**Introduction**

React is js a library for building user interfaces. React runs on a client side as a SPA (single page application).

JSX stands for JavaScript XML. JSX allows us to put HTML into JavaScript. It is simply a syntax extension of React. It allows us to directly write HTML in React.

Npm is a tool that use to install packages. Npx is a tool that is used to execute packages. Packages used by npm are installed globally you have to care about pollution for the long term. Packages used by npx are not to be installed globally so you have to be carefree for the pollution for the long term.

function App() {

  return (

    <div className="App">

      <Header title="TItle" hey="heyyy"/>

     <h1>Hello From React</h1>

    </div>

  );

}

We put everything in the App function. Header is a component, and Title and hey are props🡪

import React from 'react'

import PropTypes from 'prop-types'

const Header = ({title,hey}) => {

  return (

    <header>

        <h1>Header {title} {hey}</h1>

    </header>

  )

}

Header.propTypes={

    title: PropTypes.string,

}

export default Header

We always need to import from react library because expression after return eventually gets compiled into React.createElement(). That’s why we always need to import the react object on the top even though we don’t use it directly.

React.createElement(“h1”)

We can use “props” keyword to access properties but here I use destructuring to access each prop.

PropTypes helps us to make our code more robost and catch errors. So we need to create Header.propTypes object and set props to their types.

We can also use isRequired property 🡪

Header.propTypes={

    title: PropTypes.string.isRequired,

}

If we use inline styling then we need 2 curly braces 🡺

  <h1 style={{color:"red"}}>Header {title} {hey}</h1>

They are all gonna be camelCased. So not background-color but backgroundColor. We can also create an object for this.

const headerStyle={

    color:"red"

}

And then we can put it wherever we want🡺

<h1 style={headerStyle}>Header {title} {hey}</h1>

We can also create rafce (React arrow function arrow component) and use as a component wherever we want 🡪

import React from 'react'

const Button = () => {

  return <button className='red'>Button</button >;

}

export default Button

we are just gonna need to import it 🡪

import Button from './Button'

const Header = ({title,hey}) => {

  return (

    <header>

        <h1 style={headerStyle}>Header {title} {hey}</h1>

        <Button />

    </header>

  )

}

We can also pass parameters to the Button component 🡺

const Header = ({title,hey}) => {

  return (

    <header>

        <h1 style={headerStyle}>Header {title} {hey}</h1>

        <Button color="red" text="Click Me"/>

    </header>

  )

}

Button component🡺

import React from 'react'

const Button = ({color,text}) => {

  return <button className={color}>{text}</button >;

}

export default Button

As you can see we access parameters by destructuring the props object.

We can also add default props in case we don’t receive any props 🡺

Button.defaultProps={

  color:"red",

  text:"Default Text"

}

Virtual DOM: React uses Virtual DOM which is like a lightweight copy of the actual DOM(a virtual representation of the DOM). So for every object that exists in the original DOM, there is an object for that in React Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document. Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen. So each time there is a change in the state of our application, virtual DOM gets updated first instead of the real DOM. You may still wonder, “Aren’t we doing the same thing again and doubling our work? How can this be faster?”

When anything new is added to the application, a virtual DOM is created and it is represented as a tree. Each element in the application is a node in this tree. So, whenever there is a change in state of any element, a new Virtual DOM tree is created. This new Virtual DOM tree is then compared with the previous Virtual DOM tree and make a note of the changes. After this, it finds the best possible ways to make these changes to the real DOM. Now only the updated elements will get rendered on the page again.

**State**

State is a special object which includes any data that a component needs.

A list in a state must have a key so that react can track them quickly.

In react w do not modify the state directly. We need to use setState() method.

import React, { Component } from 'react'

export default class Info extends Component {

    state={

        count:0,

    };

     ChangeState() {

        this.setState({count:this.state.count++});

    }

  render() {

    return (

      <div>Info</div>

    )

  }

}

When we use a constructor in a class component we always need to use super() method to initialize parameterless constructor of the base class.

Every react component has a property called props 🡺

this.props

Props are read-only, meaning that they cannot be changed.

**Remove an element from state**

Users🡺

import React, { Component } from 'react'

import User from "./User"

export default class Users extends Component {

  constructor(){

    super();

    this.state={

      users:[

        {id:1,FullName:"Rashidli Vahid",age:20 },

        {id:2,FullName:"Elekberov Sabir",age:18 },

        {id:3,FullName:"Ellezova Roya",age:23 },

        {id:4,FullName:"Ehmedov Samir",age:25 },

        {id:5,FullName:"Pashayev Orxan",age:19 },

      ],

    }

  }

  DeleteHandler=(e)=>{

    e.target.closest("tr").innerHtml="";

    this.setState({users:

this.state.users.filter(u=>u.id!==+e.target.getAttribute('data-id'))});

};

  render() {

    return (

      <>

      <table className='mt-4'>

        <thead>

          <tr>

            <th>Id</th>

            <th>FullName</th>

            <th>Age</th>

          </tr>

        </thead>

        <tbody>

          {

        this.state.users.map((u)=><tr key={u.id}>

          <User deleteHandler={this.DeleteHandler} user={u}/></tr>)

      }

        </tbody>

The key attribute is used by react internally.

User 🡺

import React, { Component } from 'react'

export default class User extends Component {

  render() {

    const {id,FullName,age}=this.props.user;

    return (<>

      <td>{id}</td>

      <td>{FullName}</td>

      <td>{age}</td>

      <td><button className='btn btn-danger del' onClick={this.props.deleteHandler}

       data-id={id}>Delete</button></td>

    </>

    )

  }

}

We use filter method 🡺

 this.setState({users:

this.state.users.filter(u=>u.id!==+e.target.getAttribute('data-id'))});

so we are raising an event from the User componenet and Users componenet handles that event.

**Functional vs Class componenets**

If we have a simple stateless component that just returns a react element we can use just stateless functional componenet.

import React from 'react'

export default function Header(){

    return (

      <div className='p-3 header'><h1 className='text-center'>Welcome to the App</h1></div>

    )

}

<> </> is called a fragment.

If we need to pass props to stateless functional componenets then we will receive them as a parameter.

**Styled componenet package**

In order to write better css in jsx files we can download the following package 🡺



Then we import to our file 🡺

import styled from "styled-components"

and then we can use it in the following way 🡺

const StyledButton=styled.button`

    font-size: 20px;

    padding: 10px;

    width:30%;

    border: none;

    border-radius: 3px;

    cursor:pointer;

    background-color:#a0ff9d;

    `;

Now we can use the xml tags

<StyledButton </StyledButton>

**Hooks**

We can use hooks only in functional componenets. We can’t use them in class componenets because classes already have their own way to do the same things that hooks do. Hooks cannot be inside conditions. They mustn’t be called conditionally meaning that we cannot put them is an if statement. So react hooks cannot be inside loops, they cannot be nested in anything. React hooks must be at the top level. React hooks must be called in the exact same order in every component render.

**useState hook**

useState is a hook that is used to implement states in functional components. We import this hook from react 🡺

import React, {useState} from 'react'

We use destructuring to take useState hook from react.

const [count,setCount]=useState(3);

We always write as shown above. 1 is the initial state.

The first value(count) is the state. The second value(setCount) is the method that is used to update the state.

Every time we call setCount method or in other words when we update a state, then it will rerender our component with the new value for our state.

**Context Api**

We know that we can pass props to unrelated componenet by 2 means: Lifting the state, context api.

Lifting the state means that we will end up having plenty of states up in the App componenet and it will look messy. Also we are gonna have to pass the props from app component to components and we will go deeper this way, esentially we will pass props to componenets that do not need those props. To tackle this context api was introduced.

Context api is a simple technique that we can use to create global variables that can be passed to react componenets.

First we create a contet folder in which we will have all of our contexts 🡺



import {useState, createContext } from "react";

export const FullNameContext=createContext();

function FullNameContextProvider(props){

    const [name,setName]=useState(()=>"Vahid");

    const [surname,setSurname]=useState(()=>"Rashidli");

    function UpdateFullName({name,surname}) {

        setName((prev)=>prev=name)

        setSurname((prev)=>prev=surname)

    }

    return (

        <FullNameContext.Provider

value={{UpdateFullName,name,surname}}>

            {props.children}

        </FullNameContext.Provider>

    );

}

export default FullNameContextProvider;

useState hook is for using states. FullNameFontextProvider is just a function that returns provider. Value in the provider is just what we want to send to this props.children componenets.

“props.children” here we receive props as a parameter and by props.children we are telling that include all of its children. So everything that is wrapped inside FullNameContextProvider will be able to use this context. So in the index.js file we can wrap everything inside FullNameContextPovider

import FullNameContextProvider from "./Context/FullNameContext"

ReactDOM.render(

  <React.StrictMode>

    <FullNameContextProvider>

    <App />

    </FullNameContextProvider>

  </React.StrictMode>,

  document.getElementById('root')

);

And then wherever we want to use this context we will import useContext hook and FullNameContext🡺

import React,{useContext} from 'react'

import InfoComponent from './InfoComponent'

import { FullNameContext } from '../Context/FullNameContext'

export default function Form() {

    const {UpdateFullName}=useContext(FullNameContext);

    function SendToUpdateFullName(e) {

        e.preventDefault();

        const form=e.target.closest("form");

        const name=form.elements["name"].value

        const surname=form.elements["surname"].value

        UpdateFullName({name,surname});

        form.reset();

    }

  return (

    <>

    <form className='form'>

        <div className="form-group">

            <label htmlFor="name" className="form-label">Name</label>

            <input type="text" autoComplete='off' name="name"className="form-control" />

        </div>

        <div className="form-group">

            <label htmlFor="surName" className="form-label">Surname</label>

            <input type="text" autoComplete='off' name="surname"className="form-control" />

        </div>

        <div className="form-group">

            <button onClick={SendToUpdateFullName} className='btn btn-primary'>Submit</button>

        </div>

    </form>

    <InfoComponent/>

    </>

  )

}

UpdateFullName method will come from the context and it takes an object inside which we have to specify the name and the surname.

We can take this name and surname and display them🡺

import React,{useContext} from 'react'

import {FullNameContext} from '../Context/FullNameContext'

export default function InfoComponent() {

    const {name,surname}=useContext(FullNameContext)

  return (

    <h1>Hi {name} {surname}</h1>

  )

}