React Notes

- React is a **JavaScript library** created by Facebook in 2013.
- React is a **User Interface** (UI) library.
- React is a tool for building **UI components**
- React is used to build **single-page applications**.
- **Download Node.js** (https://nodejs.org/en)

Node.js :- Node.js is Java-script Run time environment.

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

Step-1 **Download Node.js** (https://nodejs.org/en)

Step-2 Download LTS version

Step-3 Open terminal and write “node -v” to check node version

Step-4 Open terminal and write “npx -v” to check npx version

Step-5 Open terminal and write “npm -v” to check npm version

Power Shell problem:-

Commands:

1. Get-ExecutionPolicy

2. Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass

or

3. Set-ExecutionPolicy Unrestricted

NPM:- The npm stands for **Node Package Manager** and it is the default package manager for **Node.js.**The **npm** manages all the packages and modules for node.js and consists of command-line client **npm**. The required packages and modules in the Node project are installed using **npm**. A package contains all the files needed for a module and modules are the JavaScript libraries that can be included in the Node project according to the requirement of the project.

NPX:- The npx stands for **Node Package Execute** and it comes with the npm, when you installed npm above 5.2.0 version then automatically npx will installed. It is an npm package runner that can execute any package that you want from the npm registry without even installing that package. The npx is useful during a single time use package.

NPM Package—----------Local-storage—----------Dir. To be used

NPX Package—----------Dir. To be use

Feature	npm	npx
Purpose	Manages and installs packages	Executes packages directly from the npm registry
Dependency Management	Manages project dependencies specified in package.json	Executes packages without installing them
Installation Command	npm install {your-package}	npm install -g npx
Example Command	npm install express	npx create-react-app my-app
Ideal For	Installing and managing packages	One-time use packages, running scripts, and commands

Creating react project using npx -

Step-1 , npx create-react-app fileName

Step-2 , cd (created directory)

Step-3 , npm code . (open vs-studio)

Step-4 ,npm start (start react project)

Creating react project using npm -

Step-1 , npm i create-react-app -g (only one time)

Step-2 , create-react-app fileName

Step-3 , cd (created directory**)**

Step-4 , npm code . (open vs-studio)

Step-5 ,npm start (start react project)

Creating Vite+React project using -

npm create vite@latest

npm run dev

- **React file type :-** JavaScript is a programming language used for adding interactivity to web applications. JSX stands for JavaScript XML. It is a syntax extension for JavaScript that lets you use HTML tags right inside your JavaScript file.
- **JSX is react** file extension .
- **JSX** = JavaScript + Xml

React Element	Description	Syntax

Class Element Attributes	Passes attributes to an element. The major change is that class is changed to className	
Style Element Attributes	Adds custom styling. We have to pass values in double parenthesis like {{}}	<div style= {{styleName: Value}}
Fragments	Used to create single parent component	<>//Other Components</>

ReactJS Import and Export:-

Type of Import/Export	Description	Syntax
Importing Default exports	imports the default export from modules	import MOD_NAME from “PATH”
Importing Named Values	imports the named export from modules	import {NAME} from “PATH”
Multiple imports	Used to import multiple modules can be user defined of npm packages	import MOD_NAME, {NAME} from “PATH”
Default Exports	Creates one default export. Each component can have onne default export	export default MOD_NAME

Named Exports	Creates Named Exports when there are multiple components in a single module	export default {NAME}
Multiple Exports	Exports mulitple named components	export default {NAME1, NAME2}

React Components:-

Component	Description	Syntax

Functional	Simple JS functions and are stateless	function demoComponent() {
		return (<>
		// CODE
		</>);

		}

Class-based	Uses JS classes to create stateful components	class Democomponent extends React.Component {
		
		render() {
		
		return <>//CODE</>;
		
		}
		
		}

Nested	Creates component inside another component	function demoComponent() {
		
		return (<>
		
		<Another_Component/>
		
		</>);
		
		}

Creating a React component-

. Hello.jsx (make sure your file-name first letter must be capital)

```
function Hello() {
    return '<div>Hello</div>'
}
```

Component Export -

React JS, we need to export components, functions, or values in order to use those components in another file (module). To use those exported components in another file, we have to import them first.

there are two primary ways to export and import components(or values) in React JS.

1. **Named export and import**
2. **Default export and import**

1. Default export and import :-

- Default exports are created by including a `default` tag in the export. Usually, you see default exports happen at the bottom of a file, but it's possible to define them when your component is declared.

```
const MyComponent = () => {}

export default MyComponent

// or

function MyComponent() {}

export default MyComponent

// or

export default function() {}

// or

export default () => {}
```

- When importing a default export, you don't use curly brackets.

```
import MyComponent from "./my-component"
```

- When you import a default export, you can give it whatever name you want.

```
import WhateverNameIsBest from "./my-component"
```

- modules can only have one default export.

2. Name export and import :-

- As the title suggests, named exports use the name of the function or class as their identifier.

```
export const MyFunction = () => {
  ...body
}

// or

export function MyFunction() {
  ...body
}
```

- When you want to import a named component, you use the same name it was exported with. Names must be imported inside curly brackets.

```
import { MyComponent } from "./my-component"
```

- You can rename your exports with an alias if you have collisions in a file. This is scoped to the file doing the import.

```
import { MyComponent as MyComponentAlias } from "./my-component"
```

- Named exports allow multiple exports in a single file.

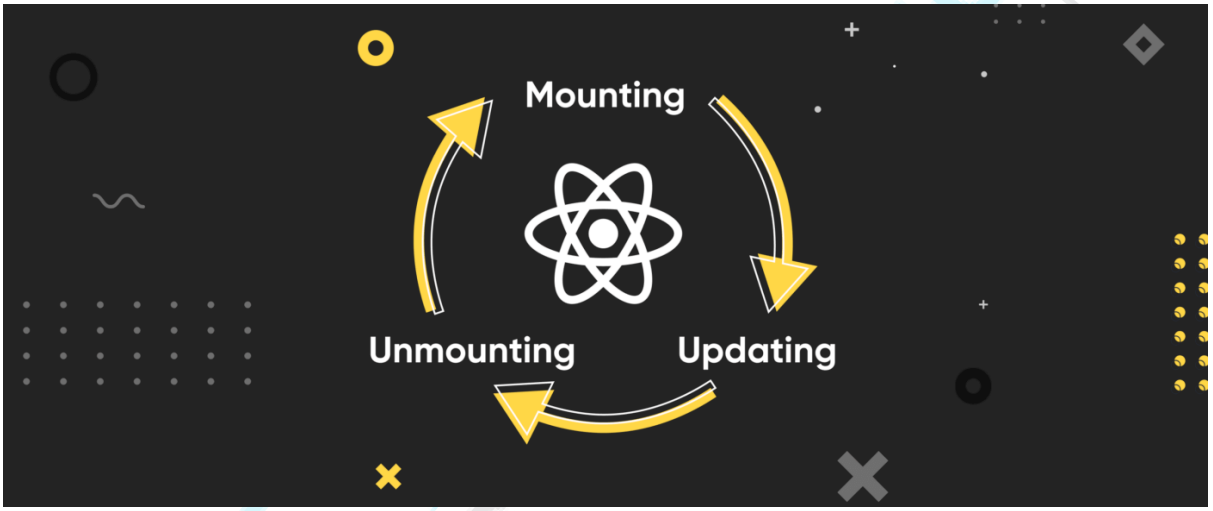
```
export const MyFunction1 = () => {};

export const MyFunction2 = () => {};

export const MyFunction3 = () => {};
```

- You can include whatever module you like in your import.

React Components LifeCycle Method:-



1. Mounting (Initialization Stage)

This is when a component is first created and added to the DOM (rendered on the screen).

- **Lifecycle Methods:**
  - **`constructor()`:** Initializes the component's state and binds methods (if necessary).

- **`render()`:** Produces the React elements to be displayed**.**
- **`componentDidMount()`:** Executes after the component is rendered in the DOM. Useful for side effects like fetching data or setting up subscriptions.

2. Updating (Growth/Modification Stage)

This stage happens whenever the component's state or props change. React re-renders the component to reflect the updates.

- **Lifecycle Methods**:
  - `shouldComponentUpdate(nextProps, nextState)`: Decides if re-rendering is needed. Optimizes performance.
  - `render()`: Re-renders the component when state or props change.
  - `componentDidUpdate(prevProps, prevState)`: Executes after updates. Useful for side effects based on updated data.

3. Unmounting (Cleanup Stage)

This occurs when the component is removed from the DOM. Cleanup tasks (e.g., unsubscribing from events or timers) are performed here.

- **Lifecycle Method**:
  - `componentWillUnmount()`: Executes just before the component is removed. Used for cleanup.

Propes

Props (short for properties) are a way to pass data from a parent component to a child component in React. They are read-only and immutable, meaning the child component cannot modify them. Props make components reusable by allowing dynamic data to be passed in.

Key Features of Props:

1. **Immutable**: Props cannot be modified by the child component.
2. **Read-Only**: Data passed via props is for display or use, not modification.
3. **Unidirectional Data Flow**: Props flow from parent to child (top-down).

Parent Component:-

```
function App() {

    return (

      <div>

        <Greeting name="Ws Cube Tech" age={25} />

        <Greeting name="Website" age={30} />

      </div>

    );

  }
```

Child Component-

```
  function Greeting(props) {

    return (

      <h1>

        Hello, {props.name}! You are {props.age} years old.

      </h1>

    );

  }
```

Props.children :-

Parent Component:-

```
function App() {
    return (
      <Wrapper>
        <h1>Hello, World!</h1>
        <p>This is a child element inside the Wrapper component.</p>
      </Wrapper>
    );
  }
```

Child Component:-

```
function Wrapper({children}) {
    return (
        <div className="wrapper">{children}</div>
    )
  }
```

React Bootstrap -

Package :-

. npm install bootstrap

. npm install react-bootstrap bootstrap

Import file in **index.js** -

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

https://react-bootstrap.netlify.app/docs/getting-started/introduction

React + Vite Tailwind-

https://tailwindcss.com/docs/guides/vite

Package :-

npm install -D tailwindcss postcss autoprefixer

npx tailwindcss init -p

Import file in **tailwind.config.js**

```
/** @type {import('tailwindcss').Config} */

content: [
      "./index.html",
      "./src/**/*.{js,ts,jsx,tsx}",
    ],
    theme: {
      extend: {},
    },
    plugins: [],
  }
```

Import file in **index.css**

```
  @tailwind base;
@tailwind components;
@tailwind utilities;
```

State Management -

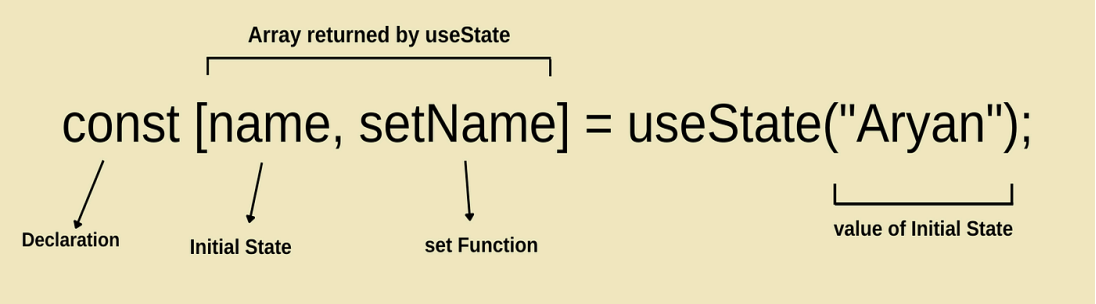
- State management is one of the important and unavoidable features of any dynamic application. React provides a simple and flexible API to support state management in a React component.
- **State management using React Hooks .**

What is a Hook in React?

A Hook in React is a special function that lets you use state and other React features in functional components. Hooks were introduced in React 16.8 to enable the use of features like state management, lifecycle methods, and context without needing to write class components.

1.useState():-

The `useState` Hook is one of the most commonly used React Hooks. It allows you to add state to functional components. Before `useState`, state management was only possible in class components. With `useState`, you can create and manage local state directly in functional components.




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

function Counter() {
    const [count, setCount] = useState(0);

    const increment = () => setCount(count + 1);
    const decrement = () => setCount(count - 1);

    return (
      <div>
        <p>Count: {count}</p>
        <button onClick={increment}>Increment</button>
        <button onClick={decrement}>Decrement</button>
      </div>
    );
  }

  export default Counter;
```

2 .useEffect():-

The `useEffect` Hook allows you to perform side effects in functional components. Side effects can include data fetching, subscriptions, manually

changing the DOM, and logging. The `useEffect` Hook runs after the component renders, making it the perfect place for any code that should run in response to a change in the component's state or props.

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

function ExampleComponent() {
  useEffect(() => {
    console.log("Component has been rendered or updated.");

    // Cleanup function (optional)
    return () => {
      console.log("Cleanup before next effect or unmount.");
    };
  }); // No dependency array, so it runs after every render

  return <h1>Hello, World!</h1>;
}

export default ExampleComponent;
```

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

  function DataFetchingComponent() {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);

    useEffect(() => {
      const fetchData = async () => {
        try {
          const response = await
fetch("https://api.example.com/data");
          if (!response.ok) {
            throw new Error("Network response was not ok");
          }
          const result = await response.json();
          setData(result);
```

```
            } catch (error) {
              setError(error);
            } finally {
              setLoading(false);
            }
          };

          fetchData();
        }, []); // Empty dependency array means this effect runs once
(on mount)

        if (loading) return <p>Loading...</p>;
        if (error) return <p>Error: {error.message}</p>;

        return (
          <div>
            <h1>Fetched Data:</h1>
            <ul>
              {data.map((item) => (
                <li key={item.id}>{item.name}</li>
              ))}
            </ul>
          </div>
        );
      }

      export default DataFetchingComponent;
```

3. useID():-

The `useId` Hook is a relatively new addition to React (introduced in React 18) that helps manage unique IDs for components. This is especially useful for ensuring that IDs are stable and do not cause issues with server-side rendering or client-side hydration.

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

```
function AccessibleForm() {
  const id = useId(); // Generate a unique ID

  return (
    <form>
      <div>
        <label htmlFor={`${id}-name`}>Name:</label>
        <input type="text" id={`${id}-name`} />
      </div>
      <div>
        <label htmlFor={`${id}-email`}>Email:</label>
        <input type="email" id={`${id}-email`} />
      </div>
      <button type="submit">Submit</button>
    </form>
  );
}

export default AccessibleForm;
```

4. useRef():-

The useRef Hook in React is a versatile tool used primarily for accessing and interacting with DOM elements, but it can also hold mutable values similar to instance variables in class components. It allows you to create a reference that persists for the full lifetime of the component without causing re-renders when its value changes.

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

function FocusInput() {
  const inputRef = useRef(null); // Create a ref

  useEffect(() => {
    // Focus the input field when the component mounts
```

```
        if (inputRef.current) {
          inputRef.current.focus();
        }
      }, []);

      return (
        <div>
          <input ref={inputRef} type="text" placeholder="Focus me!"
/>
        </div>
      );
    }

    export default FocusInput;
```

Api Fetch :-

Fetching data from an API in React is a common task, often performed using the `useEffect` Hook along with the `fetch` API or other libraries like Axios. Below, I'll outline how to fetch data from an API in a React component, including examples of error handling and loading states.

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

      function DataFetchingComponent() {
        // State variables to hold data, loading status, and error
        const [data, setData] = useState([]);
        const [loading, setLoading] = useState(true);
        const [error, setError] = useState(null);

        useEffect(() => {
          const fetchData = async () => {
            try {
              // Start the fetch operation
              const response = await
fetch("https://api.example.com/data");

              // Check if the response is ok (status in the range
200-299)
```

```
                if (!response.ok) {
                  throw new Error("Network response was not ok " +
response.statusText);
                }

                // Parse the JSON response
                const result = await response.json();
                setData(result); // Set the fetched data to state
              } catch (err) {
                // Handle errors
                setError(err);
              } finally {
                // Set loading to false once the fetch is complete
                setLoading(false);
              }
            };

            fetchData(); // Call the fetch function
          }, []); // Empty dependency array means this effect runs once
on mount

          // Render loading, error, or data
          if (loading) return <p>Loading...</p>;
          if (error) return <p>Error: {error.message}</p>;

          return (
            <div>
              <h1>Fetched Data:</h1>
              <ul>
                {data.map((item) => (
                  <li key={item.id}>{item.name}</li>
                ))}
              </ul>
            </div>
          );
        }

        export default DataFetchingComponent;
```

Using Axios:-

Using Axios for making HTTP requests in a React application is straightforward and provides several advantages over the native `fetch` API. Here's a detailed guide on how to use Axios, including installation, basic usage, and examples for common scenarios.

npm install axios

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

function DataFetchingComponent() {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        // Make a GET request
        const response = await
axios.get("https://api.example.com/data");
        setData(response.data); // Set the response data to state
      } catch (err) {
        setError(err); // Handle error
      } finally {
        setLoading(false); // Set loading to false after request
completes
      }
    };

    fetchData(); // Call the fetch function
  }, []); // Empty array means this effect runs only once

  // Conditional rendering
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error.message}</p>;
```

```
    return (
      <div>
        <h1>Fetched Data:</h1>
        <ul>
          {data.map((item) => (
            <li key={item.id}>{item.name}</li>
          ))}
        </ul>
      </div>
    );
  }

  export default DataFetchingComponent;
```

Controlled and Uncontrolled Components

- **Controlled Components: In controlled components, the form data is handled by the component's state. The state serves as the single source of truth for the input fields.**

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

function ControlledInput() {
  const [value, setValue] = useState("");

  return (
    <input
      type="text"
      value={value}
      onChange={(e) => setValue(e.target.value)}
    />
  );
}

export default ControlledInput;
```

- **Uncontrolled Components: Uncontrolled components store their own state internally and are not controlled by React. You can access the current value of the input field using a ref.**

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

function UncontrolledInput() {
  const inputRef = useRef(null);

  const handleSubmit = (e) => {
    e.preventDefault();
    alert("Input value: " + inputRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}

export default UncontrolledInput;
```

React Routing -

- Linking multiple web pages.
- Package - npm i react-router-dom
- Example:- create a folder named **component** inside the src folder and now add 3 files named.
    - ★ Home.jsx
    - ★ About.jsx
    - ★ Course.jsx

★ Blog.jsx

★ Gallery.jsx

★ Create a common file (Header.jsx)

- Work In Header file -

```
return(
    <>
      <Link to={'/'}>Home</Link>
      |
       <Link to={'/about'}>About</Link>
       |
       <Link to={'/course'}>Course</Link>
       |
      <Link to={'/Blog'}>blog</Link>
       |
        <Link to={'/gallery'}>Gallery</Link>
    </>
  )
```

- Work InMain.jsx (static) -

```
const root = ReactDOM.createRoot(document.getElementById('root'));
let routes=createBrowserRouter([
 {
   path: "/",
   element:<Home/>,
 },
   {
     path: "about",
   element:<About/>,
   },

   {
     path: "gallery",
    element:<Gallery/>,
   },

   {
```

```
    path: "Blog",
    element:<Bloglist/>,
   },
   {
    path: "Blog/:id",
    element:<Blog/>,
   },

  {
   path: "course",
   element:<Course/>,
  },

])
root.render(
  <React.StrictMode>
    <RouterProvider router={routes} />
  </React.StrictMode>
);
```

- **Work In Main.jsx (Dynamic)-**

```
const root = ReactDOM.createRoot(document.getElementById('root'));
let routes=createBrowserRouter([
 {
   path: "/",
   element:<Home/>,
 },
   {
    path: "Product/ID:",
   element:<div-any/>,
   },
])
root.render(
 <React.StrictMode>
```

```
    <RouterProvider router={routes} />
  </React.StrictMode>
);
```

React Outlet Routing -

Outlet routing in Vite is typically implemented using a routing library like React Router if you're working with React. Outlet routing allows you to define nested routes, where a parent route renders an `Outlet` component that serves as a placeholder for child routes

1. Create Layout.jsx

```
import React from 'react'
import Header from './Header'
import { Outlet } from 'react-router-dom'

export default function Layout() {
  return (
    <div>
      <header> <Header/> </header>
      <main> <Outlet/> </main>

    </div>
  )
}
```

2. Import main.jsx

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import SingleProduct from './SingleProduct.jsx'
import { createBrowserRouter, RouterProvider } from 'react-router-dom'
```

```
import NotFound from './NotFound.jsx'
import Layout from './Layout.jsx'
import Cart from './Cart.jsx'
import MainContext from './MainContext.jsx'

const ws = createBrowserRouter([
  {
    path: "/",
    element: <Layout />,
    children: [
      {
        path: "/",
        element: <App />
      },
      {
        path: "/detail-page/:id?",
        element: <SingleProduct />
      },

      {
        path: "/cart",
        element: <Cart />
      },

    ]

  },
  {
    path: "*",
    element: <NotFound />
  }

])
createRoot(document.getElementById('root')).render(
  <StrictMode>

    <MainContext>
      <RouterProvider router={ws} />
    </MainContext>
```

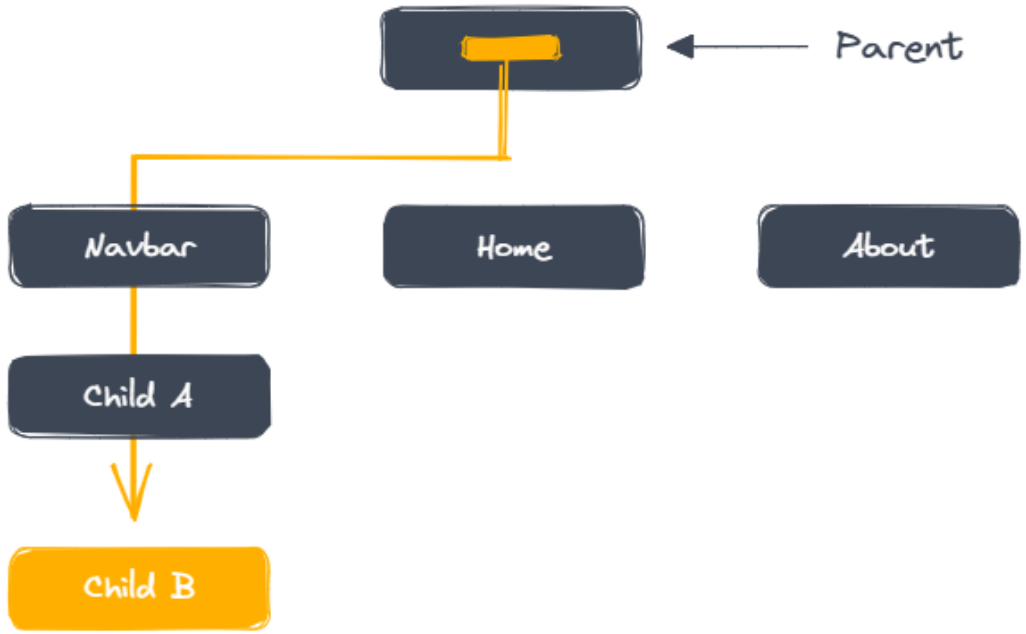
```
  </StrictMode>,
)
```

React Router's Declarative Routing:-

```
ReactDOM.createRoot(root).render(
  <BrowserRouter>
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="dashboard" element={<Dashboard />}>
        <Route index element={<RecentActivity />} />
        <Route path="project/:id" element={<Project />} />
      </Route>
    </Routes>
  </BrowserRouter>
);
```

Context API -

- **React Context is a method to pass props from parent to child component(s) and child to parent component.**
- **Context API allows data to be passed through a component tree without having to pass props manually at every level. This makes it easier to share data between components.**



-
- **Context Api-**
    - provider—>CreateContext()
    - Consumer—>useContext()

```
let counterContex=createContext();

export default function Maincontext(props) {

   let[counter,setcounter]=useState(0)

  return (

     <counterContex.Provider value={{counter,setcounter} }>

       {props.children}

    </counterContex.Provider>

  )

}

export {counterContex}
```

```
import { counterContex } from './context/Maincontext'

export default function Button({symble}) {

 let {counter,setcounter} =useContext(counterContex)

 let updateCounter=()=>{

  if(symble=="+"){

   setcounter(counter+1)

  }

  else{

   setcounter(counter-1)

  }

 }

 return (

   <div>

     <button onClick={updateCounter}>{symble}</button>

   </div>

 )

}
```

Next.js

Next.js is a React framework that enables server-side rendering (SSR) and static site generation (SSG), offering better performance, SEO, and developer experience. It's built on top of React and enhances it with additional features.

Key Features:

1. **File-based Routing**: Define routes by creating files in the `pages` directory.
2. Server-Side Rendering (SSR): Pre-renders pages on the server for better SEO and performance.
3. Static Site Generation (SSG): Generates static pages at build time.
4. API Routes: Build backend APIs directly in the `pages/api` directory.
5. Fast Refresh: Improved development experience with real-time updates.
6. Image Optimization: Built-in support for optimized images.
7. CSS & Styling: Supports CSS Modules, TailwindCSS, and other CSS frameworks.
8. Deployment: Easy deployment with platforms like Vercel (the company behind Next.js).
9. npm run dev (project run command)

When to Use Next.js?

- For websites that need **SEO optimization**.
- Applications requiring server-side logic or pre-rendered content.
- Projects requiring high performance with modern React features.

Routing in Next.js (App Router)

Next.js App Router (introduced in Next.js 13) provides a new and modern way to handle routing with the app/ directory. It focuses on server-side rendering, React Server Components, and simplifies the development of dynamic and nested routes.

```
app/
  layout.js         → Shared layout for all pages
  page.js           → Renders at `/` (Home page)
  about/
    page.js         → Renders at `/about`
  gallery/
    page.js         → Renders at `/gallery`
  contact/
    page.js         → Renders at `/contact`
```

layout.js:-

```
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <header>
          <nav>
            <ul>
```

```
            <li><Link href="/">Home</Link></li>
            <li><Link href="/about">About</Link></li>
            <li><Link href="/gallery">Gallery</Link></li>
            <li><Link href="/contact">Contact</Link></li>
          </ul>
        </nav>
      </header>
      <main>{children}</main>
      <footer>© 2024 My App</footer>
    </body>
  </html>
 );
}
```

Dynamic routes:-

Dynamic routes in Next.js enable you to create flexible paths based on dynamic parameters. They are particularly useful when dealing with data that changes

1. How to Create Dynamic Routes:-A Dynamic Segment can be created by wrapping a folder's name in square brackets: `[folderName]`. For example, `[id]` or `[slug]`.

a blog could include the following route `app/blog/[slug]/page.js` where `[slug]` is the Dynamic Segment for blog posts.

- **Use `[param]` to match paths a.**

2.Catch-All and Optional Catch-All Routes

Catch-All Routes

- **Use `[...param]` to match all paths after a base path.**
- **Example:**
  - **File: `pages/docs/[...slug].js`**
  - **Routes: `/docs/a`, `/docs/a/b`, `/docs/a/b/c`**

3.Optional Catch-All Routes

- **Use `[[...param]]` to make the parameter optional.**
- **Example:**
  - **File: `pages/docs/[[...slug]].js`**
  - **Routes: `/docs`, `/docs/a`, `/docs/a/b`**

When to Use `"use client"`:

1. Browser-Specific Features:
   Use `"use client"` when the component depends on browser-specific APIs like `window`, `localStorage`, `document`, or events like `onClick` or `onChange`.
2. Interactivity:
   If your component includes hooks like `useState`, `useEffect`, or `useContext` (React Client Hooks), it must be a client component.

3. Dynamic Behavior:
   Components that require user interactions (e.g., buttons, forms, modals) should use `"use client"`.
4. Third-Party Libraries:
   If you're using libraries that manipulate the DOM directly or are browser-only, such as `react-datepicker` or `chart.js`, those components need `"use client"`.

```
'use client';

import { useState } from 'react';

export default function Counter() {

  const [count, setCount] = useState(0);

  return (

    <div>

      <p>Count: {count}</p>

      <button onClick={() => setCount(count + 1)}>Increment</button>

    </div>

  );

}
```

Things to Remember:

1. **Nested Components:**

- If a parent component is marked as `"use client"`, all its child components will be treated as client components, even if they don't have `"use client"` specified.

2. **Performance Implications:**
   - Use `"use client"` only when necessary, as server components are more performant since they don't ship JavaScript to the browser.
3. **Server vs Client Components:**
   - Use server components for tasks like fetching data (to reduce client-side payload) and client components for interactivity.

FireBase

Firebase is a platform developed by Google that provides a suite of cloud-based tools and services to help developers build, improve, and grow web and mobile applications. It simplifies the backend of applications, enabling developers to focus more on the front-end and core features without worrying about server management or complex infrastructure.

Core Features of Firebase

1. **Database Solutions:**
   - **Realtime Database**: A NoSQL cloud-hosted database that stores and syncs data in real time across all connected clients.
   - **Cloud Firestore**: A scalable, flexible, and cloud-hosted NoSQL database designed for more complex queries and better performance.
2. **Authentication**:

- Provides easy-to-use authentication services like email/password, phone authentication, and third-party login providers (Google, Facebook, GitHub, etc.).

3. **Cloud Storage**:
   - A powerful, simple, and cost-effective object storage solution designed for storing and serving large files such as images, videos, and other user-generated content.
4. **Hosting**:
   - Offers fast and secure web hosting for static and dynamic content, including integration with single-page applications and server-side rendering frameworks like Next.js.
5. **Cloud Functions**:
   - Allows you to run backend code in response to events triggered by Firebase features or HTTPS requests without managing servers.
6. **Analytics**:
   - **Google Analytics for Firebase** provides detailed insights into app performance and user behavior.
7. **Push Notifications**:
   - **Cloud Messaging** lets you send targeted notifications and messages to users on web, iOS, and Android platforms.
8. **Crash Reporting**:
   - **Crashlytics** helps track, prioritize, and fix app crashes in real time.
9. **Performance Monitoring**:
   - Monitor your app's performance to ensure a seamless user experience.
10. **Remote Config**:

- Dynamically change app behavior and appearance without releasing a new version.

How to Start Using Firebase

1. Go to Firebase Console.
2. Create a new project.
3. Choose the Firebase products you want to use.
4. Integrate Firebase into your app by adding the provided SDK and configuration.

To implement Google Login with Firebase, follow these steps:

1. Setup Firebase

1. Go to the Firebase Console.
2. Create a project or select an existing one.
3. Navigate to **Authentication** > **Sign-in Method**.
4. Enable the **Google** sign-in provider and configure the credentials (Client ID and Client Secret from the Google Cloud Console).

2. Install Firebase in Your Project:

npm install firebase

3. Configure Firebase:(firebaseConfig.js)

```
// firebaseConfig.js

import { initializeApp } from "firebase/app";

import { getAuth, GoogleAuthProvider, signInWithPopup } from
"firebase/auth";
```

```
const firebaseConfig = {
  apiKey: "YOUR_API_KEY",
  authDomain: "YOUR_AUTH_DOMAIN",
  projectId: "YOUR_PROJECT_ID",
  storageBucket: "YOUR_STORAGE_BUCKET",
  messagingSenderId: "YOUR_MESSAGING_SENDER_ID",
  appId: "YOUR_APP_ID",
};

const app = initializeApp(firebaseConfig);
const auth = getAuth(app);
const provider = new GoogleAuthProvider();

export { auth, provider, signInWithPopup };
```

4. Create the Google Login Functionality

```
    import React from "react";
import { auth, provider, signInWithPopup } from "./firebaseConfig";
```

```
export default function GoogleLogin() {

  const handleGoogleSignIn = async () => {

    try {

      const result = await signInWithPopup(auth, provider);

      const user = result.user;

      console.log("User Info:", user);

      alert(`Welcome ${user.displayName}`);

    } catch (error) {

      console.error("Error during sign-in:", error.message);

    }

  };



  return (

    <div>

      <button onClick={handleGoogleSignIn}>Sign in with Google</button>

    </div>

  );

}
```

Firebase Realtime Database

Firebase Realtime Database allows you to store and sync data between your app and the cloud in real time. Here's how to integrate and use Firebase Realtime Database in your project:

Firebase Realtime Database allows you to store and sync data between your app and the cloud in real time. Here's how to integrate and use Firebase Realtime Database in your project:

---

1. Setup Firebase

1. Go to the Firebase Console.
2. Create a project or select an existing one.
3. Navigate to **Build** > **Realtime Database**.

2. Write Data to Realtime Database

```
import { database } from "./firebaseConfig";
import { ref, set } from "firebase/database";


function writeData(userId, name, email) {
  set(ref(database, 'users/' + userId), {
    username: name,
    email: email,
  })
    .then(() => {
      console.log("Data written successfully!");
```

```
    })
    .catch((error) => {
      console.error("Error writing data:", error);
    });
}
```

3. Read Data from Realtime Database

```
import { database } from "./firebaseConfig";
import { ref, onValue } from "firebase/database";

function readData() {
  const userRef = ref(database, 'users/user1');
  onValue(userRef, (snapshot) => {
    const data = snapshot.val();
    console.log("Data retrieved:", data);
  });
}
```

Redux Toolkit

Redux Toolkit (RTK) is an **official library for Redux** that simplifies the process of managing state in applications. It provides tools and best practices to make Redux development easier, faster, and less prone to boilerplate.

https://redux-toolkit.js.org/

Key Features of Redux Toolkit

1. **Simplified Store Setup**:
   - Use `configureStore` to set up a Redux store with default middleware, DevTools, and reducers.
2. **Code Reduction**:
   - Drastically reduces boilerplate code by combining actions, reducers, and constants into slices.
3. **Async Logic Handling**:
   - Provides `createAsyncThunk` for handling asynchronous tasks like API calls.
4. **Built-in Middleware**:
   - Comes with default middleware, including `redux-thunk` for async logic.
5. **Immutability**:
   - Uses Immer.js to simplify writing immutable updates.
6. **Best Practices Enforced**:
   - Encourages writing clean and maintainable Redux code.
7. **Developer-Friendly Tools**:

- Includes helpful utilities like `createSlice`, `createEntityAdapter`, and default DevTools integration.

Core APIs and Their Usage

1. configureStore

Simplifies store creation with sensible defaults.

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});

export default store;
```

2. createSlice

Combines actions and reducers into a single structure.

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
```

```
      state.value -= 1;
    },
  },
});

export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

3. createAsyncThunk

Simplifies asynchronous actions, such as API calls.

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit';

export const fetchData = createAsyncThunk('data/fetchData', async () =>
{
  const response = await fetch('/api/data');
  return response.json();
});

const dataSlice = createSlice({
  name: 'data',
  initialState: { items: [], status: 'idle' },
  extraReducers: (builder) => {
    builder
      .addCase(fetchData.pending, (state) => {
        state.status = 'loading';
      })
      .addCase(fetchData.fulfilled, (state, action) => {
        state.status = 'succeeded';
        state.items = action.payload;
      })
      .addCase(fetchData.rejected, (state) => {
        state.status = 'failed';
      });
  },
});
```

```
export default dataSlice.reducer;
```

4. createEntityAdapter

Manages normalized state for collections of items efficiently.

```
import { createEntityAdapter, createSlice } from '@reduxjs/toolkit';

const itemsAdapter = createEntityAdapter();

const itemsSlice = createSlice({
  name: 'items',
  initialState: itemsAdapter.getInitialState(),
  reducers: {
    addItem: itemsAdapter.addOne,
    removeItem: itemsAdapter.removeOne,
  },
});

export const { addItem, removeItem } = itemsSlice.actions;
export default itemsSlice.reducer;
```

RTK Workflow

1. **Install Redux Toolkit**:
   npm install @reduxjs/toolkit react-redux
2. **Create the Store**: Use `configureStore` to define the Redux store.
3. **Define Slices**: Use `createSlice` to manage state and define actions.

4. **Use Async Logic**: If needed, use `createAsyncThunk` to handle API calls or other async operations.
5. **Provide Store**: Wrap your app with the `Provider` component from `react-redux`.
6. **Access State and Dispatch Actions**:
    - Use `useSelector` to read state.
    - Use `useDispatch` to dispatch actions.
7. Example:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './counterSlice';

function Counter() {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <p>{count}</p>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
    </div>
  );
}

export default Counter;
```