Polyfill js makes older browsers to support the new code.

May be like the ES6 functions which are not supported in Internet Explorer so polyfill comes into rescue to make the IE to support the ES6 code.

polyfill converts our code into older code.

jsx expressions.

const h1 = <h1>Hello</h1>

the above is a jsx expression.

//components

Functional components

class components

functional components

In React, a functional component is a type of component that is primarily defined as a JavaScript function. Functional components were introduced in React as a simpler way to define and manage the behavior and rendering of UI elements. They are also sometimes referred to as "stateless components" because they do not have their own internal state.

A component name start with a capital letter.it is a good practice.

A jsx is just a react element.

a react element is a object.

a react functional component is a function.

const Title = () =>

{

return <h1>Hello world</h1>;

}

we can use functional component inside jsx in 2 ways

<Title /> and {Title()} since a Fucntional component is just a function we can call it inside our jsx.

we can write console.log() also inside jsx.

javascript jsx is very safee.

jsx will sanitize the js code first.

component composition -> using components inside a component.

react reconcillation

if we create a react app then a build tool webpack will also comes which includes babel a js code transpiler which conversts our jsx code into browser understandable language.

React.Fragment will directly adds your code into the DOM without the need of any parent div tag.

<div>

<p>Element 1</p>

<p>Element 2</p>

</div>

From the above to the below

<React.Fragment>

<p>Element 1</p>

<p>Element 2</p>

</React.Fragment>

<>

</>

these empty tags equals to the React.Fragment

A fragement will directly adds your tags to the root tag in DOM, it avoids creating unneccesary DOM tree.

when passing props

if you pass like <Body restaurent={somedata} />

and while receiving you can do two wayss

like

const Body = (props) => {

}

in this way you can access data like props.restaurent.name

where as another way is

const Body = ({restaurent}) => {

}

now you can directly access restuarent.name instead of props.restaurent.name

this is called destructuring.

PROPS DESTRUCTURING.

PROPS WITH SPREAD OPERATOR.

VIRTUAL DOM

keeping a copy of original DOM

we need virtual dom for reconcillation in react.

Reconcilliation is an algorithm that react uses to diff one dom tree from another.

the diff algorithm finds out what needs to be modified/updated and updated that small portion instead of re-rendering entire DOM.

whenever a change happened to a react application a new VirtualDom will be created with the changes happened, now the new VDOM will be compared with the old VDOM, the comparision can be done by the diffing algorithm, the diffing algorithm will detect the changes between new VDOM and old VDOM and it render the detected changes in the original DOM.

this entire process is called react Reconcillation.

Initial Render: When a component is initially rendered, a VDOM tree is created to represent the initial UI. React keeps a reference to this VDOM.

Subsequent Renders: When a change occurs, a new VDOM tree is created to represent the updated UI. This new VDOM is compared to the most recent VDOM (not the initial one) using the diffing algorithm.

Reconciliation: The differences between the new VDOM and the most recent VDOM are identified, and the actual DOM is updated accordingly.

Retaining the Most Recent VDOM: Importantly, after reconciliation, React retains a reference to the newly created VDOM. This VDOM becomes the basis for comparison in the next render cycle.

Next Render: When another change occurs, the newly created VDOM is compared to the most recent VDOM from the previous render. This process repeats in subsequent render cycles, with each new VDOM being compared to the most recent VDOM reference.

In summary, React keeps a reference to the most recent VDOM, allowing it to perform comparisons with the new VDOMs generated in subsequent renders. The old VDOM is not entirely erased after each reconciliation; instead, it's replaced by the new most recent VDOM. This approach ensures that React can efficiently track and update the UI in response to changes in application state.

Yes, in a typical React application, except for the most recent Virtual DOM (VDOM), all previously created VDOMs are eventually discarded. React retains a reference only to the latest VDOM, which is used as the basis for comparison during the next render cycle.

The reason for discarding older VDOMs is to manage memory efficiently. As the application continues to render and update, it would be impractical to keep references to all previous VDOMs in memory. Instead, React relies on the most recent VDOM to perform the reconciliation process and update the actual Document Object Model (DOM) as needed.

Here's a simplified overview of the process:

During the initial render, a VDOM is created and used to generate the initial DOM.

When a change occurs, a new VDOM is created to represent the updated UI.

The new VDOM is compared to the most recent VDOM, and differences are identified.

The actual DOM is updated based on the differences.

The most recent VDOM reference is updated to point to the new VDOM.

The previous VDOM is effectively discarded, and it can be cleaned up by JavaScript's garbage collector because there are no references to it.

Subsequent render cycles use the updated most recent VDOM as the reference point for comparisons.

By retaining only the most recent VDOM, React strikes a balance between efficient memory management and the ability to update the UI in response to changing application state. This approach allows React to perform efficiently even in applications with many render cycles and updates.

React fiber.

React Fiber is a new reconciliation algorithm for React that was introduced in React 16. It is a complete rewrite of the previous reconciliation algorithm, but it is backwards compatible, so you can start using it without changing your existing code.

React Fiber has a number of advantages over the previous reconciliation algorithm, including:

\* \*\*Incremental rendering:\*\* React Fiber can split rendering work into chunks and spread it out over multiple frames. This makes it possible to render complex UIs without blocking the main thread.

\* \*\*Better performance for animations and gestures:\*\* React Fiber can prioritize rendering of animations and gestures, so that they are smooth and responsive.

\* \*\*Concurrency support:\*\* React Fiber makes it possible to write concurrent React applications. This means that your application can continue to respond to user input and other events even while it is rendering a complex UI.

React Fiber is still under development, but it has already been used to improve the performance of many popular React applications.

Here are some examples of how React Fiber can improve the performance of your application:

\* \*\*Rendering a large list of items:\*\* With the previous reconciliation algorithm, rendering a large list of items could block the main thread and cause your application to become unresponsive. With React Fiber, rendering a large list of items can be spread out over multiple frames, so your application remains responsive.

\* \*\*Animating a complex UI:\*\* With the previous reconciliation algorithm, animating a complex UI could cause the animation to stutter or lag. With React Fiber, React can prioritize rendering of animations, so that they are smooth and responsive.

\* \*\*Handling user input while rendering a complex UI:\*\* With the previous reconciliation algorithm, handling user input while rendering a complex UI could cause your application to become unresponsive. With React Fiber, React can continue to respond to user input even while it is rendering a complex UI.

If you are building a React application, I encourage you to start using React Fiber. It can make a significant difference in the performance and responsiveness of your application.

A key should not be an index in react.

There are two main reasons why we don't use index as a key in React:

Performance: When you use index as a key, React has to re-render the entire list if any item is reordered or removed. This is because React cannot tell which items have changed based on their index alone.

Correctness: If you use index as a key, and you reorder or remove an item, React may end up re-rendering the wrong item. This is because the index of an item can change when the list is reordered.

To avoid these problems, it is best to use a unique identifier for each item in the list as the key. This will allow React to efficiently update the list when items are reordered or removed.

Here are some examples of good keys to use in React lists:

The ID of the item, if it has one.

A unique string generated for each item.

A combination of the item's type and some other unique identifier, such as the item's timestamp.

config driven UI is nothing but the UI has to be modified according to the data without any code changes.

there are two types of import/exports in react/javascript.

1.default import

export default Hello;

then you have to import like

import Hello from './Hello';

2.Named import

export Hello;

then you have to import like

import {Hello} from './Hello';

if you have a file say Details.js in which you are having two components

export const PersonalDetails = () => {}

export const ProfessionalDetails = () => {}

you have exported both the file.

now you can import like

import {PersonalDetails,ProfessionalDetails} from './Details';

and in default import exports there is no need to import with the same component name

const Hello = () => {};

export default Hello;

you can import it with any name like

import Hello from './Hello';

or

import Ravi from './Hello';

you can either say './Hello'; or './Hello.js';

named imports can also be imported like

import \* as Something from './Details';

then use it like <Something.PersonalDetails />

a react variable is a state variable,every component in react maintains a state;

hooks are special js functions.

React will update the dom only when the state of variables changes, that is why react needs state variables rather than the local variables.

when a state changes in react a component re-render will happens but that does not re-contruct or replace the entire dom tree, it will then starts the reconcillation process and through diffing algorithm it will modify the dom tree as per the results from the diffing algorithm.

useEffect

when you provide some dependencies to the useEffect dependency array then useEffect will be triggered whenever the dependencies has been changes, when you provide an empty dependency array then useEffect will not be triggered for state updates where as component will be re-rendered.

//empty dependency array -> called once after component render.

//useEffect will run after the component has rendered.

//REACT ROUTER

createBrowserRouter

RouterProvider

const appRoutes = createBrowserRouter([

{path:'/',element:<AppLayout />,errorElement:<Error />},

{path:'/about',element:<About />},

]);

this is how you can mention paths and routes.

useRouteError

Link is to provide actions to links.

we have <Outlet /> for dynamic routing.

In react if you have a button that is calling a function on clicking then you should do like this

const clickHandler = () => {

}

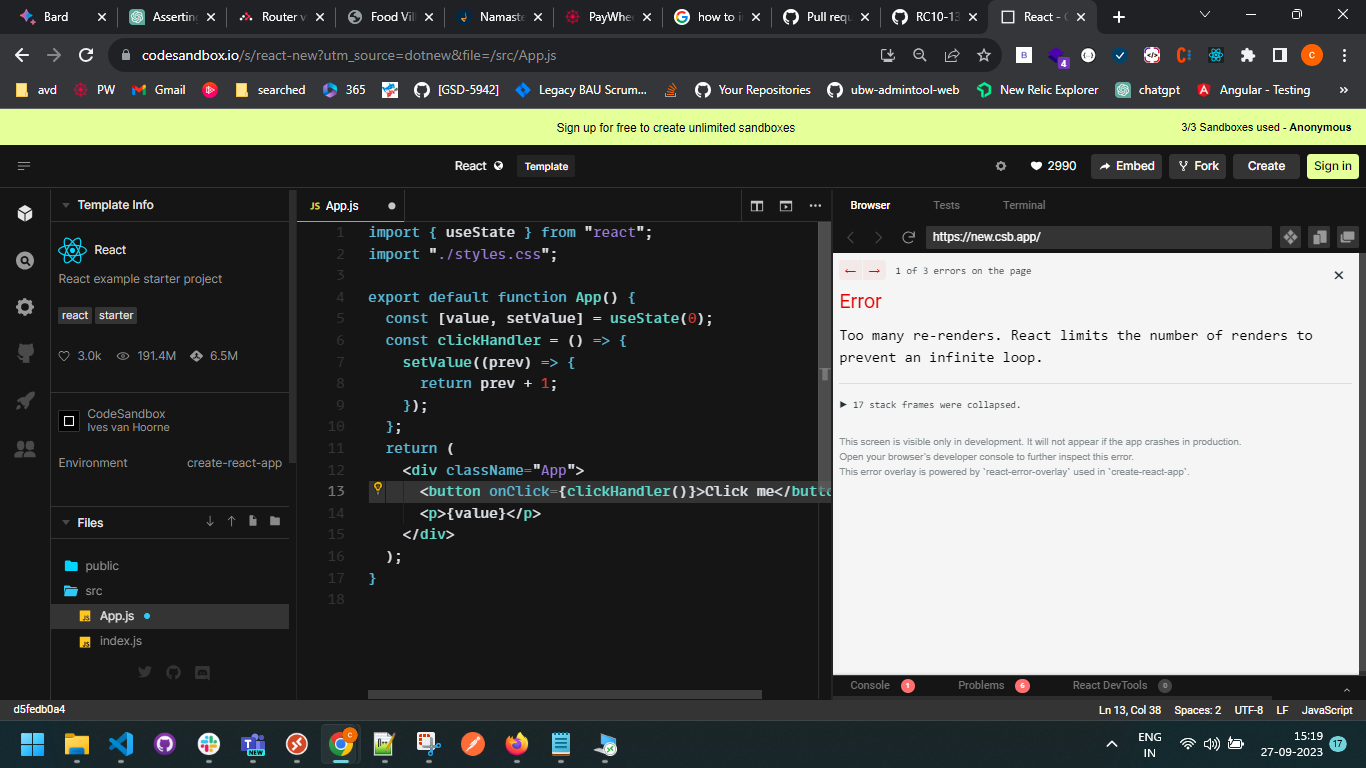
<button onClick={clickHandler} />

you should pass the reference to the click handler(onClick).

you should not do like clickHandler() because when you did like that when the component is rendered the function will be called without the actual event is being triggered that meaans without the button got clicked the function got executed that is not an expected behaviour.

this behaviour sometimes may lead to an infinite looop.

suppose lets say you have a state changing things inside a function, that function you are calling inside the button onClick as clickHandler(), so when the component renders it will calls the function then the function updates the state, when the state is updates then the component again re-renders againt the function will get called and the state updates and function called again like this the infinite loop situation arrives.



The above is the situation where infinite loop may triggered as we are calling clickHandler() instead of clickHandler, you should just pass the reference that’s it.

TAILWIND CSS

Font-bold

Text-xl, text-2xl, text-sm, text-lg, text-md, text-xl

A props is nothing but data which is being passed from one component to another componen.

"Props drilling" is a term used in React to describe a situation where you need to pass data or props through multiple levels of nested components in order to get the data to a component that needs it. It can be a common issue in React applications, especially when you have deeply nested components.

tailwind practice:

import "./styles.css";

export default function App() {

const arr = Array(15).fill("1");

return (

<div className="App">

<h1 className="text-3xl font-family: ui-monospace md:font-serif font-style: italic">

hello world

</h1>

<ul className="list-disc">

<li>This is some example of list useaage with tailwind css.</li>

<li>this is another example.</li>

</ul>

<p className="h-1000 shadow-2xl shadow-red-900 rounded-2xl opacity-100 border-2 border-green-800 text-justify hover:text-blue-900 p-2 m-2 text-red-500 whitespace-normal select-none">

So I started to walk into the water. I won't lie to you boys, I was

terrified. But I pressed on, and as I made my way past the breakers a

strange calm came over me. I don't know if it was divine intervention or

the kinship of all living things but I tell you Jerry at that moment, I

was a marine biologist.

</p>

<input

type="text"

className="text-3xl rounded-2xl p-2 m-2 caret-yellow-500 resize w-max h-1000 border-1 border-red-900"

/>

<div class="flex justify-around bg-green-500 m-2 p-2 rounded-lg">

<div className="bg-red-500">01</div>

<div className="bg-gray-500">02</div>

<div className="bg-blue-500">03</div>

</div>

<div class="bg-gray-100 p-4">

<div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-4">

<div class="bg-white p-4 rounded shadow-md">

<p>Card 1</p>

</div>

<div class="bg-white p-4 rounded shadow-md">

<p>Card 2</p>

</div>

<div class="bg-white p-4 rounded shadow-md">

<p>Card 3</p>

</div>

<div class="bg-white p-4 rounded shadow-md">

<p>Card 4</p>

</div>

</div>

<div className="flex gap-3">

<div className="flex-1 bg-red-500 p-4 rounded-xl"></div>

<div className="w-1/5 md:w-3/5 lg:flex-1 bg-red-500 p-4 rounded-xl"></div>

</div>

<div className="flex gap-1 flex-wrap justify-evenly">

{arr.map((item) => {

return (

<div className="md:w-2/4 lg:w-1/4 bg-yellow-500 p-6">Hello</div>

);

})}

</div>

<div className="grid grid-cols-2 sm:grid-cols-3 md:grid-cols-4 gap-4">

{arr.map((item) => {

return (

<div className="flex justify-center p-2 m-2 bg-blue-200 h-24 rounded sm:bg-red-100">

hello

</div>

);

})}

</div>

</div>

</div>

);

}

<div className="grid grid-cols-2 sm:grid-cols-3 md:grid-cols-4 gap-4">

{arr.map(item => {

return <div className="p-2 m-2 bg-blue-200 h-24">

hello

</div>

})}

</div>

Lifting the state up is nothing but maintaining the children’s state from the parent component, we are just moving the state from children to parent.

useContext

it is like a data storage area where we can send and use data anywhere in the application.

We have this context as follows

import { createContext } from "react";

const UserContext = createContext({

    user:{

        name:"ravi",

        email:"ravi@email.com"

    }

});

export default UserContext;

we can use it in any component like this

const user = useContext(UserContext);

{user.user.name}

We can use <UserContext.consumer>{(value)} => console.log(value) </ UserContext.consumer>

In class components.

<UserContext.Provider> to provide the value;

<UserContext.Provider value = {user}>

<Header />

<About />

<Cart />

</UserContext.Provider>

Here the context will be avilable only in header,about,cart components in the other components the context is not available as we are providing the context to those 3 components only.

REDUX  
use redux only when you are building a large scale applications which handles a lot of data is flowing in the application.

All the components will be able to access the resux store.

Redux store is a central thing, it is not tied to any component.

Unlike redux store state and props tied to our components.

In redux we have a single store for holding everything.

React components cannot directly modify redux store, we have to dispatch and action to be followed in order to modify the redux store.

If we click on cart for example then we will call a dispatch which will trigger an action which is a function which modifies our cart.

The function which is being called by the dispatcher is called reducer function.

When you click on cart it dispatches the action which is a reducer function() being called which updates a part of our redux store.

A useSelector is used to subscribe to the store.

This is the place it will tell what it is subscribing to.

You can subscribe to whatever you want, you can subscribe to the whole store, or you can subscribe to a single slice.

You can do like   
**const items = useSelector((store) => store);**

**Here you are subscribing to the whole store and you can use whatever you want from sstore using store.cart.items like that.**

**The disadvantage of subscribing to the whole store is suppose lets say your store has a cart slice and wishlist slice and you have subscribed to the whole store in cart and whenever your wishlist changes the store got re-rendered so your cart also got re-rendered even when no changes happened in cart, this is a major performance issue, when in a large application when any slice updated remaining all parts of application got re-rendered if you have subscribed to the whole application.**

**You have to subscribe to the specific portion of store.**

Configuring redux store in application.

1.create a redux store using configStore

2.create a slice using createSlice

3.provide store to the application

Higher order component is a function, it takes in a component as parameter and returns a component.

A higher order component will takes a component as a param and modifies it and returns a modified version of the component, the component which is modifying the component is called higher order component;

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

function ExpensiveCalculationComponent({ data }) {

// Using state to store an input value

const [inputValue, setInputValue] = useState('');

// Define a memoized result using useMemo

const result = useMemo(() => {

// Expensive calculation that depends on data and inputValue

console.log('Calculating result...');

return data \* parseInt(inputValue);

}, [data, inputValue]); // Dependencies array

return (

<div>

<h1>Expensive Calculation</h1>

<input

type="text"

value={inputValue}

onChange={(e) => setInputValue(e.target.value)}

placeholder="Enter a number"

/>

<p>Data: {data}</p>

<p>Input Value: {inputValue}</p>

<p>Result: {result}</p>

</div>

);

}

export default function App() {

return (

<div className="App">

<ExpensiveCalculationComponent data={10} />

</div>

);

}

We define a ExpensiveCalculationComponent that takes two props: data and inputValue.

We use the useState hook to manage the inputValue as a controlled input field.

Inside the component, we use the useMemo hook to memoize the result of an expensive calculation. The calculation depends on both data and inputValue.

We provide an array of dependencies ([data, inputValue]) to the useMemo hook. The result will be recalculated only if one of these dependencies changes.

When you type in the input field, the inputValue changes, triggering a recalculation of the result. However, if you change data, the result will also be recalculated.

Using useMemo in this way can help prevent unnecessary calculations and improve the performance of your React components.

Ref will persists it’s value between re-renders.

import { useRef, useState } from "react";

import "./styles.css";

export default function App() {

const ref = useRef(0);

let x = 0;

const [y, setY] = useState(0);

return (

<div>

<button

onClick={() => {

ref.current = ref.current + 1;

}}

>

Increase ref

</button>

<p>Ref value = {ref.current}</p>

<button

onClick={() => {

x = x + 1;

}}

>

Increase x

</button>

<p>x = {x}</p>

<button

onClick={() => {

setY(y + 1);

}}

>

Increase y

</button>

<p>y = {y}</p>

</div>

);

}

This is another example of useRef where the div or container got hide when you click anywhere in the page except on that div, if you click on that div, it wont hide where if you click anywhere in the page apart from that div then the div got disappeared, this can be seen in modals where if we open and click anywhere outside the modal the modal got closed.

The general rules of React Hooks also apply to custom Hooks; these include: Only call Hooks at the top level. Don't call Hooks inside loops, conditions, or nested functions.

The general rules of React Hooks also apply to custom Hooks; these include: Only call Hooks at the top level. Don't call Hooks inside loops, conditions, or nested functions. Only call Hooks from React function components.

https://nextui.org/docs/components/navbar