**React JS Notes**

1. **Class base Component: -**

* React started with Class based component. There was nothing like functional component, hooks, useState, useEffect.
* Writing code in class base component was very messy, no clean code. Class base component code was very big as comparedto functional component.
* Render() 🡪 We cannot make a class based component without render() method.
* API call like useEffect 🡪 **componentDidMount** is used in class based for API call.
* Component life cycle sequence –

1. Constructor
2. Render
3. componentDidMount

* ..
* ..

1. **Route: -**

* Nested Route 🡪 We can create nested route by Children of Children
* For children of children route we don’t give “/”. Example below:

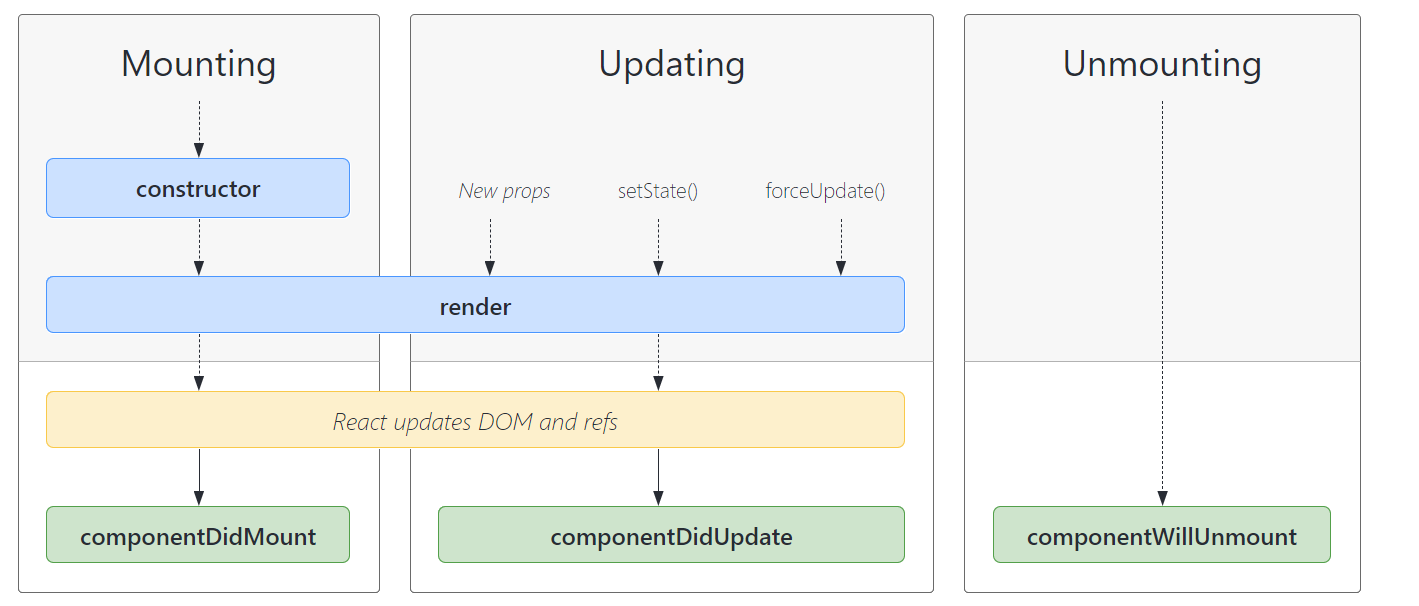


* ..

1. **Why do we do super(props) ?**
2. In a React class-based component, the **super(props)** method is used to call the constructor of the parent class (i.e., the **React.Component** class) and pass the props that were passed to the current component as an argument. This is necessary because the constructor of the parent class is responsible for initializing the component's state and props, and by calling **super(props)**, we ensure that this initialization is done before we start using the props in the child component's constructor.
3. It's also important to note that **super(props)** should be called before any other statement in the constructor because the **this** keyword can only be used after the parent class's constructor has been called.
4. Additionally, when the parent component pass the props and state to the child component, it's important to make sure that the child component can access those props and state. Calling **super(props)** makes it possible for the child component to access the props and state passed down from the parent component.
5. In summary, **super(props)** is used to call the constructor of the parent class, pass the props to the parent class, and ensure that the parent class has initialized the component's state and props before we start using them in the child component's constructor.
6. **2 Phases are there when reconcilliation happes**
7. Render Phase 🡪 Here 1st constructore is executed and then render executes
8. Commit Phase 🡪 Here 1st DOM updates and then componentDidMount executes
9. **We can make componentDidMount async but we cannot make useEffect() async ? Reason ?**

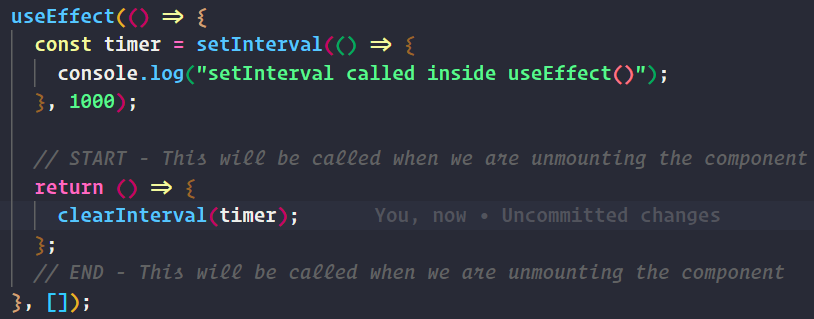
* **useEffect** is a hook that allows you to synchronously synchronize a component with an external system. It is not designed to handle asynchronous operations, such as fetching data from a remote server. On the other hand, **componentDidMount** is a lifecycle method in a class component that is called after the component has been rendered. Because it is called after the component has been rendered, it is the appropriate lifecycle method to use when you need to perform an asynchronous operation, such as fetching data. It is possible to make useEffect async by wrapping the effect with an async function but it's not recommended because it would cause an unnecessary re-rendering and it is not guaranteed to work as expected.

1. **componentDidMount()** 🡪 Will be called after 1st render().
2. **componentDidUpdate() 🡪** Will be called after every 1st render(). That is from 2nd render time this will be called
3. **React life cycle method diagram below** – https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/



1. Note 🡪 Never compare class-based life cycle with functional based life cycle.
2. useEffect 🡪 When we don’t put any dependency array in it then useEffect will be called after every render()
3. useEffect 🡪 When we put empty dependency array [], then it will be called only once in the initial render()
4. UnMount in Functional Based Component:

* We need to use return() method inside useEffect()]to unmount a component in class based component. Example below



* useEffect return will execute once we leave that component.

1. **Custome Hooks:**
2. Why do we build Hooks?

* Reusability, Readability, Testable, Modularity, Seperation of Concern, Maintainability.
* Wrap a small logic in function and can reuse it anywhere we want to.
* Modularity 🡪 Means code broken down into smaller meaning full pieces.

1. Lazy Loading :
2. We use lazy() from “react” to load component on demand



1. We use <Suspense> from “react” to bind the component. <Suspense> has fallback to load Shimmer till component gets loaded.



1. Is lazy loading / chunking necessassry for smaller APP
2. Never ever load dynamic component / lazy loading inside another component 🡪 Reason this dymanic component will load on every render() method.
3. Tailwind CSS
4. Tailwind CSS features

* CSS on the go (In the same file)
* Reusability
* Less bundle sizes
* Flexible UI (Customizable)

1. Sharing Data through PROPS – Props Drilling

* When we need to pass data to some component which are far down as child component then, that data needs to be paas from all its parent down to the child component where the data is actually needed to render.
* This process of passing the data from parent to child then to its child and so on is called PROPS Drilling.

1. Lifting the state up

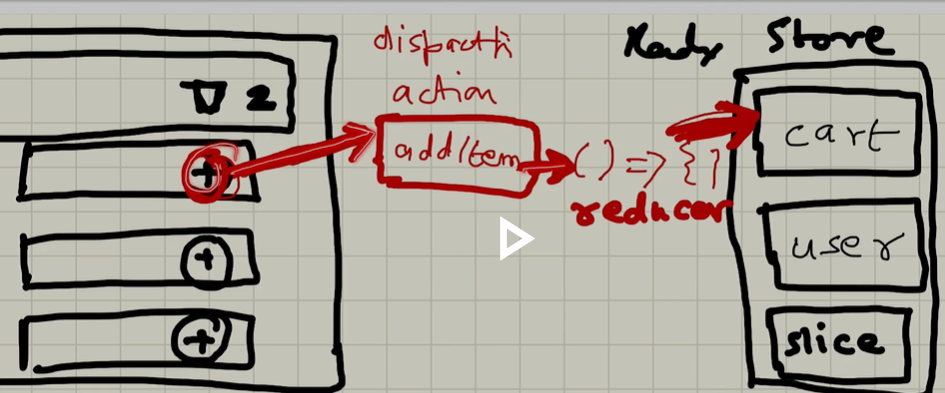
* When we want to have controls of all the sibling components state then we follow lifting the state up process by giving the access of all the sibling components state to its parent.

1. React Context 🡪 It is a shared state for whole react app.

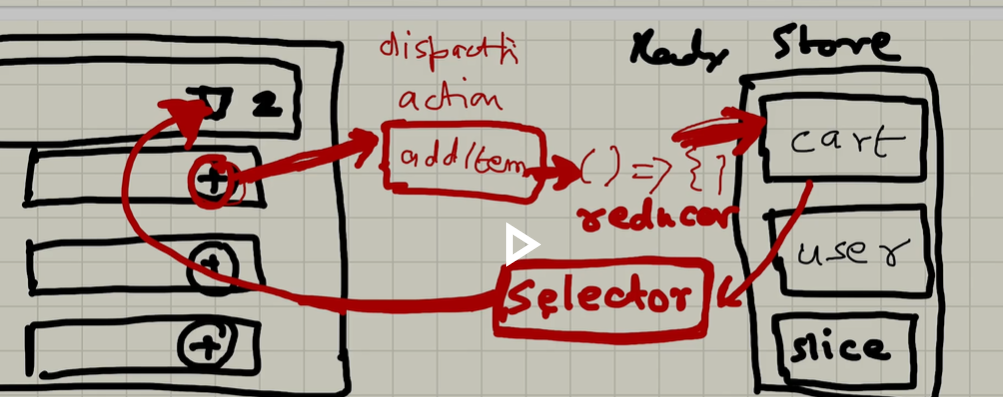
* Any component can use this data.

1. Redux 🡪 We need reduc to manage Data layer of an application

* Redux is used to handle huge amount of data
* Why do we use CONTEXT 🡪 To avoid PROPS drilling
* Context is a central place, where we keep data that can be shared amongs all other component withour PROP drilling. Any component can access / modify the context
* Redux is used for Data Management.
* Major Problem with Redux 🡪 It is complex to set up. It has a lots of learning curves.
* Redux STORE is a big object, which has different sections.
* Which component can access redux store 🡪 All the component can access
* Store has all the application data in one store
* Each specific data is seperated by logical seperation.
* These logical seperation is known as “SLICE”
* For example, userSlice, authenticationSlice, themeSlice etc….
* Our component **cannot directly modify the Store**.
* If a component needs to modyfy store then it has to “dispatch” and “action”
* Action will call a function and that function will modify the Store slice.
* The function which modifies Store is called Reducer



* Write Store 🡪 click on + 🡪 dispatch an Action 🡪 which calls a function called reducer 🡪 then reducer updates slice of store.
* Read Store 🡪 Selector is used to read from the store



* When we use Selector that means we are subscribing to the store.
* Redux Flow: -

1. Click on ADD Button
2. It dispatches an Action (dispatch comes from useDispatch from react-redux)
3. On click of button we called a function “handler” which dispatches and an action
4. Action is coming from Slice Action that we created

* ..
* ..

1. ..
2. ..
3. ..
4. ..
5. ..
6. ..
7. ..
8. ..
9. ..
10. ..
11. ..

**Begining**

1) npm init

2) npm install -D parcel

3) npm i react

4) npm i react-dom

5) Code setup

index.html

----------

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>React App</title>

</head>

<body>

<div id="root">No Data Rendered</div>

<script type="module" src="App.js"></script>

</body>

</html>

App.js

------

import React from "react";

import ReactDOM from "react-dom/client";

const root = ReactDOM.createRoot(document.getElementById("root"));

const element = <h1>Hello, world</h1>;

root.render(element);

-------

6) npx parcel index.html

7) HMR - Hot Module Reload --> Due to this parcel refreshes the browser if any changes done in code

8) Script type should be module as we are exporting the library files from node module. Example below :-

<script type="module" src="App.js"></script>

9) File watcher algorithm --> It keeps track of any changes done in any file and refreshes the page.

10) Parcel jobs behind the scene are given below :-

--> Creates a Server (http://localhost:1234/)

--> HMR - Hot Module Replacement --> Due to this parcel refreshes the browser if any changes done in code

--> File watcher algorithm keeps track of any changes done in any file and refreshes the page.

--> Bundling the code for production ready code

--> Minify (On Production Environment)

--> Cleaning all unwanted code like console.log.

To remove console.log install(npm install babel-plugin-transform-remove-console --save-dev)

// with options --> create .babelrc file and add below code in that file

{

"plugins": [["transform-remove-console", { "exclude": ["error", "warn"] }]]

}

--> Manages Dev and Production Build & it is super fast build algorithm

--> It optimises images and videos

--> Caching while Development

--> It compresses files.

--> Compatible with older versions of Browsers (It adds polyfill)

--> It help enabling HHTPS on DEV environment (npx parcel index.html --https)

--> If we run/build 2 project simultanisously then parcel auto manages ports for 2 projects

--> Parcel-Cache should be put in git ignore

--> Consistent Hashing Alorithm

--> Parcel is Zero Config bundler

--> Transitive Dependencies (One package is dependent on another and another is dependent on other)

--> Tree Shaking --> Removes all unwanted code

11) https://browserslist.dev/?q=bGFzdCAyIHZlcnNpb25z --> Contains all supported browsers and versions

12) React Reconciliation --> If any new html elements is introduces and if it does not have key,

react takes lots of time to re-render DOM.

But if we have specific and unique key attached to html element

then it just re-render it withput updating whole DOM

(Documentation - https://reactjs.org/docs/reconciliation.html)

<ul>

<li key="2014">Connecticut</li>

<li key="2015">Duke</li>

<li key="2016">Villanova</li>

</ul>

13) React.createElement gives us an Object and this object is converted into HTML and it renders in DOM

14) JSX --> It is an HTML like syntax, but it is not HTML inside JavaScript.

15) JSX code is not understood by Browser, BABEL understand and create browser readable code and then

it is executed in Browser

16) Babel --> Is a compiler for next generation JavaScript.

17) JSX --> Uses React.createElement --> Then it is converted to Object --> Then HTML --> Render in DOM

18) node\_modules --> Also has package-lock.json file.

this package-lock.json contails all Transitive Dependencies

(One package is dependent on another and another is dependent on other)

19) React Component :-->

a) React Component is of two type (Functional Component & Class Based Component)

b) Functional Component --> Name starts with the Capital Letter. Example below :-

const HeaderComponent = () => {

return <h1>Functional Header Component</h1>;

};

c) Functional Component is nothing but it it just a Functional

d) To render functional component we use --> root.render(<HeaderComponent />)

e) To render react element we use --> root.render(header). Here const header = <h1>Header</h1>

f) If we want to use react element inside function we can do with {} brace as below :-

const root = ReactDOM.createRoot(document.getElementById("root"));

const name = <h1>Bikash Shaw</h1>;

const HeaderComponent = () => {

return (

<div>

{name} <-- Like this

<h1>Functional Header Component</h1>;

</div>

);

};

root.render(<HeaderComponent />);

g) Component Composition --> If we want to use functional component into functional component

(Nested Component)

const root = ReactDOM.createRoot(document.getElementById("root"));

const NameComponent = () => {

return <h1>Bikash Shaw</h1>;

};

const HeaderComponent = () => {

return (

<div>

{NameComponent()} <-- Call it as a normal function call.

<NameComponent /> <-- Call it as JSX tag.

<h1>Functional Header Component</h1>;

</div>

);

};

root.render(<HeaderComponent />);

20) {} ==> We can write any piece of JavaScript code inside curly braces

21) Is JSX mandatory --> JSX is not mandatory.

22) Is TypeScript mandatory --> TypeScript is not mandatory

23) Is ES6 mandatory --> ES6 is not mandatory

24) React is a library and it does not restricts on any language. We can use any language.

25) JSX expression should have only one parent element.

a) React.Fragment --> React.Fragment is a component which is exported by React library.

b) React.Fragment --> It is like an empty tag.

c) React.Fragment --> shorthand for React.Fragment is <> </>

d) Can we use nested React.Fragment --> ? (Need to google it)

26) Config Driven UI -->

a)

27) Optional Chaining --> {Google it}

28) Props -->

a) Props is nothing but a properties

b) Props is passing some data into our component

c) Send data from Parent to Child we pass it as a PROPS

d)

29) Virtual DOM -->

a) Virtual DOM is not just a concept for react only. It is used in may places, react also has virtual DOM

b) Virtual DOM is a software engineering concept.

c) Virtual DOM is a representation of actual DOM in our Code.

d) Why do we need Virtual DOM ?

Ans) We need it for re-concilliation. Reconciliation is an algorithm that react uses to {diff}

One tree from other, to identify what needs to change and what shoul not change in DOM.

{Key} which is unique are used for reconciliation to happen properly.

React Fiber is a new Reconciliation Engine.

--> Why do we not use {index} as a key ?(https://robinpokorny.com/blog/index-as-a-key-is-an-anti-pattern/)

30) Export Module -->

a) Export Default --> We can only export default only one component.

b) Export default can we imported like --> import Header from "./components/Header"; (default import)

c) Export (also known as named import) --> import { Header } from "./components/Header"; (Named import)

d) If we have multiple function in one component and we want to import all the function. Below is code :-

import \* as Obj from "./components/Header";

e)

31) Data Binding -->

a) React uses One Way Data Binding

b) State --> Every component in React maintains a STATE and we can put variable into that STATE

c) Hook --> Is nothing but a normal function

d) useState() --> It creates a state variables

e) useState() --> this function returns an array. First element of an array is "variableName", second is "setVariable"

f) const [searchTxt] = useState(); --> This is how we create local state variables in React

g) useState comes as (named import) in react

h)

32) API Calls -->

a) fetch() --> browser API is used to call an API

b) We will not call fetch() directly in our component, because every time if any state variable changes,

Component get redenderd and API will be called again and again. This is bad implementation

c) Component gets renders 2 times, on PROPS change and on STATE change

d) If we dont provide "dependency Array" in side useEffect then it will get call on every re render.

e) If we dont want to call it on every re render then we pass "dependency array" in useEffect.

f) useEffect(() => {

}, []); ==> here [] is a dependency array

g) If we dont give any dependency in "dependency array" then it will get called only once after "render"

h)

33) Shimmer UI --> Load a blank page design before APIL call and fill data filling happens.

34) Never --> Never create a component in side one component.

35) Never --> Never write useState inside if condition or inside for loop, never write useState outside

functional component.

36) We can create as many as useEffect we want.

37) Forms npm package --> formik & yup--> npm i formik

38) Link Component --> At the end of the day Link Component uses <a href > only.

39)