**501 – Advance Web**

**Unit 2 : Fundamentals of React.js**

**2.1 Overview of React :**

* ReactJS is an open-source (Free for all to use) framework or library of javascript for making front-end user interfaces for the web application.
* ReactJs is developed and maintained by Facebook(Jordan Walke).
* React was developed in 2011 and released in may 2013.
* Reactjs application is made up of multiple components where each code adds into outputting a small and reuseable HTML code.
* ReactJs uses a virtual DOM mechanism to fill data in html DOM. The virtual DOM is faster because it only changes individual DOM elements instead of Entire DOM everytime.(similar to ajax that only refreshes certain section of page instead of entire page).
* Latest version of react available is 18.0.0
* Latest version of create-react-app is 5.0.1
* To Install ReactJS we need to have nodeJS and NPM pre-existing on device. Then using following code in terminal(cmd) react app can be installed:

**npx create-react-app appname**

* To run the react application we use terminal(cmd). First we navigate the app folder using cd command. If inside current directory use : **cd appname**
* Then to start the app(after getting into relevant directory) we use : **npm start** .
* React was introduced to overcome repetitive creation of DOM whenever page was loaded or reloaded which wastes a lot of memory and reduced performance of application.
* React uses the same traditional flow of data but not directly operating on the browsers DOM immediately, instead operating over a virtual DOM that only apply necessary changes to original DOM after processing entirely in memory itself.

**2.1.1 Concepts of React:**

**1. JSX :**

JSX is acronym for JavaScript XML that is and javascript syntax extension that provides facility to write HTML texts into React. It converts Html tags into react elements. It extends ES6 to make HTML text co-exist with react code.

We can write Expressions like 5+5, A-B etc using JSX inside { } curly braces. Making it possible to write:

var x = <h1>Square Root of 3 is {3\*3}</h1>;

Which would display “Square root of 3 is 9” by evaluating the expression.

We can also insert a large block of html on multiple lines by using () parentheses. For example:

var x = (

<div>

<p>this is first image</p>

<img src=”img1.jpg” height=”100” width=”150” />

</div>

);

Although since the JSX follows XML, so the HTML elements are mandatory to have a parent element that contains all others in the code and also elements must be properly closed. Error would be thrown in else cases.

To use class attribute for html tags we need to write the className instead, since in javascript the class is a reserved word.

Eg : var x = <h1 className=”redH1”>RED</h1>;

Conditional statements aren’t allowed inside JSX code as well, hence they could be evaluated outside itself or ternary operator to be used instead to suffice.

Eg:

var x = 5;

var str = <h1>{(x)>2? ”yes”:”no”}</h1>;

**2.Virtual DOM:**

Virtual DOM is an “in-memory” representation of the actual DOM. It is synced with the actual DOM. It’s similar to One-Way Data Binding. Whenever we make changes into the web app, the UI is re-rendered into the virtual DOM and mapping of differences between previous and new DOM are done. After that, the real DOM only gets applied the changes instead of recreating entire DOM again preventing wastage of memory and making application faster. This is also called reconciliation.

**3.One Way Data Binding:**

One way data binding is the concept of unidirectional dataflow. React JS follows One Way data binding, where the data/information in form of props from the parent element to child element. The data doesn’t flows vice versa. Therefore making the React application more flexible and increase efficiency. Flux pattern is used to keep it unidirectional. One way data binding gives us more control over the app.

**4.Components:**

Components are the hearts of the reactJS. React app itself is made up from multiple components each having its logics and controls. A component is a small reusable unit of code that return simple html. They take props as input and return elements for UI. With help of this we can split a larger projects into smaller discrete sections of code and work on them individually.

**5.State:**

The state is the heart of the component. It defines how the component behaves or is rendered. They make the Component Dynamic and Interactive. They are updatable over time structures that contain information about the component. State can only be accessed or used inside the component only or only by the component itself. They must be kept as simple as possible.

**6.Props:**

Props are short for Properties. They refer to immutable read-only components. Props are a way to send data from one component to another (from parent to child in react due to One Way data binding). They are objects to store value of attributes of html tags.

**2.1.3 React Interactive components: Components within components and Files:**

Components are small reusable piece of codes that are like functions that take in Props as input and return HTML. The help to break down large Codes into smaller discrete parts to work on each section individually.

Components are of two types:

* Class Components
* Function Components

1. Class Components:

These are components in form of a class that Inherit the React.Component class to use the methods from React.Components class.

The class needs to define a render method to make a component that returns html.

Example:

class Student extends React.Component{

render()

{

return <p> Name : XYZ , roll : 31 </p>;

}

}

1. Function Components:

These components are functions that return html and are similar to class components but these unlikely can be written in much less code and easy to understand.

Example:

Function Student()

{

return <p> Name : XYZ , roll : 31 </p>;

}

* **Components in Components :**

One Component can be easily referred inside another component in react js app.

To refer to the component we need to write <Component\_name /> inside another component.

Example:

function Student()

{

return <h5> I am Student of this classroom </h5>;

}

function Classroom()

{

return ( <div>

<h1>This is TYBCA Division B</h1>

<Student />

</div> );

}

var root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<Classroom />);

Gives Output:

This is TYBCA Division B

I am Student of this classroom

* **Components in Files:**

Since React components are very much reusable, you can create then in separate files to be used in multiple apps, or use them from an existing code file created earlier. To make a component re-usable by other codes you need to make the file exportable using export statement.

Example:

Function Add()

{

return <h3> { 5+ 6 } </h3>

}

export default Add;

Its recommended to keep Component and file name with first letter capitalized.

Save the component file with .js extension to make it usable in other React scripts.

To use use external file components in current app file we need to use the import statement to import the required component.

Eg .

import Add from ‘./Add.js’;

import React from ‘react’;

import ReactDOM from ‘react-dom/client’;

var root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<Add />);

gives output of 5+6 i.e 11 on browser.

**2.1.3 Passing data through Props:**

Props are short for Properties, these are the arguments passed into react components. Props are similar to html attributes and passed to component similarly.

Note: Props are Read-Only and hence their values cannot be manipulated and only read.

Example:

<Fruitbasket fruit=”Oranges” />

The component receives these argument as props object.

These props object can be used in component with prop.attribute\_name to extract/use the value of attribute sent.

Example:

function Fruitbasket(props)

{

return <h3> There are 7 {props.fruit} in the basket </h3>

}

Entire Example:

function Fruitbasket(props)

{

return <h3> There are 7 {props.fruit} in the basket </h3>

}

var root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<Fruitbasket fruit=”oranges” />);

this shows “there are 7 oranges in the basket” as output.

Props can also be used to send data from one component to another.

Props can also be used to send variables using { var\_name } as value of attribute in tag.

Example:

function Fruits(props) {

return <h3> {props.fruit} are king of all fruits </h3>; }

function FruitBasket(){

var frt = “mangoes”

return ( <>

<h1> there are five {fruit} in the basket </h1>

<Fruits fruit={frt} />

</> ); }

var root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<Fruitbasket />);

Props can also be used to send objects using variables by using {var\_name} as value of attribute in tag. Each key’s value can be accessed in other component or in general elsewhere using props.attribute\_name.key

Example:

function FruitBasket(props){

return ( <>

<h1> there are five {props.fruit.name} in the basket </h1>

<h1> they cost {props.fruit.price} each </h1>

</> ); }

var frt = { name : ”PineApples” , price : 50 }

var root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<Fruitbasket fruit={frt} />);

**2.2 Class components:**

Class Components:

Class Components of react are those classes that inherit the React.Component class in order to use the methods of it. They need to override the render method to make a component that returns html.

Example:

class Greet extends React.Component{

render()

{

return <p> Bonjour! </p>;

}

}

**2.2.1 React class and class components:**

Classes in react js were introduced by ES6. Class can be said as a container of properties and methods. It’s initialized using class keyword, and properties of it can be assigned using a constructor method.

Example:

class Book

{

constructor( name,auth ){

this.Title = name;

this.author = auth;

}

}

An object can also be created for the class by following syntax:

var var\_name = new Class\_name(args);

Example:

var book1 = new Book(“The Chronicles Of Narnia:Prince Caspian”,”C.S Lewis”);

You can also create your own methods in the class as well to use the properties of that class. Parameters can also be passed to those methods to be used in the function.

Syntax: method(parms){ …

return value;

}

Example:

class Book{

constructor(name,auth)

{

this.title = name;

this.author=auth;

}

display(price){

return “Title:” + this.title +” author: “+this.author+” price: ”+price;

}

}

var book1 = new Book(“the BFG” ,”Roald Dahl”);

book1.display(340);

To Inherit properties and methods of one class into another, extends keyword is used. This can be helpful to use methods of one class into another without hassle of writing the code again and again. Super() method is used to call the constructor of the parent class into child class to initialize properties of parent class.

Classes can also be handy in creating Class components for react js. Every Class components in react are the child classes of Component class since they need to inherit the React.component using extends keyword. React.component class provides several methods that can be used to make a class component. One of it being render() method that’s most important method of a class component.The upper hand of class components over function components is that They have a built-in state object and have access to lifecycle of the components.

**State :** state is an object where we can store the properties and values of the components in form of key value pairs as an object. When state object changes, the components are re-rendered.

State object can be created using a constructor function of that particular component class.

Syntax: this.state = { key1 : val1 , key2 : val2 , … } 🡨 this is written inside constructor

The values of state object of the class can be accessed inside methods using:

this.state.property\_name

Example: this.state.name , this.state.age etc.

And lastly to change the values of the state object we can use the setState() method.

When we want to update value of any key(property) in the state object this method is used. Its necessary to use this method to change the properties in the state object because it ensures to call render method , and all other component lifecycle methods and lets component know its been updated.

Syntax: this.setState({key:new\_value});

Example: this.setState(“age”:20);

Component LifeCycle:

Another major feature of class components is that they get access to the component lifecycle. Each component in react has a life cycle that can be monitored or manipulated during its phases.

There are 3 main phases of a lifecycle:

1)Mounting :

This is the phase where elements are put/created into DOM. This is the creation phase of the component where there are four important methods:

1. Constructor() : this method is used to initiate the state object or properties of the class component.it is an optional method, that when used should have one argument props that shall be used to construct parent class React.component using super(props).

Example: constructor(props){

super(props);

this.state = { Name : “XYZ”};

}

1. getDerivedStateFromProps():this method is used to set the values of state object according to the initial props.It takes state as the argument and returns state object with changes.

To explain this , suppose you initiate a Name property in state object but want to update it to value in the props then this is used.its optional and called just before render().

Example: getDerivedStateFromProps(props,state){

return {Name : props.name };

}

1. render(): this is the most important and compulsory method of lifecycle. Every class component mandatory needs to override this method.It is used to render the html to the DOM and give the output.

Example: render(){

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

}

1. componentDidMount(): This method Is used to run statements that are needed after the component is rendered. This method is optional as well. Example if we need to change color of some text after page is rendered it can be used. All AJAX calls need to be written in this method.

Example: componentDidMount(){

this.setState({ color:’blue’});

}

2)Updating:

This is the second phase of the lifecycle and refers to phase where component gets updated or changes are made to the component.

It has five important methods:

1. getDerivedStateFromProps(): same as mounting phase , this method is used to change the value of properties in state object to those in the props object.
2. shouldComponentUpdate(): can be used to set a boolean return value that specifies if or not the react should continue rendering. Default is true. It can be used to prevent component from rendering updates by returning false.

Example: shouldComponentUpdate(){

return true/false;

}

1. render(): ofcourse needs to be used in updating phase as well to re-render any new updates.
2. getSnapshotBeforeUpdate(): This method is used to get access to the values of State and props before they were updated.it is necessary to use componentDidUpdate() method with this method else leads to an error. Example if image gets updated, but we want to know which image it was before update after it gets updated we can use this method.

Example: getSnapshotBeforeUpdate(prevProps,prevState){

document.getElementById(“div1”).innerHTML = “Prev image : “+prevState.image;}

1. componentDidUpdate(): this method is used to run statements after the component is updated and re-rendered onto DOM. This method gets triggered after updates are made. It can be used for example to inform user what changes were made.

Example : componentDidUpdate(){

document.getElementById(“div2”).innerHTML = “ Updated color to blue “; }

3)Unmounting:

This is the phase that represents the removal of the component from the DOM. When react component gets removed unmounting phase happens and unmounting methods can be used. There is only one Built-in Unmounting method that gets called when component is unmounted.

1)componentWillUnmount():

This method is triggered when the components is about to be removed from DOM. It can be used to run some statements before some component gets unmounted finally. Like if we want to display some sort of message before some control/text disappears from screen it can be used.

Example: componentWillUnmount(){

alert(“the so and so component will be now unmouned”);}

**2.2.2 Conditional statements, Operators, Lists:**

**Conditional statements :**

React js can also provide conditional rendering, which means if condition is satisfied, then the rendering takes place. For conditional rendering, we have three ways:

1. If statement: if statement can be used to check against a condition and if it tends true , then the following block of statements gets executed else skipped over. An else or elseif can be used as well , along with if to make a choice of statements to be executed if first one tends false.

Eg. function Result(props){

if(props.marks >= 33){ return <Passed />;}

else{

return <failed/>}

}

1. && operator: && can as well be used to provide conditional rendering, if the condition before the && tends true then statement right after it is executed. If its false the statements get skipped over.

Eg: function Result(props){

return(

<h1> Rollno {props.roll}</h1>

{

props.marks > 33 && <h2> you have passed </h2>

}

);

}

1. Conditional/ternary operator: the ternary operator also known as the conditional operator can be also used for inline conditional rendering. A condition is checked against and if its true one statement is executed, else another.

Syntax: condition ? true : false.

Example : function Result(props){

return( <h1>{props.marks>33?’pass’:’fail’}</h1> ); }

**Operators :**

React may supports all the javascript operators such as:

>Arithmetic

>Logical

>Relational

>Bitwise

>Assignment

>String

>Type

, but particularly React Js has its own two Operators:

1. Spread Operator:

Spread operator (…) of react helps to quickly copy all or part of an existing array or object into another array or object. It is often used in combination with destructing.

Example : >With array :

var n1 = [1,2,3];

var n2 = [4,5,6];

var combined = […n1,…n2]; Output : [1,2,3,4,5,6]

>With object :

var ob1 = { name : ‘abc’ , age : 34 };

var obj2 = { job : ‘programmer’ , age : 24} ;

var newobj = {…obj1,…obj2};

Output : { name:’abc’ , age :24 , job:’programmer’}

\*Note\* here age gets overwritten to 24 in the new object since it considers it as an updated value if field names are same.

1. Ternary Operator:

Ternary operator also called as conditional operator is a condition evaluating operator similar to if/else statement that performs one statement if true or another if false. It can be a really handy way to suffice needs of the if/else statement when return values are not a block of code.

Syntax : Condition ? <statement if true> : <statement if false>;

Example : age > 18 ? “eligible to vote” : “not eligible to vote”;

**Lists:**

Lists are similar to arrays that can be used to display data in an ordered format that can be used to display menus on websites. In react the map() function is used to traverse through each item in the array returning a new list as result. Its an efficient way to generate lists.

Syntax of map() : Array.map(callback);

Example: var animals = [“sheep”,”cows”,”dogs”,”cats”];

var domestic = animals.map((animal) => { return <li> {animal} </li>});

The domestic will be a list containing all the values of array animals in terms of html to be rendered over a webpage.

Keys can be used to keep track of the elements of array, in-case there is an update or delete only that particular item will be re-rendered instead of entire list. Key is an attribute that helps identify which elements have been updated or deleted. It can be a unique id that can uniquely identify each element in array.

Example:

function Animals(){

var animals = [{id:1 , animal:”lion”},{ id:1 , animal:”fox”},{ id:1 , animal:”bear”}];

retrun(<ul> { animals.map(aml=>{return(<li key={aml.id}>{aml.animal}</li>)})}<ul>);

}

**2.2.3 React Events: Adding events, Passing arguments, Event objects:**

Events are change in state of an object such as click, change, checked etc. Just like HTML DOM, react can also perform actions in response to user events.

**Adding Events:** Events can be added into react within the tags. They must be written in camelCase syntax and their handlers are written in curly braces.

Syntax: <tag evEnt={}>---text----</tag>

Example: <button onClick={myfunction}> Click me </button>

Here this button on click triggers the onClick event of react and calls the myfunction component to render its code.

**Passing arguments:** arguments can also be passed into the events with the use of an arrow function.

An arrow function is a shorter way to write a function. It uses ’=>’ to define function.

Syntax : func\_name = () => { function definition };

Or if there is only return value it can be directly written without any {} or return keyword like :

Func\_name = () => “returned statement”;

Even arguments can be passed by adding argument variable in the () like:

Func\_name = ( x ) => { function definition };

To pass an argument to event using the above arrow function like :

Myfunc = (strg) => { alert(strg); }

<button onClick={() => myfunc(“hello”)}>click me</button>

Here on button click “hello” is passed to the myfunc component inside strg variable and can be used there to alert “hello”.

**Event Objects**: the events can also be used as an object that can be passed to a component function. It represents event keyword.

Example:<button onClick={(event) => myfunc(“hello”,event)}> click here </button>

Here event ( in this case onclick ) is passed as well to the function as an object , which can be used to access its properties or methods or values. Such as event.type gives what type of event took place.

**Unit 3 : Forms and Hooks in React.JS**

**3.1 Forms: (Adding forms, Handling forms, Submitting forms) :**

Forms are crucial part of any website that enables user to interact with the webpage and gather information from user. It can be useful to authenticate users, add users, take input from user, gather information about user or their requirements, feedbacks etc. Form has several components like textbox, checkboxes, buttons etc. Similar to the html even react enables form facility for the webpage.

**Adding Forms:**

In react forms can be simply added using the JSX syntax system. Simply adding form to jsx, although would work normally, submit as well refresh the page but not as it’s meant to work in react.

Example:

function FormExamp(){

return(

<form>

Input name : <input type=”text” />

</form>

);

}

Handling Forms:

In order to overcome just submission and refresh of page and form, we need to handle the data when value is changed or form gets submitted. Unlike html where form data is handled by DOM, here in react we need to handle it using functional components where a function stores data into its state and then handles it. Changes can be controlled using onChange event handler , and useState hook can be used to keep track of each input value. This would make our component look something like this:

function formHand(){

var [name,setName] = useState(“”);

return(<form>

Name:<input type=”text” value={name} onChange={(e)=>setName(e.target.value)} />

</form>);}

var root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<formhand/>

Submitting Forms:

When the forms are submitted, the data in the forms need to be handled or utilized by some client or server script. To control what happens to the data after submit button being clicked , react js has onSubmit attribute that goes with the form tag and lets us provide an event handler function component to handle the data after its submitted.

Example:

function FormExamp(){

var [name,setName] = useState("");

var handler=()=>{

alert(`name is :${name}`);

}

return(

<form onSubmit={handler}>

Input name : <input type="text" value={name} onChange={(e) => setName(e.target.value)} />

<input type="submit" />

</form>

);

}

var root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<FormExamp />);

**3.1.1 event.target.name and event. Target.event, React Memo:**

* **event.target.name:**

event.target.name contains the value of name attribute of the event target. To simply explain , lets talk of a field say textbox , it has an attribute name that contains the name of the control. By any chance if we want to access the value of name attribute of the control inside a handler function component we access it using event.target.name , that defines the name of the target control that triggered the event.

* **event.target.value:**

event.target.value contains the value of the value attribute of the event target.it allows us access to the value input in the input field of the target control that triggered the event.

Multiple fields can be handled using this by using name and value attribute of input fields as key value pairs to create an object {} of them. event.target.name and event.target.value gives the access to fields in the event handler component.

Example:

function FormExamp(){

var [obj,setVals] = useState({});

var handler=(event)=>{

setVals(values => ({…values, [event.target.name]:event.target.value}))}

var smbt=()=>{

alert(obj);}

return(

<form onSubmit={smbt}>

Input name : <input type="text" value={obj.name} onChange={handler} />

Input gender : <input type="text" value={obj.gender} onChange={handler} />

<input type="submit" />

</form>);}

var root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<FormExamp />);

* **event:**

event is an object that is used to refer to the event that was triggered due to action by the user. Like event object of onSubmit refers to submit event or onChange refers to the Change event in input field. This object is useful to access different properties or methods related to it.

Methods like preventDefault() that cancels the default behaviour of the cancellable form controls, or stopPropogation() that prevents bubbling of events to parent element.

* **event.target:**

event.target is used to access the properties of the target i.e the form-control that triggered the event.It refers to the element that triggered the event. The target property gets the element on which the event originally occurred.

* **React Memo:**

In a script there are chances that a component may get re-rendered again and again unnecessarily. As a developer we need to make our code well optimized and do performance improvements so our code remains blazing fast at all times. In react js one of the way to do so is by wrapping the component with memo. React memo helps us do that, the react memo is a higher order component that wraps itself around a component so that the component doesn’t do unnecessary re-renderings.if only there are changes in the props passed to function then only it will re-render the component.if props remain same, component is kept untouched same as before without re-rendering.

We can either put memo over function bracket like:

var myfunc = React.memo(( {props} ) => {} );

or with export while making external component file:

var myfunc = ({props}) => {

……

Functional component statements

……

};

export default memo(myfunc);

Example : File memodemo.js :

import { memo } from 'react';

var Memodemo = ({roll}) =>

{

console.log("changed roll");

return (<>

<h3> welcome,{roll} </h3>

<h3> it is a memo component that doesnt re-render unless roll is changed</h3>

<h3> to notice re-rendering use inspect-console mode to see how many times change message was displayed</h3>

</>

);

}

export default memo(Memodemo);

file index.js :

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import Memodemo from './memodemo.js'

import { useState } from 'react';

function MyForm() {

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

const [roll, setRoll] = useState("");

return (

<form>

<Memodemo roll={roll}/>

<label>Enter your name:

<input

type="text"

value={name}

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

/>

</label>

<label>Enter your roll:

<input

type="text"

value={roll}

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

/>

</label>

<br/><br/>

name entered was : {name}

</form>

)

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<MyForm />);

here this app only re-renders the memodemo component if there is a change in roll number that is the prop to that memo component otherwise only name component is re-rendered .

**3.1.2 Components (TextArea, Drop down list (SELECT)):**

* **TextArea:**

Text areas in the form are handled differently in react component forms, unlike html. As we know the text area in html places its value between <textarea></textarea> tags. In react the value of text area is places in the value

Attribute of the text area tag. The useState hook is used to manage the value of text area.

Example:

import {useState} from ‘react’;

import ReactDOM from ‘react-dom/client’;

function TextAreaDemo(){

var [TA,setTA] = useState(“text area content will appear here”);

var handler = (event) =>{

setTA(event.target.value); }

return( <form>

<textarea value={TA} onChange={handler} />

</form>);

};

var root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<TextAreaDemo />);

* **Drop Down list(Select) :**

Drop down list as well is handled differently in react js than in basic HTML. In html the selected value was defined using the selected attribute but in react js it is access with value attribute on select tag.

Example :

import {useState} from ‘react’;

import ReactDOM from ‘react-dom/client’;

function SelectDemo(){

var [pokemon,setStarter] = useState(“Your stater pokemon”);

var handler = (event) =>{

setStarter(event.target.value); }

return( <form>

<select value={pokemon} onChange={handler}>

<option value="select">selection</option>

<option value="charmander">charmender</option>

<option value="squirtle">squirtle </option>

<option value="bulbasaur">bulbasaur </option>

</select>

<br/><br/>

Your starter pokemon selected : {pokemon}

</form>);

}

var root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<SelectDemo />);

**3.2 Hooks: Concepts and Advantages:**

Hooks are special features in react js that allow you to use state and other react features without having need of making a class. Basically they are functions that hook-into react state and life cycle features and make them available in functional components without using class. Hooks need to be imported from the ‘react’ package.

Rules of using hooks:

1. Only React function components can call hooks.
2. Only top level of the component can call hooks.
3. Hooks cannot be conditional.
4. Hooks would not work with react class component

Advantages of Hooks:

1. **There is no need to refactor the functional component into class component**: if we want to use the state or lifecycle features, we don’t need to remake a functional component into a class component, all required state and lifecycle features are at your hand using hooks.
2. **Reduces needs of Classes and class components in the code**: as hooks suffice the needs of the class in a code, we don’t have needs to create classes and it reduces lines of codes.
3. **Provides a more concise and elegant way to use the life cycle events like mounting, update or unmounting.**
4. **There is no more need of “this” keyword:** there is no more needs of using “this” keyword to access the state. It makes less confusing for everyone.
5. **There is no need of method binding:** since there is no need of “this” keyword, there is no need of binding methods to “this” to clear out context to the class and it reduces code.
6. **It’s easy to decouple logic from UI:** we can easily split the UI and Logic using the hooks so, both can be independently reusable.
7. **Enables Sharing stateful logic between components:** with class based approach it’s hard to share stateful logics between several components. Hooks can extract these stateful logics and enable to use them separately.

**3.2.1 useState, useEffect, useContext:**

* **useState:**

useState hook is one of the most important hooks in the react environment, that can be used to set or retrieve state. States are the general properties or data that need to be monitored in code.

useState hook can be imported into our component by using:

import { useState } from ‘react’; statement.

It can be used by calling useState function component in our code. It takes initial value as a parameter and returns current state(value) and function that updates that state.

Now we’ll see an example of use state hook:

import { useState } from ‘react’;

import ReactDOM from “react-dom/client”;

function UseStateDemo(){

let [name,setName] = useState(“”);

return(

<form>

<input type=”text” value={name} onChange={(e) => setName(e.target.value)} />

<h1> Name You entered : {name} </h1>

</form>

}

var root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render(<UseStateDemo />);

* **useEffect :**

This Hook allows us to add effects or more precisely ‘side effects’ into our functional component such as ajax calls, fetching datas from server, Updating the DOM directly, timers etc. it has two arguments , one is a call back function that defines what effect this hook does in the application and second is dependencies that’s optional.

To import useEffect Hook in our application we write the following line:

Import { useEffect } from ‘react’;

Example :

import { useEffect } from ‘react’;

function User({name}){

useEffect(() => {

document.title = name;

}, [name]);

return <h1>{name}</h1>;

}

These effects needs to be cleaned up as well in order to save up memory so we need to make a cleanup function for the effect as follows :

import { useState,useEffect } from ‘react’;

function User({name}){

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

useEffect(() => {

var timer = setTimeout(() => {

setCount((count) => count + 1);

},1000);

return () => clearTimeout(timer)},[]);

return <h1>{name}</h1>;

}

Here clearTimeout function clears the timeout of the timer set by the setTimeout function.

* **useContext:**

This Hook is a functional component hook that is used to pass data that is called context into a child component.

A context is a way in react to manage state globally. It can be used with the useState hook to share state between nested components.

The issue that caused creation of useContext hook was that when we wanted to pass data between several child/nested components we had to pass it as props to the further one.

To prevent this we have the useContext hook that allows us to use the created contexts freely in any components.

Importing hook :

import { useState , useContext , createContext } from ‘react’;

Creating a context:

To create a context we need to use the createContext() function. That can be imported with the useContext hook.

Syntax :

var MyContext = createContext();

Context provider :

To create a context provider we need to wrap child component in Context Provider to supply value;

function parent(){

var[name,setName] = useState(“user123”);

return( <MyContext.Provider value={name}>

<h1>Welcome ${user}</h1>

<child1 namer={name} />

</MyContext.Provider>);

}

Accessing Context using useContext hook :

To access the data in the created context using the useContext hook in the child component we want we use the following syntax:

var user = userContext(MyContext);

Entire Example :

import { useState , useContext , createContext } from ‘react’;

function parent(){

var[name,setName] = useState(“user123”);

return( <MyContext.Provider value={name}>

<h1>{`Welcome ${name}`}</h1>

<child1 namer={name} />

</MyContext.Provider>);

}

function child1(){

return( <>

<h1>child1</h1>

<child2 />

</>

);

}

function child2(){

var name = useContext(MyContext);

return(

<>

<h1>child 2</h1>

<h2>{`Welcome to child2 ${user} `}</h2>

</>

);

}

3.2.2 useRef, useReducer, useCallback, useMemo

* **useRef:**

the useRef hook is used to persist i.e retain the values of the state/data between renders. It can be used to store a mutable value that does not require re-rendering when its value is updated. It can be also used to access DOM element directly.

useRef() hook return an object called current. useRef might also be able to keep a track of the statechanges or in easier words keep track of previous state values.

To import useRef hook :

import { useRef } from ‘react’;

To make a reference value of a HTML element:

We can perform DOM manipulations using the useRef by referencing to the HTML element by adding ref attribute to the element as follows:

Syntax : <input type=”text” ref={inputElement} />

Example :

import {useState,useEffect,useRef} from ‘react’;

function UseRefDemo(){

var [curr,setCurr] = useState(“”);

var prev = useRef(“”);

useEffect(()=>{

prev.current = curr;

},[curr]);

return(

<>

<input type=”text” value={curr} onChange = {(e)=>setCurr(e.target.value)}/>

<h2> Currently Rendered Valu : {curr} </h2>

<h2> Value before Re-Rendering: {prev.current}</h2>

</>

);

}

The above example demonstrates use of useRef for keeping track of values before the re-rending.

* **useReducer:**

the useReducer hook is a hook that works similar to useState Hook but allows custom state logic unlike useState hook. In simpler words we can use this to manipulate or set value of state with our custom logic instead.

It takes two arguments first is a reducer function and second is initial state. The reducer function is used to define the custom state logic and the initial state is the initial value of the state. It returns the current state and the dispatch method.

Importing useReducer :

import {useReducer} from ‘react’;

Example:

const num= {count: 0};

function reducer(state, action) {

switch (action.type) {

case 'increment':

return {count: state.count + 1};

case 'decrement':

return {count: state.count - 1};

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer,num);

return (

<>

Count: {state.count}

<button onClick={() => dispatch({type: 'decrement'})}>-</button>

<button onClick={() => dispatch({type: 'increment'})}>+</button>

</>

);

}

* **useCallback**:

the useCallback hook is useful to prevent any component child from rerendering repeatedly without any need. The useCallback hook takes in two values the callback function and the dependencies in form of array and returns a memonized version of callback function that only rerenders if the dependencies values are changed/updated.

Importing useCallback:

import {useCallback} from ‘react’;

Syntax of useCallback:

const memonized = useCallback( () => {

code statements ..

}, [a,b],);

Example:

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

function Button(props) {

return <button onClick={props.onClick}>{props.name}</button>;

}

function App() {

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

const [isActive, setActive] = useState(false);

const handleCount = useCallback(() => {setCount(count + 1);

console.log("re-rendered handleconst");

}, [count]);

const handleShow = useCallback(() => {setActive(!isActive);

console.log("re-rendered handleshow");

}, [isActive]);

return (

<div className="App">

<p>inspect to view console that shows number of time each funcn re-rendered</p>

<p>this shows that particular functions only re-renders if the value is changed</p>

{isActive && (

<div>

<h1>{count}</h1>

<Button onClick={handleCount} name="Increment" />

</div>

)}

<Button

onClick={handleShow}

name={isActive ? "Hide Counter" : "Show Counter"}

/>

</div>

);

}

export default App;

this examples shows demonstration of re-rendering of particular function only if the value of its dependency have changed else doesn’t re-render. Also only the function whose dependency changed re-rendered not all of them.

* **useMemo**:

this hook is a functional component that is used to return a memonized value. A memonized value or function is such that it remembers the computed output in cache and if same values are computed again it doesn’t recomputed everytime, instead returns the value from cache. For example if a memonized function is computing addition of two numbers , 1 and 2 and the output is 3 the value 3 is stored in cache and if we re-run the function again with no change in numbers being added , the function doesn’t re-compute , instead it just returns value 3 from the cache.

useMemo hook is used to gain improvement in performance of our React application.

useMemo hook takes two arguements :

1)function that returns value

2)dependency array

To import useMemo Hook:

import {useMemo} from ‘react’;

Syntax of useMemo Hook:

const memVal = useMemo( function , []);

Example:

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

function App() {

const [number, setNumber] = useState(0)

const squaredNum = useMemo(()=> {

return squareNum(number);

}, [number])

const [counter, setCounter] = useState(0);

const onChangeHandler = (e) => {

setNumber(e.target.value);

}

const counterHander = () => {

setCounter(counter + 1);

}

return (

<div className="App">

<input type="number" placeholder="Enter a number"

value={number} onChange={onChangeHandler}>

</input>

<div>OUTPUT: {squaredNum}</div>

<button onClick= {counterHander}>Counter ++</button>

<div>Counter : {counter}</div>

</div>

);

}

function squareNum(number){

console.log("Squaring will be done!");

return Math.pow(number, 2);

}

export default App;

the above demo only calls the squareNum function if the number gets changed else returns the cache value of previous result. Also while not being affected by the re-rendering of the click count changing function.

**3.2.3 Hook: Building custom hook, advantages and use**

* React apart from providing various built-in hooks also provides us facility to create our own hooks and use it in our applications. Such user-defined hooks are to be called as Custom Hooks.
* Custom Hooks are normal javascript functions that begin with prefix “use” and might use other built-in or custom hooks within them.
* Creating A Custom Hook is just like creating a javascript function whose name starts with “use”. Below is an example.

Creating custom hook Example:

import {useState,useEffect} from ‘react’;

function useMyCustom( init , componentName){

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

function resetCount(){

setCount(count+1);

}

useEffect(() => {

console.log(“The button of the “ + componentName + “ is clicked “ + count + “ times.”); } , [count,componentName]);

return resetCounter;

}

export default useMyCustom ;

Using the Custom Hook in application :

import React from “react”;

import useMyCustom from “./useMyCustom”;

function MyComponent(props){

const clickedButton = useMyCustom(0,”MyComponent”);

return( <button onClick={clickedButton}>Click</button>);

}

export default MyComponent;

Use of Custom Hooks:

The main reason of creating custom hooks is to maintain the concept of DRY(Don’t Repeat Yourself) in your React apps. i.e if you have some functional component to be made , using the built-in hooks you might have to repeat the code and logic into many files. Instead we create a custom hook and call it into multiple files as many time we want.

Advantages Of Custom Hooks:

Custom React JS hooks offer these major benefits over Built-in hooks:

1. Reusability :

React js custom hooks offer reusability as they can be reused any number of times in various applications or different components of same application.

1. Readability :

Instead of using High Order Components, hooks can be used to improve the readability of code. Code becomes complex to read due to layers of HOCs, while using Custom hooks we can create more simpler, cleaner and readable codes.

1. Testability :

While doing integration tests over several HOCs it becomes a difficult work. Using custom hooks instead can allow to combine containers and components into one component and perform tests over them.

1. Enhanced rendering:

Custom Hooks also enhances rendering speed of the code since it does not need to be re-rendered again and again while rendering whole code.

1. Combining Effects :

Custom hooks since providing us facility to include built-in hooks and other custom hooks as well, we can combine features of several hooks into one custom hook.