**React Installation**

1. Next.js is a framework of react and react is promoting to shift there for production purposes.

2. Now we can not setup react like old times. Now have to use VITE, a powerful tool to install react.

3. Installation steps:



Steps: npm create vite > React > Just JS

**Shortcut: rsc**

**React facts**

1. React is made to make Developer life easy. Not performance Optimization.
2. React is a javascript object
3. React creates a separate dom which is known as virtual DOM
4. …num , is a spread operator. It is used to copy the whole thing. Like, arr = [1,2,3] narr = [...arr]
5. JSX simplifies react. Like:

const element = React.createElement("div", null,

[React.createElement("p",null, "hello world"),

React.createElement("p",null, "hello world"),

React.createElement("p",null, "hello world")]);

Simplified:

<div> <p> hello world </p> <p> hello world </p> <p> hello world </p> </div>

1. Babel transpiles XML into Javascript
2. State is an important thing in React that helps to simplify process through state variable.
3. State is an array of var and function. So, in [var, function], function helps to update the value of var
4. 2 types of rendering – Client side rendering and server side rendering

**Virtual DOM**

DOM is fast but the process of showing the change of DOM is slow.



To make it fast, we can do:

1. Optimize Batch update: As an example, for a 10000 time loop, if we do batch update (div.innerHTML=value) in every loop, then it gets much slower. But after the loop, if we do the batch update, then it takes no time. So, without, virtualDOM, we can optimize batch update to make the site faster.

2. Less DOM operation

Here comes Virtual DOM. It acts like a rough DOM sheet to use reconciliation algorithm to do the exact change and make the operation much faster.



Example:



Here, when we insert gggggg, the normal DOM changes every element. But Virtual DOM only changes the element after gggggg or the necessary parts by comparing.

\*\*Virtual DOM compares between previous UI and new UI and render only the changed part, not the whole page.

**Start with react**

1. Run with vite: npm run dev / npx vite
2. Build react project: npm run build / npx vite build

\*vite combines the whole scattered project and centralize them for deployment

**File Description and Components**

We are creating single page application using react. That means, only one index.html. And single page aplication loads only once.

vite.config.js – Auto generated file. Nothing to change in general

**package.json** – All used package and the version with every description

* dependencies packages will go for final production not devdependencies
* when downloading the node modules, the files are downloaded according to the dependencies and devdependencies and also auto-generates package-lock.json

package-lock.json – nothing to change

index.html – the whole application stays inside the div of root. And it loads only once. So, performance enhances.

.gitignore – the things that won’t be pushed into the github

**Src** – The whole application stays. The main place to work.

* Main.jsx: is the starting file to run the react app
* Index.css: auto generated
* App.jsx: this is connected with the main.jsx. There is a function
* App.css: auto generated
* Assets: auto generated

Public – by default vite logo

Node\_module - All dependency package files installed, nothing to work with

dist – nothing to work on

Organize react files

1. Remove index.css, App.css with the link in main.jsx
2. App.jsx:

function App() {

  return (

    <>

    <h1>React App</h1>

    </>

  )

}

export default App

1. In assets, delete react.svg and create css and images folder
2. In scr, create pages and component folder

VS code extentions:

1. Auto close tag
2. Auto Import - ES6, TS, JSX, TSX
3. Auto Rename Tag
4. ESLint – for auto correction and suggestion
5. npm Intellisense – help to import from node modles
6. Path Intellisense – auto path suggestion
7. Postman – simulate rest api
8. Prettier
9. Reactjs code snippets – provide shortcut ex: rsc, rcc
10. Snipped – screenshot – ctrl+shift+p
11. Stylelint – fix css mistakes
12. Tailwind CSS IntelliSense – suggests css class
13. Thunder client – same like postman
14. VSCode React Refactor – helps to put something in variable or function
15. vscode-icons

Newly created files:

Inside src/

/assets/css, /assets/images

/component

/pages

So, basic file structure in src –



Difference between html and **JSX**:

1. we can directly use js **inside, {}**
2. Always return a **single parent element**. Example: everything inside <div><div/>, <span><span/> etc.
3. There is a **self close** option for tags. Ex: <img/>, which is not in html
4. Here, we have to use **‘className’** instead of ‘class’ in html
5. All attribute has to be camel case. In html, onclick. But in JSX, **onClick**
6. We used inline css in html(style=”color:red;”). But we have to make it object here**(style={{color:’red’}})**.

JSX essential:

1. Inline if else –

      {marks>80?<h1>Brilliant Result</h1>:<h1>Average result</h1>}

1. Immediately invoke (this execute immediately) –

        {(()=>{

          //Write any js code. But return it to show, as this is a function.

        })()}

1. Loop –

          {

            arr.map((item,i)=>{

              return<li key={i.toString()}>{item}</li>

              //key is to catch the iteration number

            })

          }

Here, we are using .map instead of for or while loop because .map returns something. Where for and while just executes codes.

1. **Conditional rendering** (after login, we will see logout. Vice-versa) –

We can do this by using

a. if else,

b. switch statement,

c. ternery operator,

d. logical &&,

e. immediately invoke function

1. if else 🡪 problem: expensive, as code is too much

  if (status==true){

    return (

      <div>

        <h1>Login Status</h1>

        <button>Logout</button>

      </div>

    );

  }

  else{

    return (

      <div>

        <h1>Logout Status</h1>

        <button>Login</button>

      </div>

    );

  }

Because of (problem: expensive, as code is too much) the problem, we have to reduce code. So –

const LoginStatus=(status)=>{

    if (status){

        return <button>Logout Btn</button>

    }

    else{

        return <button>Login Btn</button>

    }

}

const CDRen\_ifelse = () => {

    return (

        <div>

            <h1>Login status</h1>

            {LoginStatus(false)}

        </div>

    );

};

export default CDRen\_ifelse;

1. switch 🡪

const header = ()=>{

    return <h1>Login status(switch)</h1>

};

const CDRen\_switch = () => {

    const status = true;

    switch(status){

        case true:

            return (

                <div>

                    {header()}

                    <button>Logout</button>

                </div>

            )

        case false:

            return (

            <div>

                {header()}

                <button>Login</button>

            </div>)

        default:

            return null

    }

};

export default CDRen\_switch;

1. ternary 🡪

import React from 'react';

const CDRen\_ternary = () => {

    let status = true

    return (

        <div>

            <h1>Login status(Ternary)</h1>

            {

                status?

                <button>Logout</button>

                :

                <button>Login</button>

            }

        </div>

    );

};

export default CDRen\_ternary;

1. logical &&(only one condition and clean) 🡪

import React from 'react';

const CDRen\_ternary = () => {

    let status = true

    return (

        <div>

            <h1>Login status(Ternary)</h1>

            {

                status?

                <button>Logout</button>

                :

                <button>Login</button>

            }

        </div>

    );

};

export default CDRen\_ternary;

1. Invoke function(the main moto is conditional rendering. Executes by its own after initiating “{(()=>{})()}” ) 🡪
2. import React from 'react';
3. const CDRen\_Im\_invoke = () => {
4. const status = true;
5. return (
6. <div>
7. <h1>Login status (Immediately invoke) </h1>
8. {(()=>{
9. if(status==true){
10. return <button>Logout Button</button>
11. }
12. else{
13. return <button>Login Button</button>
14. }
15. })()}
16. </div>
17. );
18. };
19. export default CDRen\_Im\_invoke;

Passing PROPS:

Props is properties. It can be variable, or string or anything. Now passing props is **passing the components of parent to child**. That means a component of parent can be accessed by all of it’s child. This is **unidirectional**. That means, data will go from parent to child. And the child can’t change the data of parent. So, it is **readonly**.

Passing props 🡪

PARENT.jsx

import Child1 from "./component/Child1";

const PARENT = () => {

    return (

        <div>

            <Child1 title="This is passing props"

            description="This passing props description"/>

        </div>

    );

};

export default PARENT;

Child1.jsx

import React from 'react';

const Child1 = (props) => {

    return (

        <div>

            <h1>{props.title}</h1>

            <h2>{props.description}</h2>

        </div>

    );

};

export default Child1;

Passing json obj 🡪

PARENT.jsx

const ItemObj = {

        name: "Joy",

        age: 76,

        city:"Dhaka"

    }

    return (

        <div>

            <Child1

            item={ItemObj}

            title="This is passing props"

            description="This passing props description"/>

        </div>

Child.jsx

            <ul>

                <li>Name: {props.item['name']}</li>

                <li>Age: {props.item['age']}</li>

                <li>City: {props.item['city']}</li>

            </ul>

Passing function 🡪

PARENT.jsx

const func = ()=>{

        alert("Function passed!")

    }

    return (

        <div>

            <Child1

            BtnClick={func}/>

Child1.jsx

<button onClick={props.BtnClick}>Pass Function</button>

Responding 2 events –

1. **Button Click event management:**
2. Direct execute on load

<button onClick={alert("Click")}>Onclick handler direct</button>

1. Execute after click

<button onClick={()=>{alert("Click")}}>Onclick handler after click</button>

1. Execute after click but for normal function

    function Demo(){

        alert('Normal Function')

    }

            <button onClick={Demo}>Onclick handler noraml function</button>

1. **Form submission event management:**

Problem – submit reloads everytime. Which is a big concern.

            <form>

                <input/>

                <button type="submit">Submit</button>

          </form>

Solve – Use event.preventDefault() . Because it is a natural behavior of form to reload after submit. But this function prevents that to happen.

    const PostFormData=(event)=>{

        event.preventDefault();

    };

    return (

        <div>

            <form onSubmit={PostFormData}>

                <input/>

                <button type="submit">Submit</button>

            </form>

        </div>

    );

React Hook–

React hook is just a collection of some methods. Here are the hooks below:

useRef(): Can **direct access** **DOM** element. In js, we used document.getElementById or ClassName. But use useRef() made that easy to use. And when we change a value using this method, react does **not Rerender or load again**.

Changing HTML –

const Hook\_useRef = () => {

    const Header = useRef()

    const action = ()=>{

        Header.current.innerHTML = "<ul><li>AMM</li><li>JAM</li></ul>"

    }

    return (

        <div>

            <h1 ref={Header}></h1>

            <button onClick={action}>Hook useRef</button>

        </div>

    );

Change attribute value–

    const hook1 = useRef()

    const action = ()=>{

        hook1.current.src = 'https://i.postimg.cc/KjK1wL3c/bulb-off.png'

        hook1.current.setAttribute('height','200px')

        hook1.current.setAttribute('width','200px')

    }

    return (

        <div>

            <img ref={hook1} src='https://i.postimg.cc/6QyTynzr/bulb-on.png'></img>

            <button onClick={action}>Lights</button>

        </div>

    );

Change attribute value(input) –

If :

            <input ref={fname} placeholder='FirstName'/>

            <input ref={lname} placeholder='LastName'/>

Then:

    const fname = useRef()

    const lname = useRef()

but this increases complexity. So,

            <input ref={(a)=>{fname=a}} placeholder='FirstName'/>

            <input ref={(a)=>{lname=a}} placeholder='LastName'/>

    let fname,lname = useRef()

Change CSS -

Install bootstrap: npm i [bootstrap@5.3.2](mailto:bootstrap@5.3.2)

Remove a bootstrap element(text-success) and add element(text-danger)

const Hook\_useRef\_css = () => {

    let cssHeader={useRef}

    const change = () =>{

        cssHeader.classList.remove('text-sucess')

        cssHeader.classList.add('text-danger')

    }

    return (

        <div>

            <h2 ref={(a)=>cssHeader=a} className='text-success'>This is a css change for useRef</h2>

            <button onClick={change}> Change CSS</button>

        </div>

    );

};

Persisted mutable value –

This **renders a value without reloading**. Useref(0) means initial value is 0. We may need to change a value on server side that user don’t need to know. In that case,

const Hook\_useRef\_Mutable = () => {

    let num = useRef(0)

    let action = ()=>{

        num.current++;

        console.log(num.current)

    }

    return (

        <div>

            <button onClick={action}>Mutable useRef Hook</button>

        </div>

    );

};

Caching expensive computations –

We can cache or store the results so that it doesn’t call everytime the components renders. This will be done using useRef.

So, here we will do fetch data, store data and show data

const Hook\_useRef\_cache = () => {

    const store = useRef(null)

    const myTag = useRef()

    const getData = async () => {

        const response = await fetch('https://dummyjson.com/products/1')

        store.current = await response.json()

    }

    const showData = () =>{

        myTag.current.innerText = JSON.stringify(store.current)

    }

    return (

        <div>

            <h2>Api call and cache with useRef: </h2>

            <p ref={myTag}></p>

            <button onClick={getData}>Get data </button>

            <button onClick={showData}>Show data</button>

        </div>

    );

};

useState(): we use this for state management. State is an object that holds the data of a component. **When data changes, component refresh automatically**. This is the biggest advantage.

In “const [number, setNumber] = useState(0)” – number is the data of the state. And setNumber is the function used to modify the data.

Example1 (changing number):

const Hook\_useState = () => {

    const [num, setNum] = useState(0);

    const change = ()=>{

        setNum(num+1);

    }

    return (

        <div>

            <h2>useState : {num}</h2>

            <button onClick={change}>Increase</button>

        </div>

    );

};

Example2 (using Obj):

const Hook\_useState\_obj = () => {

    const [myObj, setObj] = useState({

        key1: "value1",

        key2: "value2",

        key3: "value3"

    })

    const change = ()=>{

        setObj(

            prevObj=>({

                ...prevObj,

                key1: "new value1",

                key2: "new value2"

            })

        )

    }

    return (

        <div>

            <h2>{myObj.key1}</h2>

            <h2>{myObj.key2}</h2>

            <h2>{myObj.key3}</h2>

            <button onClick={change}>Change</button>

        </div>

    );

};