**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. React projects like the ones we create via "npx create-react-app appName" support JSX syntax. It gets compiled to standard JS code behind the scenes. It is a special, non-standard syntax which is enabled in React projects.

In react we use declarative paradigm and for events we use OnEvent: OnClick, ..etc. The naming convention for events is the Handler suffix so that we can differentiate it from our own functions.

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.

React component is a js function that returns html. React component is a JS function that typically returns some HTML (or, to be precise: JSX) code which will be shown on the screen when that component is used.

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()**

What actually functions return in react is React.createElement and the React default function is imported from react library. So under the hood react library is used everywhere. React.createElement() takes 3 arguments. The first argument is the type of the tag. The second argument is an object that configures the element (more secifically an object which sets all the attributes of the element). The rest of arguments are the content. For example, if we have a following tag 🡺

<div>

    <h1>Hey</h1>

    <h2 items={expenses}>Bro</h2>

</div>

Then it will be converted into the following expression 🡺

React.createElement("div",{}, React.createElement("h1",{},"Hey"),React.createElement("h2",{items:expenses},"Bro"))

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**

React always runs only once. So if we want to change something on the page then we must use states, we are actually going to change the states to see the changes on the page. No matter what we want to change let it be text or whatever we can change with the id of an element we must use states because if we don’t we won’t see those changes at all!

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. It's required for React to correctly identify and update (if needed) the list elements.

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 component 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 Button = (props) => {

  const StyledButton = Styled.button`

    font: inherit;

    padding: 0.5rem 1.5rem;

    border: 1px solid #8b005d;

    color: white;

    background: #8b005d;

    box-shadow: 0 0 4px rgba(0, 0, 0, 0.26);

    cursor: pointer;

@media(min-width:768px){...}

  &:focus {

    outline: none;

  }

  &:hover,

  &:active {

    background: #ac0e77;

    border-color: #ac0e77;

    box-shadow: 0 0 8px rgba(0, 0, 0, 0.26);

  }

  `;

  return (

    <StyledButton type={props.type} className="button" onClick={props.onClick}>

      {props.children}

    </StyledButton>

  );

};

We can also add pseudo classes to the tags we just need “&” and that’s it. Now we can use the xml tags

<StyledButton </StyledButton>

It also gives us uniqueness for classes as it hashes the class names so that the styles are different and only applied to this specific button.

& refers to the styled element which is button in this case. So if we want to target a nested element that is inside this button we would have to simply use & label{ …} for example

If we have css for a specific component then we need to import that to our component 🡺



import "./Form.css";

Now the css is also going to be included.

**Css Modules**

CSS Modules is not an official spec or an implementation in the browser but rather a process in a build step (with the help of Webpack or Browserify) that changes class names and selectors to be scoped. So it bacially a way for us to scope styles so that they are applied to specific elements.

There is also another way to add styling, namely by Css modules. So we add a css file with the name Button.module.css for example🡺



Then when we import the element.module.css file we need to specify the name 🡺

import styles from "./Button.module.css";

and eventually we will add objects to classNames 🡺

 <button type={props.type} className={styles.button} onClick={props.onClick}>

      {props.children}

    </button>

Under the hood styles is going to get converted into an object and it will have all the selectors as properties. So all of the followings will be applied to this button🡺

.button {

  font: inherit;

  padding: 0.5rem 1.5rem;

  border: 1px solid #8b005d;

  color: white;

  background: #8b005d;

  box-shadow: 0 0 4px rgba(0, 0, 0, 0.26);

  cursor: pointer;

}

.button:focus {

  outline: none;

}

If we have selector names with hypens then we can use strings for accessing the properties of “styles” object🡺

className={styles["form-control"]}

The class name of the button will look like this 🡺

Button\_button\_\_2j07w

ComponenetName.TagName\_\_hash

**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'

this hook takes an initial state and return not only that value but also a function which is used to update the state🡺

useState(initialState) 🡺 it returns an array in which the first element is this initialState and the second value is the setValueName function. So we use array destructuring🡺

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

Also we use const for values because once we update the value we are going to get a brand new value since the entire component function will get rerendered and the old value will be updated by the new one.

Then when we update we just use setName function 🡺

setName("Updated")

The reason why we use this method to update is because not only we update the value by this method but also react will rerender the component that has this state. So we want to call the component function again when our state changes. By calling setName funtion that is happening.

Standard JS variables don’t cause React components to be revaluated. React doesn't care whether you changed some variable values. It'll not re-evaluate the component function. It only does that for changes to registered state values (created via useState).

It is important to note that setName function doesn’t update the value right away but it schedules it. So inside that function if we console that value after using setName we will see the old value not the updated one. So it is because it doesn’t update that value immediately. It just schedules it for later.

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.

All these hooks must only be called inside of react component functions. They can’t be called outside of those functions also they shouldn’t be called in nested functions. They gotta be called at the top level.

It is okay to have multiple states (multiple pieces of states) in one component. However, there is an alternative to having multiple states that absolutely depends on us as to if we wanna use multiple states or just one state.

If we opt for having all of the states on 1 object then we can do the following 🡺

const [userInput,setUserInput]=useState({

  enteredTitle:"",

  enteredDate:"",

  enteredAmount:"",

});

And if we want to update one of the states 🡺

setUserInput({

  ...userInput,

  enteredTitle:e.target.value})// this will override the enteredTitle that was copied by ...userInput

The difference is that if we have one object holding all of our states then when we want to update one state we end up updating all of our states.

If we depend on the previous state, then It is not good to update as shown above. As we know react schedules the setState function and it will run it later. Therefore, if we schedule a lot of state updates at the same time we can be depending on an outdated or an incorrect snapshot of the state if we use the approach schown above. So we explicitly need to have access to the previous state.

When we use setState function, this function receives a property by default by react which is the previous state🡺

setUserInput((prev) => {

  return { ...prev, enteredTitle: e.target.value };

});

This is the correct way of updating because here we know exactly that the property that we receive from react is actually the previous state. React will guarantee that the snapshot of the state it gives you in the inner function will always be the latest state’s snapshot.

**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>

  )

}

**Props**

No matter how many properties we are passing, we are always gonna get one object that holds all the properties 🡺

function App() {

  return (

    <div className="App">

      <MainWrapper />

      <Dummy title="Title" Desc="desc" />

    </div>

  );

}

Then in the functional component we will receive it as a parameter as one object 🡺

function Dummy(props) {

  return (

    <>

      <h1>{props.title}</h1>

    </>

  );

}

export default Dummy;

**Props.children (Compositions)**

Normally, we can’t wrap elements inside a custom componenet. We need to specify props.children 🡺

function App() {

  return (

    <Dummy>

      <div className="App">

        <MainWrapper />

      </div>

    </Dummy>

  );

}

So in order to wrap everything inside Dummy component we need to specify all of the children of Dummy component with props.children

function Dummy(props) {

  return <h1>{props.children}</h1>;

}

export default Dummy;

“children” is a reserved name. The value of children is always the content between the opening and closing tags of our custom component (Dummy).

So the children of Dummy component will be inside h1 tags. Whenever we combine components we are using composition.

**Comunicating from bottom up (child to parent)**

Let’s say in NewExpense component we have a method called saveExpenseDataHandler function and we want to pass it to its child component which is ExpenseForm in our case🡺

const saveExpenseDataHandler=(expenseData)=>{}

This is the method and we pass it to the ExpenseForm component

<ExpenseForm onSaveExpenseData={saveExpenseDataHandler}/>

Then we receive this method inside props in the ExpenseForm component.🡺

props.onSaveExpenseData(expenseData)

Keep in mind that the name of the method is the attribute name designated for the method in the NewExpense component.

**Controlled & Uncontrolled components**

If we have a parent component that has a method for changing the state of a child component then this child component is also called a controlled component.

The components that do not have any states are called presentational/dump/stateless components. Because they don’t have any internal state they are just for displaying some data.

**Two-Way Binding**

Two-way binding is basically when we change the value of the state and put that value directly into the input as a value 🡺

const [date,setDate]=useState("2022-02-20")

Now we just put the date into the input as a value 🡺

<input type="date" value={date}/>

Now this is called Two-way binding.

**Ternary expressions**

In react, inside the curly braces {} for loops or if statements are not allowed. Insead, we write ternary expressions 🡺

 {array.length>0?array:<p>Empty String</p>}