inside { }. In this case we are using Ternary operator.

- Better approach to conditionally rendering

For large and more complicated apps this approach is not very optimal.

We can create a function that is responsible for rendering that part and call that function inside return of JSX. eg.

```
function Person (props) {
```

```
let [show, setShow] = useState (false)
```

```
→ function showPerson () {
```

```
if (show) return (<p> Hello World </p>)
```

```
else return null
```

```
}
```

```
return (<div>
```

```
{ showPerson () } → calling function.
```

```
</div>
```

```
)
```

```
}
```

Method responsible for conditional rendering

• Rendering a list of elements using map:

We can render a list/array using map function as shown below:

src > components > `display.js` > `Display` > `renderList` > `people.map() callback`

```
1 import React, { useState } from "react";
2
3 export default function Display(props) {
4     let [show, setShow] = useState(true);
5     let [people, setPeople] = useState([
6         {
7             name: "Hyder",
8         },
9         {
10            name: "Jan",
11        },
12        { name: "Manan" },
13    ]);
14
15     let renderList = () => {
16         if (show) {
17             return people.map((person) => {
18                 return (
19                     <div>
20                         <p> {person.name} </p>
21                     </div>
22                 );
23             });
24         } else return null;
25     };
26
27     return (
28         <div>
29             {renderList()}
30             <input class='display' disabled='true' value={props.count} />
31         </div>
32     );
33 }
34 }
```

people holds an array of objects

renderList method is responsible for dynamically generating JSX for list using map fn.

JSX returned for each element of array.

- Updating State Immutably

We should never set state directly but always use functions such as `useState` or function declared during state hook to set state only. The reason for this restriction is that React uses them to re-render the components on state change.

If we directly set state there will be no component re-rendering also we should copy state using spread operator. Since state is a JS object assignment will only create another reference to state object and not copy it.

- List and Keys

For lists there is a special property called Key. Key for each element is unique. It uses key to allow React to efficiently modify virtual DOM rather than re-rendering entire list which for very large lists can be very inefficient.

Styling React Components

Most basic limitation of using inline style is that we cannot use Pseudo classes such as Hover , after etc . Now we can use it via CSS file but then it would be globally scoped .

Radium : is a 3rd party package for React that allows us to overcome restrictions of inline styles . Allows us to use Pseudo Selectors , media queries

- `npm i --save radium` → installs Radium package
- import Radium from 'radium'
- wrap component in Radium()

```

src > components > controls.js > controls
1 import React from "react";
2
3 import Radium from "radium"; -① import Radium
4
5 function controls(props) {
6   const styles = {
7     counterUp: {
8       backgroundColor: "green",
9       color: "white",
10      transition: "transform all .2s",
11      ":hover": {
12        transform: "scale(0.8)",
13      },
14    },
15
16    counterDown: {
17      backgroundColor: "red",
18      color: "white",
19    },
20  };
21
22  return [
23    <div className='controls'>
24      <button
25        style={styles.counterUp}
26        onClick={props.up}
27        className='counter_up'>
28          Counter + 1
29        </button>
30      <button
31        style={styles.counterDown}
32        onClick={props.down}
33        className='counter_down'>
34          Counter - 1
35        </button>
36
37      <input
38        onKeyPress={(e) => {
39          if (e.key === "Enter") {
40            let value = parseInt(e.target.value);
41            props.textInc(value);
42          }
43        }}
44      />
45    </div>
46  ];
47}
48
49 export default Radium(controls); ② export component wrapped in Radium
50

```

This shows we need to wrap
pseudo selectors in quotation marks
This is equivalent to:

```

  .counterUp :hover {
    transform: scale(0.8);
  }

```

② export component wrapped in Radium

Introducing Styled Components:

Styled components library for styling components with ease

① npm i --save styled-components

② import styled from 'styled-components';

const Button = styled.button` → regular JS feature called
Tagged Templates

understanding `styled.button``` syntax:

Here styled is an object and button is a method. instead of having parenthesis we are having backticks in which we can pass text which is passed into button method.

This styled object that we are importing has a method for each HTML element eg

`styled.button```

`styled.h1```

`styled.p```

etc ..

Code Example of Styled Components:

```
src > components > counter_styled.js > Counter_Styled > IncreaseBtn
1 import React, { useState } from "react";
2
3 import styled from "styled-components"; -① importing styled components
4
5 function Counter_Styled() {
6   let [count, setCount] = useState(0);
7
8   let Div = styled.div` -② Div is a div with all these styling preattached
9     margin: auto auto;
10    margin-top: 60px;
11  `;
12
13   let Counter_Display = styled.div` -③ Counter-Display is a div which is
14     font-size: 20px;
15     color: gray;
16   `;
17
18   let IncreaseBtn = styled.button` -④ Pseudo selector
19     background-color: green;
20     padding: 20px;
21     color: white;
22
23     :hover { -⑤
24       transform: scale(0.8); } } };
25   `;
26
27   return (
28     <Div> @
29       <Counter_Display>{count}</Counter_Display> ⑥
30
31       <div>
32         <IncreaseBtn ⑦
33           onClick={(e) => {
34             setCount(count + 1);
35           }}>
36           Increase Counter
37         </IncreaseBtn>
38       </div>
39     </Div>
40   );
41 }
42
43 export default Counter_Styled;
```

instead of using JSX we use variables which are generated by styled components as marked by @ , ⑥ , ⑦ , ⑧

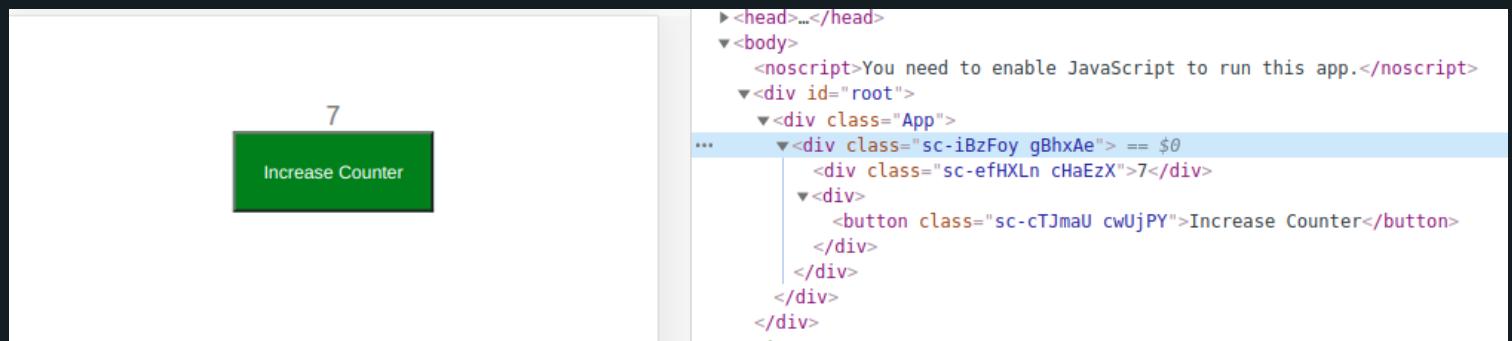
How Styled components work internally:

using previous Counter code we wrote as styled component we will

try to understand how styled components works internally

if we inspect our react app using developer tools, we will notice

elements with strange class names attached to them



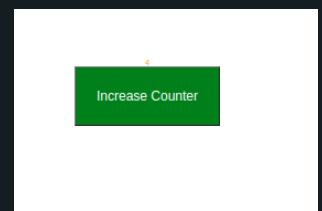
These classes are generated by styled components library.
These classes are defined in the head of HTML.

Thus style components takes one CSS that we pass in backticks places in `<style>` tag in the head section with a unique name. It just adds elements with appropriate classes.

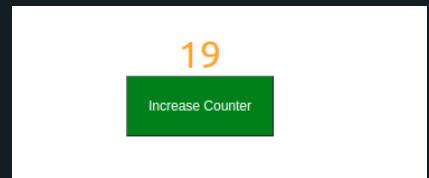
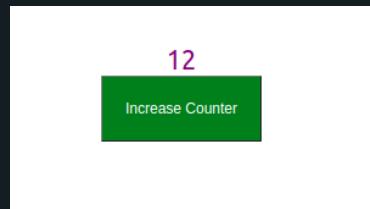
→ styles created inside head section of HTML

Dynamic Styles with Styled Components:

We use props to control dynamic styles. We pass props to component. This prop is then accessible inside backticks that we attach to that particular component.



Similarly the value of count smaller is the size and color also differs



```
src > components > counter_styled.js > Counter_Styled
1 import React, { useState } from "react";
2
3 import styled from "styled-components";
4
5 function Counter_Styled() {
6   let [count, setCount] = useState(0); } → initializing count state
7
8   let Div = styled.div` margin: auto auto; margin-top: 60px;
9   `;
10
11 let Counter_Display = styled.div` font-size: ${props => {
12   return props.count * 2; }}px;
13
14 color: ${props => {
15   if (props.count > 5 && props.count < 10) {
16     return "green";
17   } else if (props.count > 10 && props.count < 15) {
18     return "purple";
19   } else return "orange";
20 }}`;
21
22
23 let IncreaseBtn = styled.button` background-color: green; padding: 20px; color: white;
24
25 :hover {
26   transform: scale(0.8);
27 }
28
29 return (
30   <Div>
31     <Counter_Display count={count}>{count}</Counter_Display>
32   </Div>
33   <div>
34     <IncreaseBtn
35       onClick={(e) => {
36         setCount(count + 1);
37       }}>
38       Increase Counter
39     </IncreaseBtn>
40   </div>
41 )
42
43
44
45
46
47
48
49
50
51
52
53 }
```

→ dynamically assigning font size based on count value which is passed via props to component.

→ similarly for dynamically assigning color.

→ passing count as prop to this styled component. value of this prop determines color and font-size.

SYNTAX

property : \${ (props) => { logic ... return value; }}

CSS Modules :

- CSS scoped to file or components . Rather than bloating JS with CSS .

To use CSS Modules there is a need to tweak build configuration we need .

We need to eject from react by npm run eject . This will expose all the underlying configurations of webpack present in config folder .

- we need to make changes in two files

webpack.config.dev.js

webpack.config.prod.js

```
{ test : /\.css$/,
use : [
  styleLoaders.resolve ('style-loader'),
  { loader : styleLoaders.resolve ('css-loader') },
  options : {
    importLoaders : 1,
    modules : true,
    localIdentName : '[name]__[local]___[hash:base64:5]' } ] }
```

enables modules

} we need to add these two lines

provides a unique name to classes in DOM

Now it will also us to import from CSS in another manner.

import classes from '`./App.css`'

`App.css`

inside JS

`classes.es6`

Thus CSS Modules allow us to

`.es6 {`

`}`

`.blue`

use classes that are scoped to component level only.

Now there are different ways of including CSS in React

- ① Radium
- ② Styled Components
- ③ CSS Modules

NOTE: for React scripts greater than 2.0 we donot need to eject for using module CSS. Rather we can use directly by changing syntax.

① Naming CSS files: `XY2.module.css`

② Importing CSS in JS: `import classes from './XY2.module.css'`

This allows us to unlock CSS modules without ejecting

• Using Global Styles with CSS Modules:

We can define a global CSS class (means it will not be transformed) by using the following Syntax

```
:global .Post { ... }
```

This makes it global in nature.

we can use a global style directly in any component as

```
<Component className="Post" />
```

Example using CSS Modules without ejection:

```
App.js          display.module.css      Display.js X
src > components > css_module_counter > Display.js > Display
1  import React from "react";
2
3  import styles from "./display.module.css"; -① importing styles
4
5  export default function Display({ count }) {
6    console.log(styles);
7    return (
8      <div className={styles.display}>
9        <input className={styles.display_input} value={count} disabled />
10     </div>
11   );
12 }
13
```

```
App.js          display.module.css      Display.js ●
src > components > css_module_counter > display.module.css > .display
1  .display {
2    padding: 30px;
3    background-color: aquamarine;
4    text-align: center;
5  }
6
7  .display_input {
8    font-size: 25px;
9    padding: 10px;
10   margin: auto auto;
11   text-align: center;
12 }
```

③ writing Normal CSS saving as name.module.css

Debugging React Applications:

- Understanding Error Messages :-

React shows errors in console by showing line number where error has occurred. These are syntactical errors which are easier to debug. However errors also do occur due to incorrect logic and are called as Logical Errors.

- Finding Logical Errors using dev tools & Source Maps :-

Sources → debugger breakpoints

- React Developer Tools:- 3rd party extension available for chrome.

Downloadable from chrome web store.

- Using Error Boundaries

Error Boundaries are higher order components with sole job of handling errors. ErrorBoundary wraps components inside it.

Error Boundaries should be used only in cases where we know code can fail. This is used to show custom error messages.

• Implementing Error Boundaries :

- 01) Create a new folder 'ErrorBoundary' inside src folder.
- 02) Create a new ErrorBoundary.js component as following

```
src > ErrorBoundary > JS ErrorBoundary.js > ...
1 import React, { Component } from "react";
2
3 class ErrorBoundary extends Component {
4     state = {
5         hasError: false,
6         errorMessage: "",
7     };
8
9     componentDidCatch(error, info) {
10         this.setState({ hasError: true, errorMessage: error });
11     }
12
13     render() {
14         if (this.state.hasError) {
15             return <h1>{this.state.errorMessage}</h1>;
16         } else return this.props.children;
17     }
18 }
19
20 export default ErrorBoundary;
```

catches the error from all the descendants of child components

→ serves errorMessage

→ else serves all the children components that are wrapped by ErrorBoundary Component.

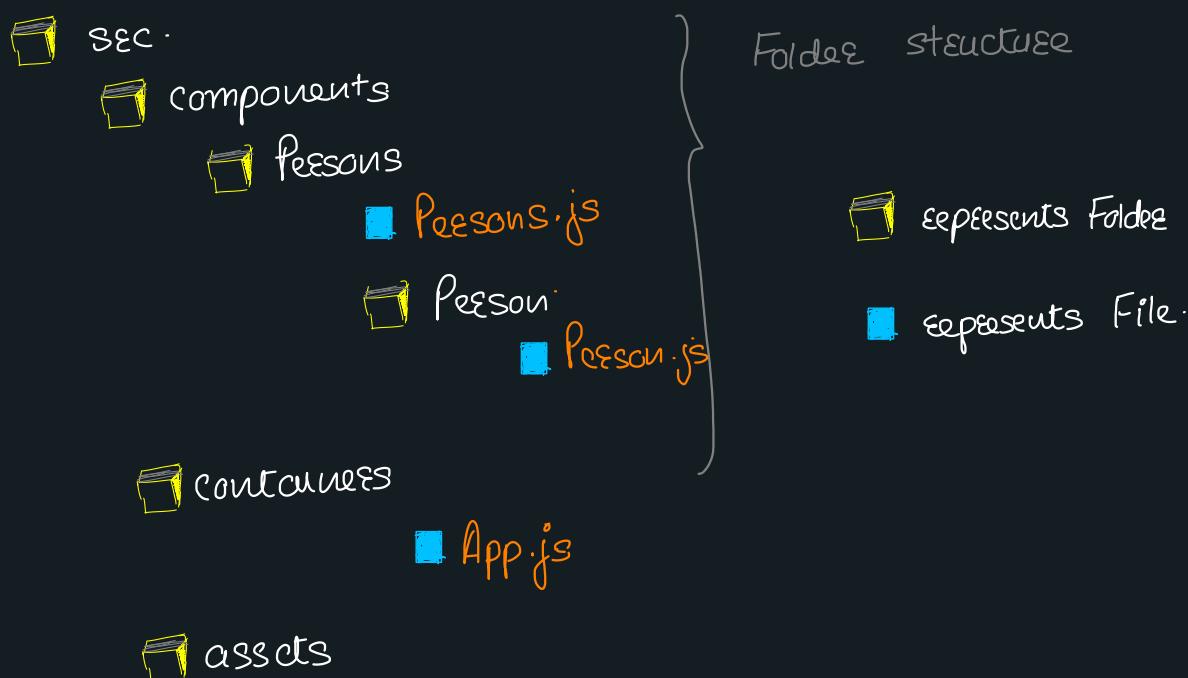
```
src > JS App.js > ...
1 import React, { useState } from "react";
2
3 import Display from "./components/css_module_counter/Display";
4
5 import ErrorBoundary from "./ErrorBoundary/ErrorBoundary";
6
7 export default function App() {
8     let [count, setCount] = useState(0);
9
10    setTimeout(() => {
11        setCount(count + 1);
12    }, 1000);
13
14    function renderDisplay() {
15        if (count > 10) {
16            throw new Error("Count Greater than 40");
17        } else return <Display count={count} />;
18    }
19
20    return <ErrorBoundary>{renderDisplay()}</ErrorBoundary>;
21
22 }
```

- Deeper dive into React Internals and Components:-

→ Components Deep Dive

Project Structure:-

Usually components which are responsible for state Management should not deal with too much JSX. It means for stateful components endee method should be clean.



Comparing Stateless and Stateful Components

Stateful component is one which manages state.

Historically class based component referred to stateful while as functional component meant dumb or presentational components.

Generally there should be a lot of dumb or presentational components. If we have fewer and fixed number of stateful components then understanding flow of data becomes easy.

This also allows us to reuse presentational components throughout project without any interlaces.

① Class based vs Functional Components

→ class based components: are simple javascript classes that

extends React.Component class. Historically these components

were only way to manage state in a component.

② Access to State: class based components have direct access to state. Sets state using setState()

- **Lifecycle Hooks:** have access to lifecycle hooks by default.
- **Access State and Props:** Class based component uses this keyword because in case of class based components STATE and PROPS are properties of class.
- **Functional Components:** are Javascript functions that receive props as an argument by default.
 - **Access to state:** is via useState hook.
 - **Lifecycle Hooks:** are still not perfectly supported.
 - **Access to State and Props:** is via useState() hook and props is directly accessible as an argument.

When to use what: although the diff b/w class based & functional components is narrowing day by day. But still we can choose based on following

- using a version of React which does not support hooks we will have no choice but to use class based components

- if lifecycle hooks are required we still need to use class based

- Understanding Component Lifecycle:

First takeaway is that it is only available in class based components

Functional components have only an equivalent of lifecycle available using Hooks.

class based lifecycle Methods: Any React class based component will have following methods available. React automatically executes these methods at certain point of time

- constructor()
- shouldComponentUpdate()
- componentDidUpdate()
- componentDidMount()
- render()
- getDerivedStateFromProps()
- getSnapshotBeforeUpdate()
- componentDidCatch()
- componentWillUnmount()

Component Lifecycle - Creation :-

When a React Component is created then first of all constructor() method is executed.

①

constructor(props)

actually constructor is not

React lifecycle Hook but simple JS.

- Incase if we donot mention constructor method. It is called automatically.
- by default. However if we explicitly mention constructor call then.
- we also need to call super(props) as first statement of constructor.
- constructor is to be used for initialization eg setting up state.
 - However it should not be used for AJAX calls or local storage etc. called SIDE EFFECTS.

②

getDesieredStateFromProps (props, state)

runs after constructor.

USECASE: whenever props change, we can sync state to them.

③

render()

endee method should be used to prepare & structure JSX code.

④

Render Child Components

renders all child components

and once all these lifecycle hooks finish ComponentDidMount executes

⑤

componentDidMount()

is a very important lifecycle hook

can be used for SIDE EFFECTS eg reaching out to web, causing

HTTP SEEURE.

NOTE: state should not be updated in this lifecycle Method because it will cause re-rendering.

However state can be set asynchronously. When response is received

- Component lifecycle - Update (Props Change)

getDerivedStateFromProps (props, state)

→ syncs state to props

shouldComponentUpdate (nextProps, nextState)

may cancel updating process
decides whether to continue update or not.

render()

propagates & structures JSX

updateChild Component Props

getSnapshotBeforeUpdate (prevProps, prevState)

last minute Dom operations

componentDidUpdate()

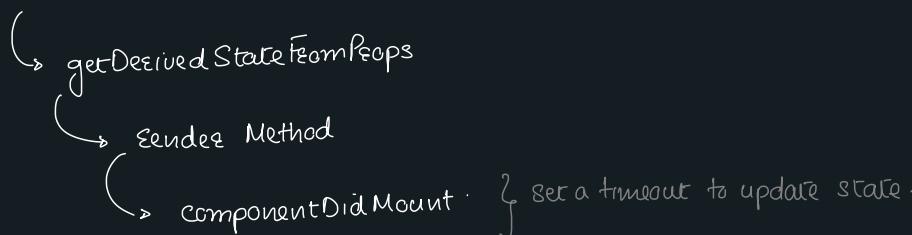
used to perform side effects
eg. API requests, local storage
Access etc.

Note: we should not try to set state in componentDidMount directly because it will lead to re-rendering.

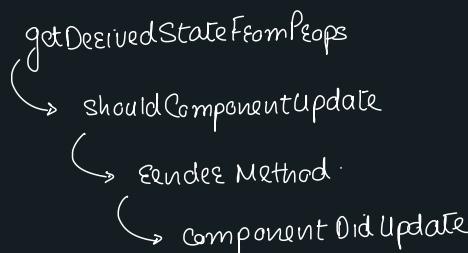
However state can be set Asynchronously.

• Component Update Lifecycle - State Changes

constructor



• STATE UPDATED



```
index.js App.js <div>Hello from App.JS - Count {this.state.count}</div>
```

```
src > containers > App.js > ...
1 import React, { Component } from "react";
2
3 class App extends Component {
4   constructor(props) {
5     super(props);
6
7     console.log("[App.js] Constructor Running!");
8
9     this.state = {
10       count: 0,
11     };
12   }
13
14   static getDerivedStateFromProps(props, state) {
15     console.log("[App.js] getDerievedStateFromProps Running!", state);
16
17     return null;
18   }
19
20   componentDidMount() {
21     console.log("[App.js] componentDidMount");
22
23     setTimeout(() => {
24       this.setState({ count: this.state.count + 1 });
25     }, 3000);
26   }
27
28   componentDidUpdate() {
29     console.log("[App.js] ComponentDidUpdate");
30   }
31
32   shouldComponentUpdate(nextProps, nextState) {
33     console.log("[App.js] shouldComponentUpdate", nextState);
34     return true;
35   }
36
37   render() {
38     console.log("[App.js] Render Method Running!");
39
40     return <div>Hello from App.JS - Count {this.state.count}</div>;
41   }
42 }
43
44 export default App;
```

```
[HMR] Waiting for update signal from WDS...
[App.js] Constructor Running!
[App.js] getDerievedStateFromProps Running! ▶ {count: 0}
[App.js] Render Method Running!
[App.js] componentDidMount
[App.js] getDerievedStateFromProps Running! ▶ {count: 1}
[App.js] shouldComponentUpdate ▶ {count: 1}
[App.js] Render Method Running!
[App.js] ComponentDidUpdate
```

• Lifecycle Methods in Functional Components

Now for functional components we don't have access to lifecycle methods

But we use an equivalent Hooks for the same.

React Hooks is supported from React 16.0.8

These are different React hooks that exist as useState, useEffect.

• useEffect React Hook

is second most important React Hook after useState.

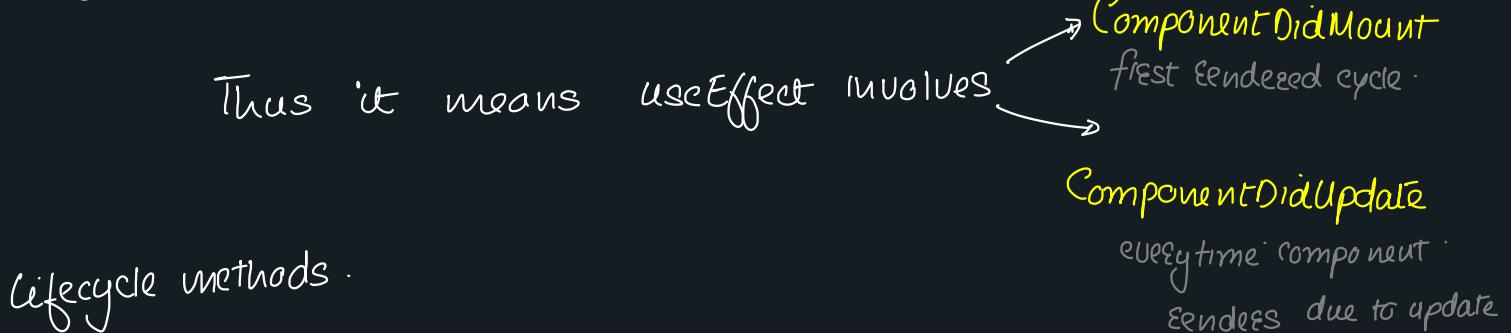
useEffect combines the functionality of usecases of all lifecycle methods in a single React Hook

• Working of useEffect React Hook

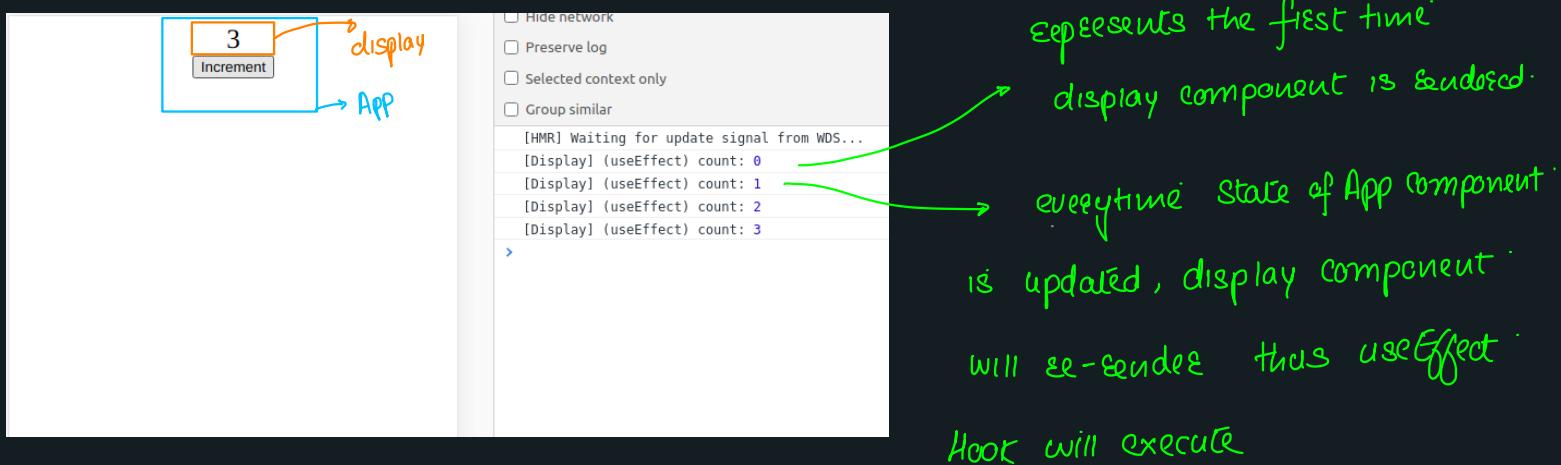
• useEffect Hook takes in a function

• This function runs or is executed everytime component is rendered.

Thus it means useEffect involves



Example on useEffect :



Important

- Controlling useEffect Behaviour :

As we have already mentioned that useEffect is combination of certain lifecycle methods. eg Lifecycle Methods such as componentDidMount & componentDidUpdate.

& componentDidUpdate .

As default behaviour useEffect will execute everytime component is rendered .

To control the behaviour of useEffect it has a dependency .

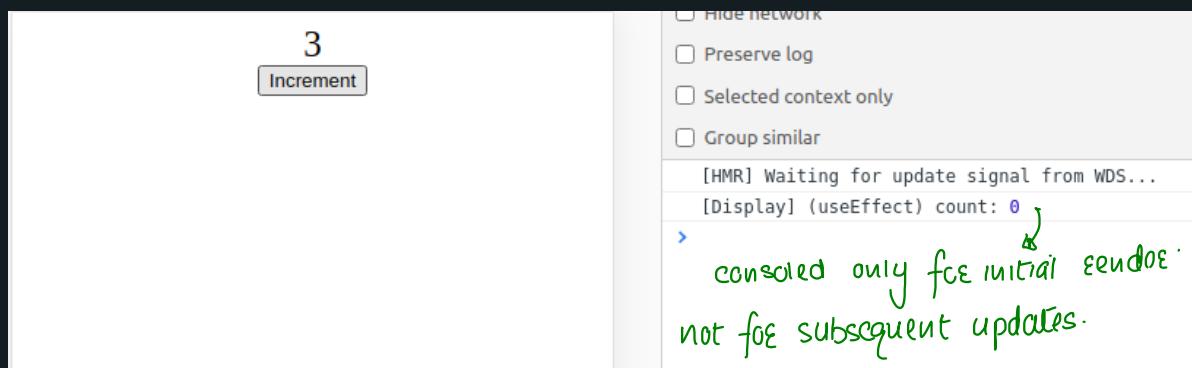
list in the form of an array .

`useEffect(() => { }, [])`

call back function dependency list

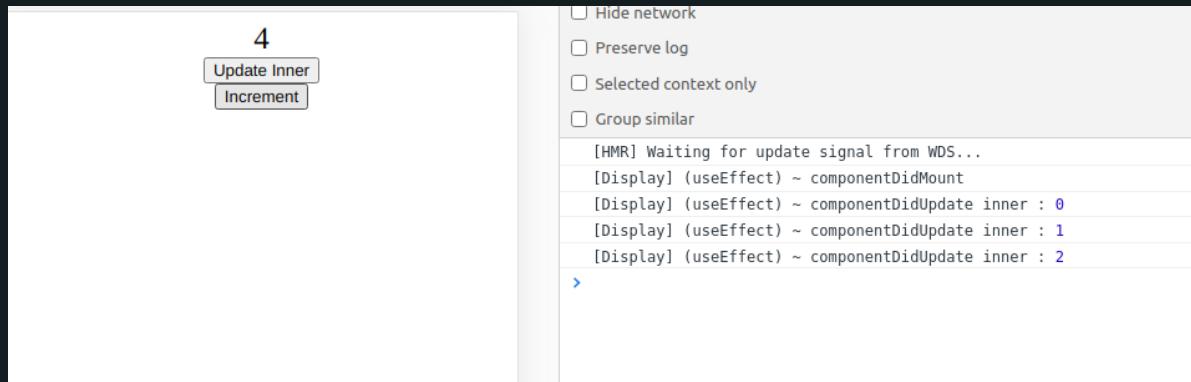
- if dependency list is passed as an empty array, call back function will execute only once when component is initially rendered. Thus it will be equal to `componentDidMount`.

`useEffect(() => {}, []) = componentDidMount`



```
src > components > Display.js > Display
1 import React, { useEffect } from "react";
2
3 import styles from "./Display.module.css";
4
5 export default function Display(props) {
6   useEffect(() => {
7     console.log("[Display] (useEffect)", "count:", props.count);
8   }, []);
9   return (
10     <div>
11       <div className={styles.display}> {props.count} </div>
12     </div>
13   );
14 }
15 }
```

- if dependency list contains some variables
useEffect will act as ComponentDidUpdate lifecycle method



```
src > components > Display.js > ...
1 import React, { useEffect, useState } from "react";
2
3 import styles from "./Display.module.css";
4
5 export default function Display(props) {
6   let [inner, setInner] = useState(0);
7
8   /* ComponentDidMount */
9
10  useEffect(() => {
11    console.log("[Display] (useEffect) ~ componentDidMount");
12  }, []);
13
14  useEffect(() => {
15    console.log("[Display] (useEffect) ~ componentDidUpdate", "inner :", inner);
16  }, [inner]);
17
18  return (
19    <div>
20      <div className={styles.display}> {props.count} </div>
21      <button
22        onClick={() => {
23          |   setInner(inner + 1);
24        }}>
25          Update Inner
26        </button>
27      </div>
28  );
29}
```

- Cleaning up with Lifecycle Hooks and useEffect :

Before a component is to be removed , Component class provides us with a lifecycle method known as `componentWillUnmount`.

This method is used for cleanup work such as dissociating any event listeners.

Incase of functional components this can be obtained by returning a function inside call back function of useEffect.

```

3
Update Inner
Remove Display Increment

 Hide network
 Preserve log
 Selected context only
 Group similar
[HMR] Waiting for update signal from WDS...
[Display] (useEffect) ~ componentDidMount
[Display] (useEffect) ~ componentDidUpdate inner : 0
[Display] (useEffect) ~ componentDidUpdate inner : 1
[Display] (useEffect) ~ componentDidUpdate inner : 2
[Display] (useEffect) ~ componentDidUpdate inner : 3

>
 Hide network
 Preserve log
 Selected context only
 Group similar
[HMR] Waiting for update signal from WDS...
[Display] (useEffect) ~ componentDidMount
[Display] (useEffect) ~ componentDidUpdate inner : 0
[Display] (useEffect) ~ componentDidUpdate inner : 1
[Display] (useEffect) ~ componentDidUpdate inner : 2
[Display] (useEffect) ~ componentDidUpdate inner : 3
[Display] (useEffect) ~ componentWillUnmount , Cleaning Up
>

```

```
src > components > Display.js > Display > useEffect() callback  
1 import React, { useEffect, useState } from "react";  
2  
3 import styles from "./Display.module.css";  
4  
5 export default function Display(props) {  
6   let [inner, setInner] = useState(0);  
7  
8   /* ComponentDidMount */  
9  
10  useEffect(() => {  
11    console.log("[Display] (useEffect) ~ componentDidMount");  
12    return () => {  
13      console.log("[Display] (useEffect) ~ componentWillUnmount , Cleaning Up");  
14    };  
15  }, []); → empty dependency list  
16  
17  useEffect(() => {  
18    console.log("[Display] (useEffect) ~ componentDidUpdate", "inner :", inner);  
19  }, [inner]);  
20  
21  return (  
22    <div>  
23      <div className={styles.display}> {props.count} </div>  
24      <button  
25        onClick={() => {  
26          setInner(inner + 1);  
27        }}>  
28        Update Inner  
29      </button>  
30    </div>  
31  );  
32}  
33
```

runs as cleanup
function ~
componentWill
Unmount

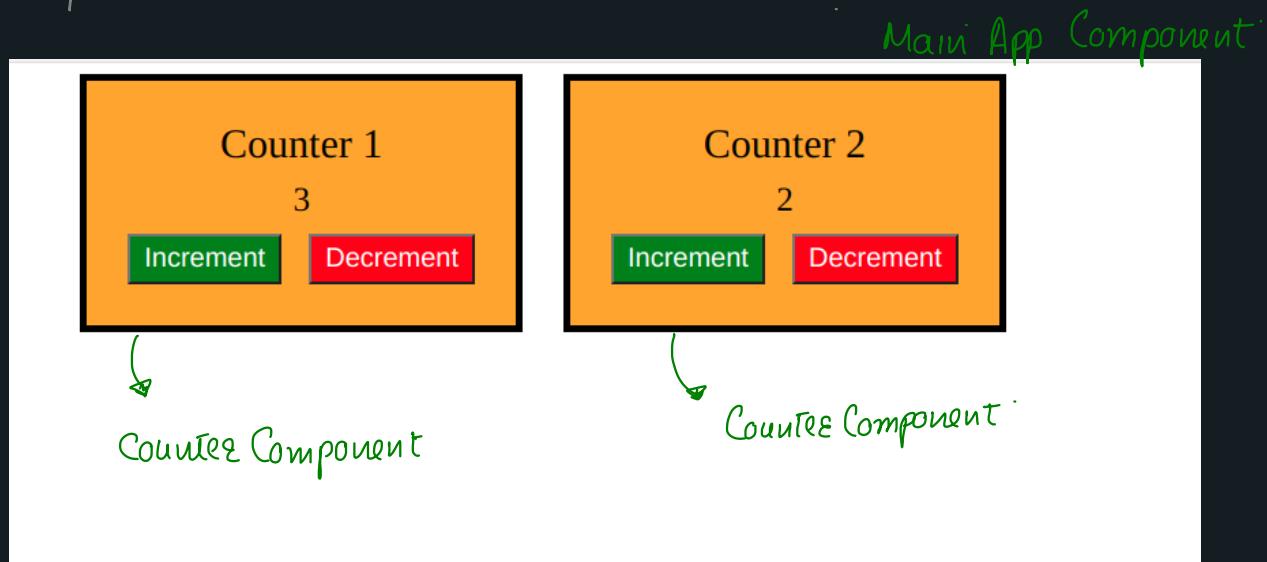
Note for this to act as component
will unmount . the dependency list
needs to be an empty array.

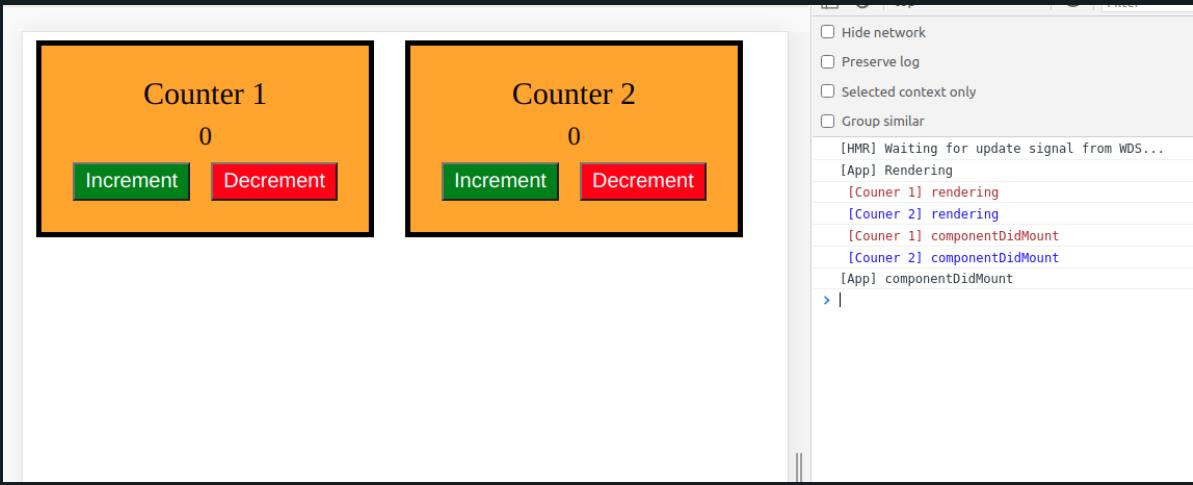
- `shouldComponentUpdate` for classbased component

OPTIMIZATION

in React whenever state of parent component changes, all the child components are re-rendered atleast in React Virtual DOM. This is how React works under the hood. However this behaviour can be optimized by using `shouldComponentUpdate` lifecycle hook.

Let us consider an example : Suppose we have an App component which has 2 child components (counters). The count value is stored by App component. Whenever state of component is updated both child components are re-rendered.





```

App.module.css   Counter.js   App.js   Counter.module.css
src > containers > App.js > App > componentDidUpdate
1  import React, { useState } from "react";
2
3 import Counter from "../components/Counter";
4
5 export default class App extends React.Component {
6   state = {
7     count1: 0,
8     count2: 0,
9   };
10
11 incrementHandler = (number) => {
12   switch (number) {
13     case 1:
14       this.setState({ count1: this.state.count1 + 1 });
15       break;
16
17     case 2:
18       this.setState({ count2: this.state.count2 + 1 });
19       break;
20
21     default:
22       return;
23   };
24 }
25
26 componentDidMount() {
27   console.log("[App] componentDidMount");
28 }
29
30 componentDidUpdate() {
31   console.log("[App] componentDidUpdate");
32 }
33
34 render() {
35   console.log("[App] Rendering");
36   return (
37     <div className='main_wrapper'>
38       <Counter
39         count={this.state.count1}
40         incHandler={() => {
41           this.incrementHandler(1);
42         }}
43         name={1}
44       />
45
46       <Counter
47         count={this.state.count2}
48         incHandler={() => {
49           this.incrementHandler(2);
50         }}
51         name={2}
52       />
53     </div>
54   );
55 }
56 }

```

```

src > components > Counter.js > Counter > constructor
1  import React, { Component } from "react";
2
3 import styles from "./Counter.module.css";
4
5 export default class Counter extends Component {
6   constructor(props) {
7     super(props);
8
9     this.state = {};
10
11    switch (this.props.name) {
12      case 1:
13        this.state.color = "color:brown";
14        break;
15
16      case 2:
17        this.state.color = "color:blue";
18        break;
19
20      default:
21        return;
22    }
23 }
24
25 componentDidMount() {
26   console.log(
27     `%c [Counter ${this.props.name}] componentDidMount`,
28     this.state.color
29   );
30 }
31
32 componentDidUpdate() {
33   console.log(
34     `%c [Counter ${this.props.name}] componentDidUpdate`,
35     this.state.color
36   );
37 }
38
39 shouldComponentUpdate() {
40   console.log(
41     `%c [Counter ${this.props.name}] shouldComponentUpdate`,
42     "color:red"
43   );
44
45   return true;
46 }
47
48 render() {
49   console.log(`%c [Counter ${this.props.name}] rendering`, this.state.color);
50   return (
51     <div className={styles.wrapper}>
52       <div className={styles.name}>Counter {this.props.name}</div>
53       <div className={styles.display}>{this.props.count}</div>
54       <div className={styles.controls}>
55         <button className={styles.green} onClick={this.props.incHandler}>
56           Increment
57         </button>
58         <button className={styles.red}>Decrement</button>
59       </div>
60     </div>
61   );
62 }
63 }

```

upon clicking this button, we get sideways console.log.

as it is clearly noticeable that by only incrementing count of a single counter both counters are re-rendered because App component is re-rendered. Clearly this is inefficient.

using `shouldComponentUpdate` for Optimization:

in this lifecycle method we can check if component needs to be rendered. In this particular case we can check that for a counter component prop value is different than existing prop value. If prop values are different, we will update component otherwise we will not update.

App.module.css Counter.js App.js Counter.module.css

src > components > Counter.js > Counter > shouldComponentUpdate

```

30 }
31
32 componentDidUpdate() {
33   console.log(
34     `%c [Counter ${this.props.name}] componentDidUpdate`,
35     this.state.color
36   );
37
38
39 shouldComponentUpdate(nextProps, nextState) {
40   console.log(
41     `%c [Counter ${this.props.name}] shouldComponentUpdate`,
42     "color:red"
43   );
44
45   if (nextProps.count !== this.props.count) return true;
46   else return false;
47 }
48
49 render() {
50   console.log(`%c [Counter ${this.props.name}] rendering`, this.state.color);
51   return (
52     <div className={styles.wrapper}>
53       <div className={styles.name}>Counter {this.props.name}</div>
54       <div className={styles.display}>{this.props.count}</div>
55       <div className={styles.controls}>
56         <button className={styles.green} onClick={this.props.incHandler}>
57           Increment
58         </button>
59         <button className={styles.red}>Decrement</button>
60       </div>
61     </div>
62   );
63 }
64
65

```

Here we check if component needs to be re-rendered

Counter 1

1

Increment Decrement

Counter 2

0

Increment Decrement

checkboxes:

- Hide network
- Preserve log
- Selected context only
- Group similar

[HMR] Waiting for update signal from WDS...

[App] Rendering

[Counter 1] rendering

[Counter 2] rendering

[Counter 1] componentDidMount

[Counter 2] componentDidMount

[App] componentDidMount

[App] Rendering

[Counter 1] shouldComponentUpdate

[Counter 1] rendering

[Counter 2] shouldComponentUpdate

[Counter 1] componentDidUpdate

[App] componentDidUpdate

Counter 1 is re-rendered.

Counter 2.

does not re-render because shouldComponentUpdate returns false.

lifecycle method returns false.

• Optimizing Functional Components using `React.memo()`:

For functional Components that do not need to change everytime parent component renders we can use `React.memo`.

`React.memo` → stands for memoization retains snapshot of component.

To use `React.memo` we need to use it as higher order component eg.

```
function XYZ () {  
    export default React.memo(XYZ)  
    return (  
        .....  
    )  
}
```

When should we Optimize:

Now when should we consider using these optimizations. We need to understand, that these also come at some cost. If we have parent component that directly holds data of child component.

then checking each time state changes will only add extra burden of execution.

- Pure Components instead of ShouldComponentUpdate

PureComponent is simply a normal component which comes with a preconfigured life cycle method that checks for equivalence of all the props.

Thus in situation where ShouldComponentUpdate needs to check for equivalence of all props it is better to use Pure Component

```
class X42 extends PureComponent { .....
```

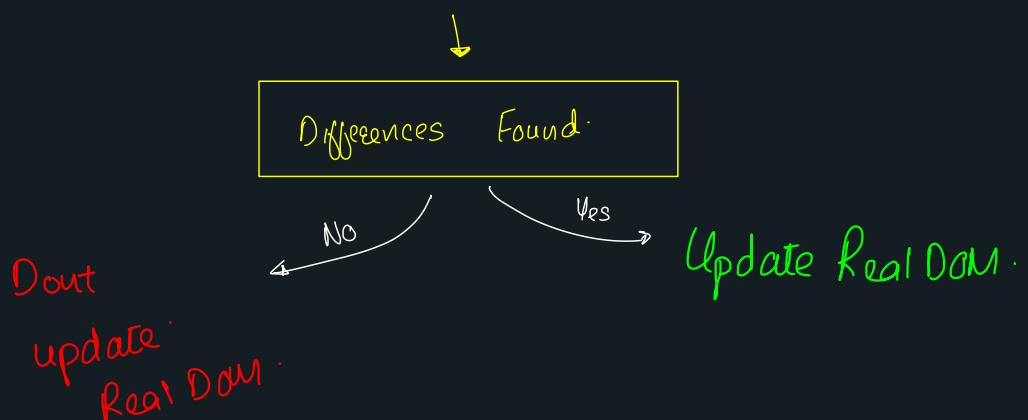
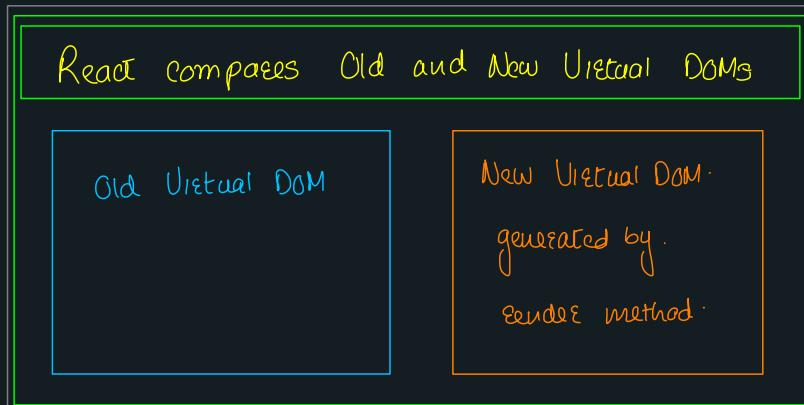
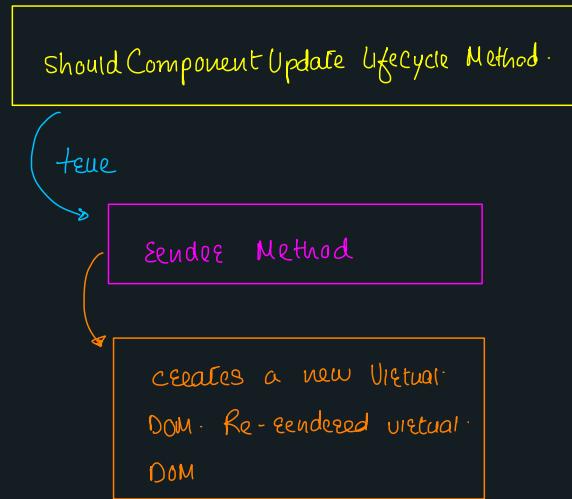
How REACT UPDATES THE DOM

Whenever shouldComponentUpdate lifecycle Method returns true .
render method is executed. Render method generates virtual
DOM . It must be noted that render method does not affect the
real DOM directly .

Once new virtual DOM is created , React compares it
with old virtual DOM to figure out any differences . If no
differences are found , real DOM is left untouched .
If differences are noticed , real DOM is updated
accordingly based on the differences and not entirely .

Virtual DOM is pretty fast as compared to real DOM
mutations . Thus React tries best to limit interactions with
React DOM .

React DOM Update Flow Chart :-



• Rendering JSX Elements :-

Now in React we can return JSX with a single root element.

This root element can contain as many as nested child elements.

However this is not entirely true because React allows us

to return an array of elements as long as each element has

a unique key.

How can we return adjacent elements?

```
class XYZ extends React.Component {  
  render() {  
    return ( [ <div key='a'> Hello </div> ,  
             <div key='b'> World </div> ,  
             <div key='c'> Hi </div>  
           ] )  
  }  
}
```

There is another feature build into React for this. That does not generate any HTML in DOM. It works as a higher order component.

④ Create a new folder in src called hoc

hoc stands for higher order components

• inside hoc folder create a file named Aux.js

Aux.js will contain following code.

```
const Aux = props => props.children
```

```
export default Aux
```

④ Usage : we use Aux component as a wrapper.

• using React.Fragment

React 16 comes with with Aux component build in called

React.Fragment. Fragment component has same use case.

as our self created Aux component.

• Higher Order Components (Introduction)

Components which wraps another components & may add some extra features to existing wrapped components.

features can be extra JSX

↳ styling

↳ extra logic

CONVENTION:

naming of HOC starts with With_name.....

- There are two forms of defining Higher Order Components

1) Normal React Components

2) Normal Javascript based functions

- Normal React Component based HOC: suppose we create a HOC that

adds styling (add more JSX)

HOC > WithClass.js

const WithClass = (props) =>

```
(  <div className={props.classes}>
    {props.children}
  </div>
)
```

export default WithClass

App.js

import WithClass from './hoc/...'

const App = () =>

return (

<WithClass classes={style}>

<div>

<h1> Hello World </h1>

</div>

</WithClass>

}

- Higher Order Components using Normal JS Functions: use when to add logic

hoc > withClass.js

```
const withClass = (WrappedComponent, classes) => {
  return (props) => {
    <div className={classes}>
      <WrappedComponent {...props} />
    </div>
  }
};

export default withClass;
```

App.js

```
import withClass from './hoc/..';

const App = () => {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  )
};

export default withClass(App, center);
```

- Setting State that depends upon previous state:

In many situations one new state depends upon previous state eg.

Let us consider counter example as follows

```
src > containers > App2.js > [e] default
1 import React from "react";
2
3 class App2 extends React.Component {
4   state = {
5     count: 0,
6   };
7
8   handleIncrement = () => {
9     this.setState({ count: this.state.count + 1 });
10
11
12   render() {
13     return (
14       <div>
15         <input value={this.state.count} />{" "}
16         <button onClick={this.handleIncrement}> Increment</button>
17       </div>
18     );
19   }
20 }
21
22 export default App2;
```

Problem with this Approach:

The problem with this approach is that although `useState` seems synchronous but actually it is not. React decides when to update state internally based on resource availability.

Thus even though `useState` might seem synchronous in reality it is not guaranteed that `this.state` — holds true consistent value.

Hence for situations where new state relies on previous state we use another approach. `useState` takes a function as follows

```
this.setState ((prevState, props) => {  
    return {  
        count: prevState.count + 1  
    }  
})
```

• PropTypes in React

Need for PropTypes:

Suppose we are working on a library or in a collaborative environment where other people can use our components incorrectly. Thus it would be beneficial if we can provide some sort of instructions about what props a component accepts and type of props.

This can be achieved using extra package called PropTypes.

① Installation : npm install --save prop-types

② Import : import PropTypes from 'prop-types'

③ Usage : after component definition

→ is a special property for any JS object
that each watches out for.

Counter.propTypes = {

Component
Name

handleIncrement: PropTypes.func,

count: PropTypes.number

- Accessing DOM via React : refs

We can select any element in DOM using refs or references

On any element we can add special property ref.

There are couple of ways of using refs

- Function Based : older approach.

```
class App extends Component {  
    render() {  
        return (  
            <div>  
                <input value="Hello" disabled={true} ref={inputEl => this.inputElement = inputEl}>  
            </div>  
        )  
    }  
}
```

ref takes in a function using which we set a global component level property inputElement

```
componentDidMount() {
```

```
    this.inputElement.focus()  
}
```

we can use this element inside component didMount lifecycle Method.

However this works only in class based components and not on functional components

- `React.createRef()` : Previous functional approach that we discussed was used in older versions of React. React provides us with `createRef()` which is used as follows. This is more Modern approach

```

class App extends React.Component {
  ▪ inputElementRef = React.createRef();
  render() {
    return (
      <div>
        <input ref={this.inputElementRef} />
      </div>
    )
  }
  componentDidMount() {
    console.log(this.inputElementRef.current)
  }
}

```

- Refs in functional Components with React Hooks:

Incase of functional components we cannot use `React.createRef()` either we have to make use of a React Hook Known as `useRef()`

Following example will help us to understand

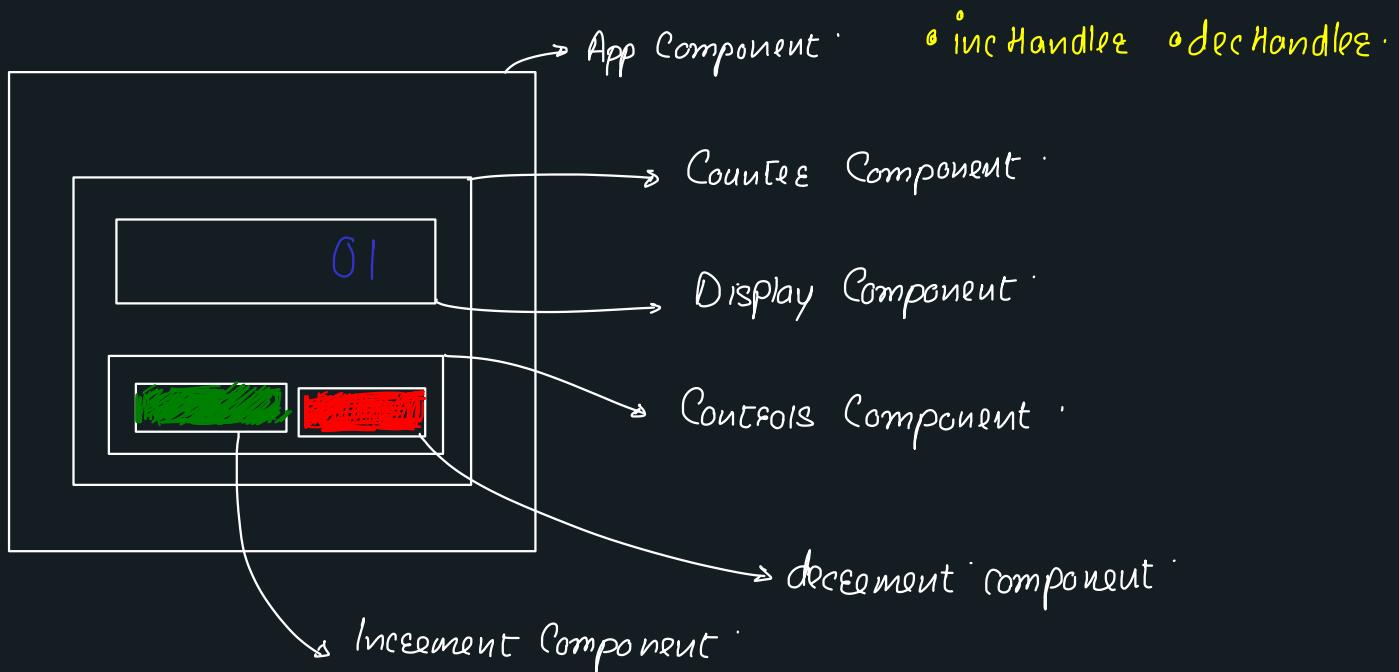
```
import React, { useRef } from 'react';
```

```
const App = (props) => {
  const mainRef = useRef(null);
  return (
    <div>
      <div ref={mainRef}> HelloWorld </div>
    </div>
  )
}
```

Understanding Prop Chain Problem:

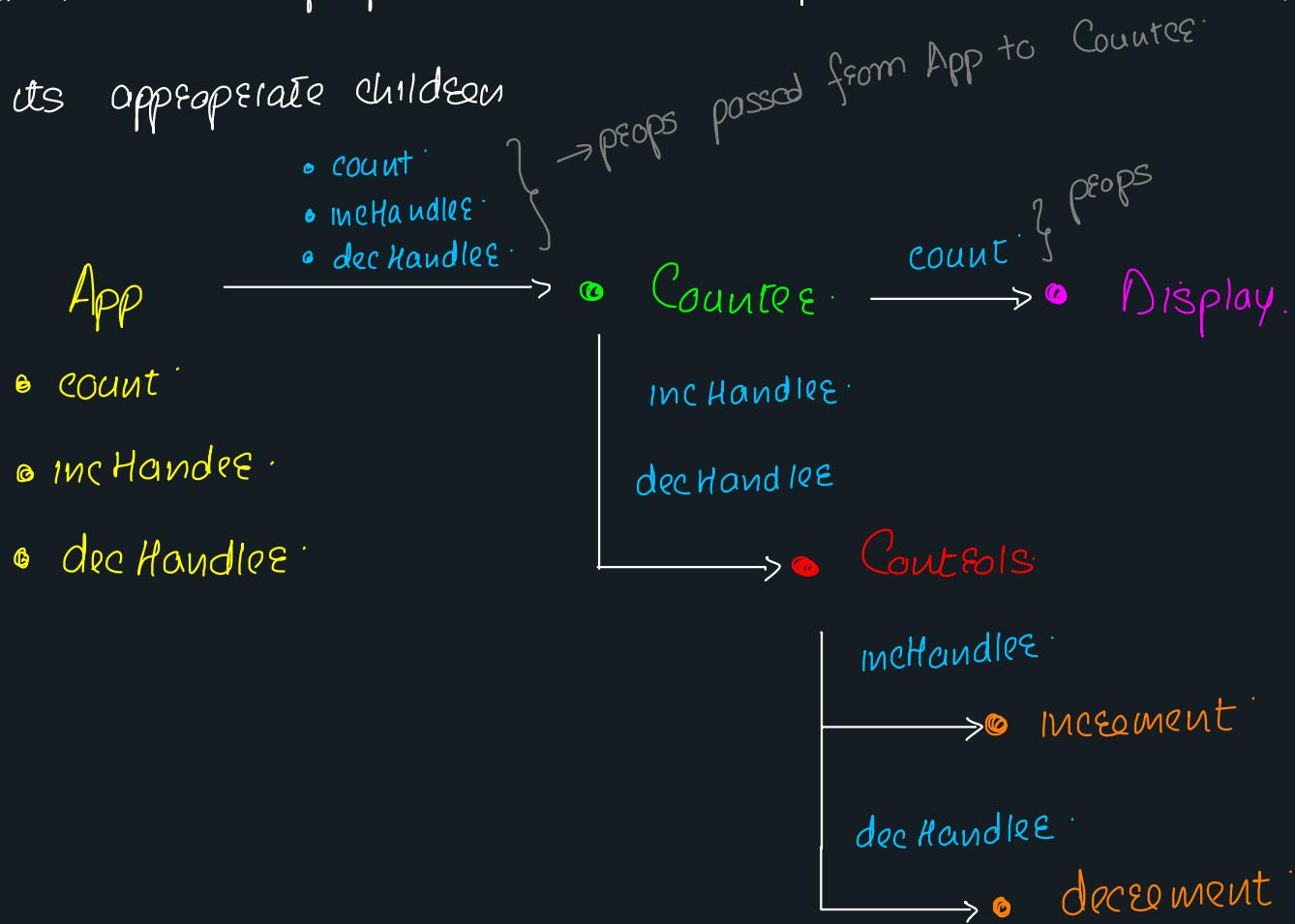
Let us try to consider prop chain problem using a simple example. Suppose we have a counter App which can be dissected as follows:

state = { count: 0 }



App component maintains state & contains handles

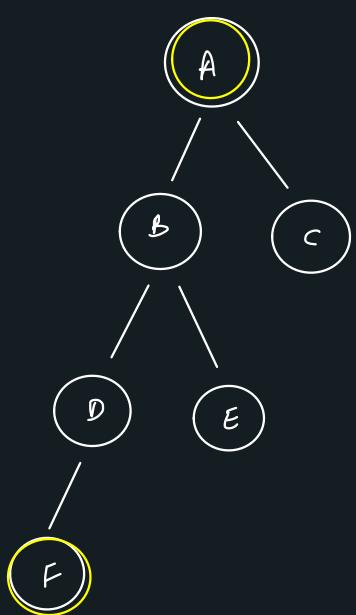
For this Application to work properly App needs to pass count and Handles as props to its child component and child component to its appropriate children



Thus for increment component to use incHandlee it needs to trace back a chain of props from Controls → Counter → App

Although for shallow components this might not be an issue but for deeply nested components it can be tedious.

To help with problem chaining React provides us with CONTEXT API. Context API provides us to pass props from a root component to deeply nested child component without need to pass props through the entire hierarchy eg.



context API allows to pass props directly from A to F without need of passing props through B and D in this case.

Context (Documentation from Reactjs.org)

Context provides a way to pass data through component tree without having to pass props manually at every level.

API for Context - `React.createContext`

Context is based on a Provider Consumer Model. Thus it has:

2 parts • Provider • Consumer



allows subscribers to subscribe or

listen to context changes

- accepts a value prop to be passed to consuming components
- all consumers will re-render whenever value prop changes

- a react component that subscribes to context changes lets subscribe to a context

- requires function as a child function receives context value and returns a react node

1. Creating context in `src > context`

```
src > context > CounterContext.js > ...
1 import React from "react";
2
3 const CounterContext = React.createContext({
4   count: 0,
5   incHandler: () => {},
6   decHandler: () => {},
7   clearHandler: () => {},
8 });
9
10 export default CounterContext;
11
```

2. Establishing Context Provider

```
src > components > Counter > Counter.js > Counter > render > clearHandler
1 import React, { Component } from "react";
2
3 import Display from "./Display/Display";
4 import Controls from "./Controls/Controls";
5
6 import CounterContext from "../../context/CounterContext";
7
8 export default class Counter extends Component {
9   state = {
10     count: 0,
11   };
12
13   incHandler = () => {
14     this.setState((prevState, props) => {
15       return {
16         count: prevState.count + 1,
17       };
18     });
19   };
20
21   decHandler = () => {
22     this.setState((prevState, props) => {
23       return {
24         count: prevState.count - 1,
25       };
26     });
27   };
28
29   clearHandler = () => {
30     this.setState((prevState, props) => {
31       return {
32         count: 0,
33       };
34     });
35   };
36
37   render() {
38     return (
39       <div style={styles}>
40         <CounterContext.Provider →
41           value={[
42             count: this.state.count,
43             incHandler: this.incHandler,
44             decHandler: this.decHandler,
45             clearHandler: this.clearHandler,
46           ]}>
47           <Display />
48           <Controls />
49         </CounterContext.Provider> →
50       </div>
51     );
52   }
53 }
```

Provide Value

only these components can access context

```
src > components > Counter > Display > Display.js > Display > render
1 import React, { Component } from "react";
2
3 import CounterContext from "../../../../context/CounterContext";
4
5 export default class Display extends Component {
6   render() {
7     return (
8       <div style={styles}>
9         <CounterContext.Consumer → CONSUMER>
10        {(context) => [
11          return (
12            <input
13              style={styles.input}
14              value={context.count}
15              onChange={() => {}}
16            />
17          );
18        ]};
19      </CounterContext.Consumer>
20    </div>
21  );
22}
```

function

```
src > containers > App.js > App > render
1 import React from "react";
2
3 import Counter from "../components/Counter/Counter";
4
5 class App extends React.Component {
6   render() {
7     return (
8       <div
9         style={{
10           height: "100vh",
11           width: "100vw",
12           display: "flex",
13           justifyContent: "center",
14           alignItems: "center",
15         }}>
16         <Counter />
17       </div>
18     );
19   }
20 }
21
22 export default App;
```

```
src > components > Counter > Controls > Controls.js > Controls > render
1 import React, { Component } from "react";
2
3 import CounterContext from "../../../../context/CounterContext";
4
5 export default class Controls extends Component {
6   render() {
7     return (
8       <div style={styles.wrapper}>
9         <CounterContext.Consumer>
10        {(context) => [
11          return (
12            <div>
13              <button onClick={context.incHandler} style={styles.btnGreen}>
14                INC
15              </button>
16              <button onClick={context.decHandler} style={styles.btnOrange}>
17                DEC
18              </button>
19              <button onClick={context.clearHandler} style={styles.btnRed}>
20                CLEAR
21              </button>
22            </div>
23          );
24        ]};
25      </CounterContext.Consumer>
26    </div>
27  );
28 }
29
30 const styles = {
31   wrapper: {
32     display: "flex",
33     marginTop: "30px",
34     justifyContent: "center",
35   },
36
37   btnRed: {
38     padding: "15px 20px",
39     backgroundColor: "red",
40     margin: "auto 10px",
41   },
42   btnGreen: {
43     padding: "15px 20px",
44     backgroundColor: "green",
45     margin: "auto 10px",
46   },
47
48   btnOrange: {
49     padding: "15px 20px",
50     backgroundColor: "orange",
51     margin: "auto 10px",
52   },
53 }
```

• ContextType :

- works with class based components only
- provides more elegant way of writing consumer.

Previously

```
<Context.Consumer>
```

```
{(context) => {  
    ...  
}}
```

} context is only accessible inside this block

```
</Context.Consumer>
```

- Hence we cannot use context in lifecycle methods or outside this block.

with context Type :

```
src > components > Counter > Display > Display.js > render
```

```
1 import React, { Component } from "react";  
2  
3 import CounterContext from "../../../../context/CounterContext";  
4  
5 export default class Display extends Component {  
6     static contextType = CounterContext; ① assign , here static is used so React can directly use contextType properly without creating an object of class  
7     componentDidMount() {  
8         console.log(this.context);  
9     }  
10  
11     render() {  
12         return (  
13             <div style={styles}>  
14                 <input  
15                     style={styles.input}  
16                     value={this.context.count} ② usage all value of provider can be  
17                     onChange={() => {}}  
18                 />  
19                 </div>  
20             );  
21         };  
22     }  
23 }
```

① assign , here static is used so React can directly use contextType properly without creating an object of class

② usage all value of provider can be used as this.context. —

Thus context type increases scope of context inside consumer component.

• Context for Functional Components :

In case of functional components, context is provided by useContext Hook

```
import React from 'react';
import CounterContext from '.../.../context/CounterContext'; ① import Context
import {useContext} from 'React'; ② import Hook
```

```
const Display = (props) => {
  const counterContext = useContext(CounterContext); ③ bind context
  return (
    <div>
      <input value={counterContext.count} ④ use context
        disabled={true}>
      </div>
    )
}
```

