**REACTJS : 30-4-2023**

---------------------

**Q)explain Boilerplate**

A React boilerplate refers to a standardized set of files and configurations that provide a starting point for building a React application. It includes the basic project structure, configuration files, and dependencies needed to set up and run a React project.

**A typical React boilerplate includes the following:**

**1)build folder :** build folder is a directory that contains the compiled and optimized version of your React application. It is typically generated when you build your application for production or deployment

**2)Node modules:** Node modules are JavaScript packages or libraries that can be installed and used in Node.js applications. They encapsulate reusable code, functions, or utilities that developers can incorporate into their projects. Node modules can be installed via npm (Node Package Manager) or yarn, which are package managers for Node.js. These modules can provide various functionalities, such as HTTP servers, database connectors, utility functions, UI components, and more. In a React project, Node modules are commonly used to install and manage dependencies like React itself, routing libraries, state management tools (e.g., Redux), CSS frameworks, and other third-party libraries required for the application

**3)Public folder:** This folder contains the static assets that are publicly accessible, such as HTML files, images, or fonts.

**4)Src folder:** This folder contains the main source code of the application. It usually includes the following files:

index.js: The entry point of the application where React is rendered into the DOM.

**App.js:** The main component of the application that represents the root of the component tree. It typically renders other components and manages the application's state.

**Components:** A directory that holds reusable React components used throughout the application. Components are usually organized in separate files and folders based on their functionality.

**Styles:** A directory that contains CSS or CSS-in-JS files to style the components.

**Assets:** A directory for storing project-specific assets like images or fonts.

**Redux:** A directory to manage the Redux-related files, such as actions, reducers, and the store configuration, if Redux is used in the project.

**Routes:** A directory that defines the application's routing logic if it utilizes a library like React Router.

**5)Package.json:** This file lists the project's dependencies, scripts, and other metadata. It specifies the required packages and their versions.

A boilerplate provides a standardized project structure and configurations, saving time and effort when setting up a new React application. It sets up the initial environment, allowing developers to focus on building features rather than worrying about setting up the basic infrastructure. Additionally, using a boilerplate ensures consistency across projects within a team or community by establishing best practices and conventions

-------------------------------------------------------------------------------------------------------

**Q) Write a functional component explain ?**  
**index.js**

import React from 'react'

import ReactDOM from 'react-dom'

function App(){

return <h1> welcome to reactjs </h1>

}

ReactDOM.render(<App />,document.getElementById('root'))

------------------------

**1)Importing React and ReactDOM:**

import React from 'react'

import ReactDOM from 'react-dom'

This code imports the React and ReactDOM libraries. React is the main library for building React components and managing the application's state, while ReactDOM is responsible for rendering React components into the DOM (Document Object Model).

**2)Defining the App component:**

function App() {

return <h1>Welcome to ReactJS</h1>

}

This code defines a functional component called App. Functional components are a way to define React components using JavaScript functions. In this case, the App component is a regular JavaScript function that returns JSX. JSX is a syntax extension for JavaScript that allows you to write HTML-like code within your JavaScript code. In this case, the App component returns an <h1> element with the text "Welcome to ReactJS".

**3)Rendering the App component:**

ReactDOM.render(<App />, document.getElementById('root'))

This code uses the ReactDOM.render() method to render the App component. It takes two arguments: the first argument is the JSX element to render, which in this case is the <App /> component, and the second argument is the DOM element where the component should be rendered. In this code, the component will be rendered inside the element with the id 'root'. This element is typically a <div> in the HTML file where your React application is mounted.

When this code is executed, it will render the App component, which will display the "Welcome to ReactJS" heading within the designated DOM element. This is the entry point of the application, where React takes over the rendering process and manages the component tree.

Note: The code you provided uses the function declaration syntax for the App component. You can also use the arrow function syntax (const App = () => { ... }) to define functional components. Both syntaxes are valid and achieve the same result.

-------------------------------------------------------------------------------------------------

**Q) props for functional components**

Certainly! Here's an example of a React program using props with functional components:

Index.js (Working in VS code )

import React from 'react';

function Greeting(props) {

return <h1>Hello, {props.name}!</h1>;

}

function App() {

return (

<div>

<Greeting name="Alice" />

<Greeting name="Bob" />

<Greeting name="Charlie" />

</div>

);

}

export default App;

-----------------

In this example, we have a functional component called Greeting that accepts a name prop. It renders an <h1> element that displays a personalized greeting with the value of the name prop.

The App component is the entry point of the application. Inside the App component, we render multiple instances of the Greeting component, passing different names as props.

When the App component is rendered, it will generate the following HTML:

<div>

<h1>Hello, Alice!</h1>

<h1>Hello, Bob!</h1>

<h1>Hello, Charlie!</h1>

</div>

Each Greeting component receives a different name prop, resulting in a personalized greeting for each rendered instance.

By utilizing props, functional components can receive external data and customize their behavior and rendering based on the values passed to them.

-----------------------------------------------

**Q) what is drawback in REACTJS with example**

**Example- 1**

While ReactJS offers numerous advantages for building web applications, there are also some drawbacks to consider. Here are a few common drawbacks of ReactJS:

**Steep Learning Curve:** ReactJS has a steep learning curve, especially for developers who are new to JavaScript frameworks or have a limited understanding of JavaScript itself. React introduces concepts like JSX (a syntax extension for JavaScript) and component-based architecture, which can be challenging to grasp initially.

**Complexity for Small Projects:** ReactJS is a powerful library designed for building complex applications. However, for small projects or simple websites, using ReactJS may introduce unnecessary complexity. In such cases, a simpler framework or vanilla JavaScript may be more appropriate.

**Tooling and Configuration Overhead:** ReactJS requires additional tooling and build configuration. Setting up a development environment with tools like Babel, Webpack, or Create React App may involve a certain level of complexity. Managing the build process and ensuring compatibility across different versions of dependencies can also be challenging.

**Lack of Official Opinionated Structure:** ReactJS is a library, not a framework, which means it provides flexibility but leaves architectural decisions up to the developer. This lack of opinionated structure can lead to inconsistent project structures and coding patterns among different development teams.

**Performance Considerations:** Although ReactJS is known for its efficient rendering algorithm (virtual DOM), improper usage or inefficient component design can impact performance. In some cases, excessive re-renders or inefficient state management can lead to suboptimal performance. It's important to optimize React components and employ performance optimization techniques when necessary.

Here's an example that demonstrates one of the drawbacks mentioned above—the learning curve:

// MyComponent.js

import React from 'react';

import ReactDOM from 'react-dom';

const MyComponent = () => {

return <h1>Hello, React!</h1>;

};

ReactDOM.render(<MyComponent />, document.getElementById('root'));

------

In the above code, we have a simple React component that renders a heading. However, if a developer who is unfamiliar with ReactJS encounters this code, they may find it confusing. The usage of JSX and the component-based approach might be unfamiliar, requiring them to invest time in learning React's syntax and concepts before feeling comfortable working with it.

It's worth noting that while there are drawbacks, ReactJS has a large and supportive community, extensive documentation, and a wide range of third-party libraries and tools available. These resources can help developers overcome the challenges and make the most of React's capabilities.

------------------------------------------------------------------------------------------

**Q) what are the drawback in REACTJS**

**example-2**

While ReactJS offers many advantages, there are also some drawbacks to consider:

**Steep Learning Curve:** React has a steep learning curve, especially for developers who are new to JavaScript frameworks or functional programming concepts. It requires an understanding of concepts like JSX, virtual DOM, state management, and component lifecycle.

**Boilerplate Code:** React encourages writing reusable components, but this can result in some boilerplate code, especially for complex applications. Setting up the initial project structure and configuring tools can also be time-consuming.

**Complex Configuration:** As React is a library and not a full-fledged framework, additional libraries and tools are often required for a complete development setup. Configuring build tools like webpack or setting up a development server can be challenging for beginners.

**High Pace of Change:** React ecosystem evolves rapidly, which means there are frequent updates, new libraries, and changes in best practices. This can result in the need for regular updates and adaptations in existing codebases, requiring developers to stay up-to-date with the latest trends.

**Poor SEO Performance:** By default, React applications are rendered on the client-side, which can affect search engine optimization (SEO). Although server-side rendering (SSR) can address this issue, it adds complexity to the development process.

**Lack of Opinionated Structure:** React doesn't enforce a specific project structure or architecture, leaving it up to developers to make decisions. This can lead to inconsistent codebases if not properly managed.

React Knowledge Alone May Not Be Enough: While React is a popular library, it's not a complete solution for building web applications. Additional knowledge and integration with other libraries or frameworks like Redux, React Router, or TypeScript may be required for more advanced features and better state management.

It's important to note that many of these drawbacks can be mitigated with proper training, experience, and the use of additional tools and libraries. React's vibrant ecosystem, large community, and extensive documentation also help in overcoming these challenges.

--------------------------------------------------------

**1) What is different between Functional Component and Class Component?**

(Functional Components or stateless components are the most widely used type of components with a basic syntax of JavaScript Functions that return the JSX code or the HTML code.

They are called as stateless components because they accept the data and display it accordingly and are mainly responsible for the rendering of the UI.

Class based or stateful Components are the ES6 classes that extend the component class of the react library. They are known as stateful components because they are responsible for the implementation of the logic. It has different phases of the React lifecycle including rendering, mounting, updating and unmounting stage.)

In React, there are two ways to create a component: Functional Components and Class Components.

Functional Components are JavaScript functions that return a React element. They are stateless and receive data as props, which they use to render the UI. Here's an example of a functional component that displays a welcome message:

**for functional Component :**

**App.js (default code)**

import './App.css';

import {Component} from 'react';

class App extends Component {

}

export default App;

-----------------------

**index.js code:**

import React from 'react';

import ReactDOM from 'react-dom';

function Greeting(props) {

return <h1>Hello, {props.name}!</h1>;

}

function App() {

return (

<div>

<Greeting name="Alice" />

<Greeting name="Bob" />

<Greeting name="Charlie" />

</div>

);

}

ReactDOM.render(<App/> ,document.getElementById('root'));

**-------------**

**same program for Class Component :**

**index.js code:**

import React from 'react';

import ReactDOM from 'react-dom';

class Greeting extends React.Component {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

class Clock extends React.Component {

render() {

return (

<div>

<Greeting name="Alice" />

<Greeting name="Bob" />

<Greeting name="Charlie" />

</div>

);

}

}

ReactDOM.render(<Clock/> ,document.getElementById('root'));

------------

Class Components, on the other hand, are ES6 classes that extend the React.Component class. They have a render() method that returns a React element. They can have state and lifecycle methods like componentDidMount() and componentWillUnmount(). Here's an example of a class component that displays the current time:

**index.js (Working)**

import React from 'react';

import ReactDOM from 'react-dom';

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

componentDidMount() {

this.timerID = setInterval(

() => this.tick(),

1000

);

}

componentWillUnmount() {

clearInterval(this.timerID);

}

tick() {

this.setState({

date: new Date()

});

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

ReactDOM.render(<Clock/> ,document.getElementById('root'));

In summary, the main differences between Functional Components and Class Components are:

Functional Components are simple functions that return a React element, while Class Components are ES6 classes that extend React.Component.

Functional Components are stateless and receive data as props, while Class Components can have state and lifecycle methods.

Functional Components are easier to read and write, while Class Components have more features and are better suited for complex components with state and lifecycle methods

**------------------------------------------------------------------------------------------------**

**2) Statefull Component and Stateless Component (My Question)? (Ans-1, Ques)**

In React, components are classified into two main types based on their capability to manage state - Stateful Component and Stateless Component.

**Stateful Component:**

Stateful components are also known as Class components, and they are capable of managing state and lifecycle methods.

They are defined using ES6 class syntax and extend the React.Component class.

They use the this.state object to store and manage the component's data, and the data can be updated using the setState() method.

They can also implement lifecycle methods such as componentDidMount(), componentDidUpdate(), etc.

Here is an example of a stateful component:

index.js (Working in VScode)

import React from 'react';

import ReactDOM from 'react-dom';

function Greeting(props) {

return <h1>Hello, {props.name}!</h1>;

}

function App() {

return (

<div>

<Greeting name="Alice" />

<Greeting name="Bob" />

<Greeting name="Charlie" />

</div>

);

}

ReactDOM.render(< App/>, document.getElementById('root'))

----------------------------

**Stateless Component:**

Stateless components are also known as Functional components, and they do not have any state or lifecycle methods.

They are defined using a function that returns a React element, and they receive data through props.

They are easier to write and understand than stateful components, and they are typically used for simple UI components.

Here is an example of a stateless component:

import React from 'react';

import ReactDOM from 'react-dom'

function Greeting(props) {

return (

<div>

<h2>Hello, {props.name}!</h2>

</div>

);

}

ReactDOM.render(<Greeting />, document.getElementById('root') );

------------------------------------------------------------------------------------------------

**3) How to pass the data from one Component to another Component? What are the different ways?**

In React, there are several ways to pass data from one component to another component:

Props: Props (short for "properties") are a way to pass data from a parent component to a child component. The parent component passes data to the child component through props. The child component receives the props as an argument and uses them to render its output. Here's an example:

Parent component:

**(Parent Component)**

**index.js (Waiting VS code)**

import React from 'react';

import ReactDOM from 'react-dom'

import ChildComponent from './Child component';

function ParentComponent() {

const name = "John";

return (

<div>

<ChildComponent name={name} />

</div>

);

}

ReactDOM.render(<ParentComponent/> ,document.getElementById('root'));

----------------

**Child component:**

ChildComponent.js(Waiting VS code)

import React from 'react';

function ChildComponent(props) {

return (

<div>

<h2>Hello, {props.name}!</h2>

</div>

);

}

export default ChildComponent;

----------------------------------------

**Context:** Context provides a way to pass data through the component tree without having to pass props down manually at every level. It's useful when you have data that needs to be accessed by many components at different levels of the component tree. Here's an example:

index.js(ParentComponent) (Working in VS code)

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

import ReactDOM from 'react-dom'

import ChildComponent from './ChildComponent'

export const MyContext = createContext();

function ParentComponent() {

const [name, setName] = useState('John');

return (

<MyContext.Provider value={name}>

<ChildComponent />

</MyContext.Provider>

);

}

ReactDOM.render(<ParentComponent/> ,document.getElementById('root'));

------------------------------------

**Child component:**

**ChildComponent.js** (Working in VS code)

import React, { useContext } from 'react';

import { MyContext } from './index';

function ChildComponent() {

const name = useContext(MyContext);

return (

<div>

<p>

<b></b> Hello, {name}!</p>

</div>

);

}

export default ChildComponent;

------------------------------------------------

**Redux:** Redux is a state management library for JavaScript applications. It provides a way to manage global state in your application and makes it easy to pass data between components. Here's an example:

index.js (waiting)

import React from 'react';

import { useSelector } from 'react-redux';

import ChildComponent from './ChildComponent';

function ParentComponent() {

const name = "John";

return (

<div>

<ChildComponent name={name} />

</div>

);

}

-------------------

function ChildComponent() {

const name = useSelector(state => state.name);

return (

<div>

<p>Hello, {name}!</p>

</div>

);

}

In this example, the parent component passes data to the child component through props, but the child component also gets data from the Redux store using the useSelector hook

---------------------------------------------------------------------------------

**4) In what scenario we will use Context? (working in VS)**

React Context is used when data needs to be passed down through multiple levels of components without having to explicitly pass it down through props at every level.

For example, imagine a website with a user authentication system. After the user logs in, you want to pass the user data down to all the components that need it, like the navigation menu and the user profile page. Instead of passing the user data through props down each level of the component tree, we can use the Context API to make the user data available to all components that need it.

Here is an example:

**UserContext.js**

// Creating the context

import { createContext } from 'react';

const UserContext = createContext({});

export default UserContext;

--------------------------------------

**// index.js**

import React from 'react';

import ReactDOM from 'react-dom';

import UserContext from './UserContext';

import Navigation from './Navigation';

import UserProfile from './UserProfile';

function App() {

const user = { name: 'John Doe', email: 'johndoe@example.com' };

return (

<UserContext.Provider value={user}>

<Navigation />

<UserProfile />

</UserContext.Provider>

);

}

ReactDOM.render(<App/> ,document.getElementById('root'));

-----------------

**// Navigation.js**

import React, { useContext } from 'react';

import UserContext from './UserContext';

function Navigation() {

const user = useContext(UserContext);

return (

<nav>

<ul>

<li>Welcome, {user.name}!</li>

</ul>

</nav>

);

}

export default Navigation;

--------------------

**// UserProfile.js**

import React, { useContext } from 'react';

import UserContext from './UserContext';

function UserProfile() {

const user = useContext(UserContext);

return (

<div>

<h2>User Profile</h2>

<p>Name: {user.name}</p>

<p>Email: {user.email}</p>

</div>

);

}

export default UserProfile;

----------------------------

In this example, we create a UserContext using the createContext function from React. We then wrap the Navigation and UserProfile components in a UserContext.Provider component, passing the user object as the value prop.

In the Navigation and UserProfile components, we use the useContext hook to access the user object from the UserContext. This allows us to avoid passing the user object down as props through multiple levels of the component tree

------------------------------------------------------------------------------------------------------

**Q) What is Virtual DOM?(working in VS)**

The Virtual DOM (VDOM) is a programming concept in React that allows the efficient updating of the UI. It is a lightweight copy of the real DOM, which is used by React to track changes and update the actual DOM.

Whenever a component's state changes, React creates a new Virtual DOM tree, which represents the updated UI. React then compares the new Virtual DOM tree with the previous one to find out the difference between them, which is known as the "diffing" process. Once the differences are identified, React updates only the affected parts of the actual DOM, instead of re-rendering the entire UI. This process helps to improve the performance of the application.

Here's an example of how the Virtual DOM works in React:

Suppose you have a simple component that renders a button and a counter. Every time the button is clicked, the counter increments by one.

index.js

import ReactDOM from 'react-dom';

import React, { useState } from 'react';

function Counter() {

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

const handleClick = () => {

setCount(count + 1);

};

return (

<div>

<button onClick={handleClick}>Click me</button>

<p>You clicked {count} times</p>

</div>

);

}

ReactDOM.render(<Counter/> ,document.getElementById('root'));

export default Counter;

When the button is clicked, the handleClick function updates the component's state, which triggers a re-render. React creates a new Virtual DOM tree that represents the updated UI, and compares it with the previous tree. It then updates only the parts of the actual DOM that have changed, which in this case is the counter element. The entire process is handled by React, and the developer doesn't need to worry about manually updating the DOM

-------------------------------------------------------------------------------------------------------

**Q)How Virtual DOM works?**

In React, the Virtual DOM is a lightweight copy of the actual DOM (Document Object Model). It is a JavaScript representation of the actual HTML DOM, used to manage the component's state and data. Whenever any data changes in the component, the entire Virtual DOM gets updated instead of the actual DOM.

Here's how the Virtual DOM works in React:

Whenever any change is made to the data or state of a component, the entire Virtual DOM gets re-rendered.

React compares the updated Virtual DOM with the previous Virtual DOM to identify the changes that occurred in the component.

React identifies the minimum number of changes required to update the actual DOM and applies those changes only.

The updated content is then displayed in the browser.

This approach is faster compared to updating the actual DOM as updating the actual DOM is an expensive operation. By using the Virtual DOM, React minimizes the changes required to update the actual DOM, thus improving the performance and user experience.

Here's an example to illustrate how the Virtual DOM works:

Suppose we have a component with a button that updates the component's state. When the user clicks the button, the state is updated, and the entire Virtual DOM gets re-rendered. React then identifies the minimum number of changes required to update the actual DOM, applies those changes, and updates the content displayed on the screen. This process is repeated every time the user clicks the button, resulting in a smooth and fast user experience

-------------------------------------------------------------------------------------------------------

**5) How can we pass the data from Child to Parent?**

In React, data usually flows from parent components to child components via props. However, if you want to pass data from a child component to a parent component, you can use callback functions.

Here's an example:

**// index.js( ParentComponent.js)- (Working in VS)**

import { useState } from 'react';

import ChildComponent from './ChildComponent';

function ParentComponent() {

const [dataFromChild, setDataFromChild] = useState('');

const handleDataFromChild = (data) => {

setDataFromChild(data);

};

return (

<div>

<p>Data from child: {dataFromChild}</p>

<ChildComponent onData={handleDataFromChild} />

</div>

);

}

export default ParentComponent;

-------------------------------------------

**// ChildComponent.js(Working in VS)**

import { useState } from 'react';

function ChildComponent({ onData }) {

const [inputValue, setInputValue] = useState('');

const handleSubmit = (e) => {

e.preventDefault();

onData(inputValue);

};

return (

<form onSubmit={handleSubmit}>

<label>

Enter data:

<input

type="text"

value={inputValue}

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

/>

</label>

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

</form>

);

}

export default ChildComponent;

--------------------------------------------

In this example, the ParentComponent renders a ChildComponent and passes a callback function handleDataFromChild to it via the onData prop. The ChildComponent uses this callback function to send data back to the parent component when the form is submitted -------------------------------------------------------------------------------------------------------

**6) If you work with React then have you worked on form Element? Like submitting for CRUD operations?**

Sure! Here's an example of a React form element with CRUD (Create, Read, Update, Delete) operations using React hooks and the Axios library for HTTP requests:

**index.js (Working VS code)**

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

import ReactDOM from 'react-dom'

import axios from 'axios';

const Form = () => {

const [data, setData] = useState([]);

const [formData, setFormData] = useState({ name: '', email: '', phone: '',dateOfBirth:'' });

const [editing, setEditing] = useState(false);

const [editingId, setEditingId] = useState('');

useEffect(() => {

axios.get('http://localhost:3000/employees').then((response) => {

setData(response.data);

});

}, []);

const handleInputChange = (event) => {

const { name, value } = event.target;

setFormData({ ...formData, [name]: value });

};

const handleSubmit = (event) => {

event.preventDefault();

if (editing) {

axios

.put(`http://localhost:3000/employees/${editingId}`, formData)

.then((response) => {

setData((prevData) => {

return prevData.map((item) =>

item.id === editingId ? response.data : item

);

});

});

setEditing(false);

setEditingId('');

} else {

axios.post('http://localhost:3000/employees', formData).then((response) => {

setData((prevData) => [...prevData, response.data]);

});

}

setFormData({ name: '', email: '', phone: '', dateOfBirth:'' });

};

const handleEdit = (item) => {

setFormData(item);

setEditing(true);

setEditingId(item.id);

};

const handleDelete = (id) => {

axios.delete(`http://localhost:3000/employees/${id}`).then(() => {

setData((prevData) => {

return prevData.filter((item) => item.id !== id);

});

});

};

return (

<div>

<h1>Form</h1>

<form onSubmit={handleSubmit}>

<input

type="text"

name="name"

value={formData.name}

onChange={handleInputChange}

placeholder="Name"

/>

<input

type="email"

name="email"

value={formData.email}

onChange={handleInputChange}

placeholder="Email"

/>

<input

type="tel"

name="phone"

value={formData.phone}

onChange={handleInputChange}

placeholder="Phone"

/>

<input

type="dob"

name="dateOfBirth"

value={formData.dateOfBirth}

onChange={handleInputChange}

placeholder="dateOfBirth"

/>

<button type="submit">{editing ? 'Save' : 'Add'}</button>

</form>

<h1>Data</h1>

<ul>

{data.map((item) => (

<li key={item.id}>

{item.name} ({item.email}, {item.phone},{item.dateOfBirth}),{' '}

<button onClick={() => handleEdit(item)}>Edit</button>{' '}

<button onClick={() => handleDelete(item.id)}>Delete</button>

</li>

))}

</ul>

</div>

);

};

ReactDOM.render(<Form/> ,document.getElementById('root'));

-----------------------

This form element uses Axios to send HTTP requests to a fake API (http://localhost:3000/employees), which handles the CRUD operations for the data. When the form is submitted, it sends a POST request to create a new data item, or a PUT request to update an existing item if the form is in editing mode. When an item is deleted, a DELETE request is sent to the API. The data is stored in the component's state using the useState hook, and the useEffect

-------------------------

Execute the following command to install JSON server

npm install -g json-server

Execute the following command to start the server

json-server --watch db.json

This automatically creates db.json file in the root project folder. Copy and paste the following JSON data in db.json file.

{

"employees": [

{

"id": 1,

"name": "Mark",

"gender": "Male",

"contactPreference": "Email",

"email": "mark@pragimtech.com",

"dateOfBirth": "1988/10/25",

"department": "3",

"isActive": true,

"photoPath": "assets/images/mark.png"

},

{

"id": 2,

"name": "Mary",

"gender": "Female",

"contactPreference": "Phone",

"phoneNumber": 2345978640,

"dateOfBirth": "1979/11/20",

"department": "2",

"isActive": true,

"photoPath": "assets/images/mary.png"

},

{

"id": 3,

"name": "John",

"gender": "Male",

"contactPreference": "Phone",

"phoneNumber": 5432978640,

"dateOfBirth": "1976/3/25",

"department": "3",

"isActive": false,

"photoPath": "assets/images/john.png

}

]

}

At this point, fire up the browser and navigate to http://localhost:3000/employees/ to see the list of all employees. You can test this REST API using a tool like fiddler.

-------------------------------------------------------------------------------------------------------

**7) In React how we can call an API? Suppose we have”n” point we have get the data from particular point right?** (waiting)

To call an API in React, we can make use of the built-in fetch method or any external library like Axios, Superagent, etc. Here's an example of calling an API using the fetch method:

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error(error));

In the above example, we are making a GET request to the https://api.example.com/data endpoint and parsing the response as JSON using the json() method. Once we get the data, we are logging it to the console. If there's any error during the API call, we are logging the error to the console as well.

To get the data from a particular point, we can modify the URL of the API endpoint accordingly. For example, if we have to get the data from point n, we can pass the point number as a parameter to the API endpoint:

javascript

Copy code

fetch(`https://api.example.com/data/${n}`)

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error(error));

In the above example, we are passing the n parameter to the API endpoint by interpolating it inside the URL using the ${} syntax

-------------------------------------------------------------------------------------------------------

**8) Life Cycle Methods Of Component In React JS?**

Definition of lifecyle :

Each and every class component in react, has their own lifeCycle, which we can monitor in 4 phases **Mounting, Updating ,UnMounting, Exception handling.** All lifeCycle methods inherited from React.Component class.

---------------

**Mounting,** the mounting phase refers to the lifecycle methods that are called when a component is being created and inserted into the DOM.

**index.js (Working in VS code-NIT )**

import React from 'react';

import ReactDOM from 'react-dom'

class App extends React.Component{

constructor(props){

super();

this.state={

n2:20

}

console.log('constructor executed');

}

static getDerivedStateFromProps(props,state){

console.log('getDerivedStateFromProps executed');

}

render(){

console.log('render executed');

return <h1>Hellow</h1>

}

componentDidMount(){

console.log('componentDidMount executed');

}

}

ReactDOM.render(<App />, document.getElementById('root'));

---------------

**Updating:(Working in VS code-NIT)**

the updating phase refers to the lifecycle methods that are called when a component is re-rendered due to changes in its props or state.

import React from 'react';

import ReactDOM from 'react-dom'

class App extends React.Component{

constructor(props){

super();

this.state={

n2:20

}

console.log('constructor executed');

setTimeout(() => {

this.setState({

n2:30

})

console.log('state updated')

}, 5000);

}

static getDerivedStateFromProps(props,state){

console.log('getDerivedStateFromProps executed');

}

render(){

console.log('render executed');

return <h1>Life cycle methods</h1>

}

componentDidMount(){

console.log('componentDidMount executed');

}

shouldComponentUpdate(props,state){

console.log('shouldComponentUpdate called')

return true;

}

getSnapshotBeforeUpdate(prevProps,prevState){

console.log('getSnapshotBeforeUpdate called')

}

componentDidUpdate(prevProps, prevState){

console.log('componentDidUpdate called')

}

}

ReactDOM.render(<App n1='10' />,document.getElementById('root'));

------------------------------------

**UnMounting :(Working in VS code-NIT)**

In ReactJS, the unmounting phase refers to the lifecycle methods that are called when a component is being removed from the DOM

import React from 'react';

import ReactDOM from 'react-dom'

class App extends React.Component {

constructor() {

super();

this.state = {show: true};

}

toggle = () => {

this.setState({show: !this.state.show});

}

render() {

return (

<div>

{this.state.show && <Child />}

<button type="button" onClick={this.toggle}>Delete Header</button>

</div>

);

}

}

class Child extends React.Component {

componentWillUnmount() {

alert("The component unmounted.");

}

render() {

return (

<h1>Hello World!</h1>

);

}

}

ReactDOM.render(<App />, document.getElementById('root'));

------------------------------------------------------------------------------------------------------

**9) for Routing and navigation which libraries using in your project?**

React Router install: npm install react-router-dom@5.2.0

React Router is a standard library for routing in React.

There are several popular libraries for routing and navigation in React projects, but two of the most commonly used ones are React Router and Reach Router.

React Router is a comprehensive routing library for React, with features like nested routes, dynamic routing, and server-side rendering. Here's an example of how to use React Router to define some routes:

**index.js (not working)**

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

function App() {

return (

<Router>

<Switch>

<Route exact path="/">

<HomePage />

</Route>

<Route path="/about">

<AboutPage />

</Route>

<Route path="/contact">

<ContactPage />

</Route>

</Switch>

</Router>

);

}

In this example, we've imported several components from React Router, including BrowserRouter, Route, and Switch. We use BrowserRouter to wrap our application and provide routing functionality, and Route to define individual routes. The Switch component ensures that only one route is matched at a time.

Reach Router is another routing library for React, with a focus on accessibility and simplicity. Here's an example of how to use Reach Router to define some routes:

---------------------

index.js (not working)

import { Router, Link } from '@reach/router';

function App() {

return (

<div>

<nav>

<Link to="/">Home</Link>

<Link to="/about">About</Link>

<Link to="/contact">Contact</Link>

</nav>

<Router>

<HomePage path="/" />

<AboutPage path="/about" />

<ContactPage path="/contact" />

</Router>

</div>

);

}

In this example, we've imported Router and Link components from Reach Router. We use Link to create navigation links, and Router to define individual routes with the path prop.

Both React Router and Reach Router are powerful libraries with their own strengths and weaknesses. The choice of which one to use depends on your specific needs and preferences.

--------------------------------------------------

**React Router is a standard library for routing in React.**

Yes, React Router is a widely used standard library for routing in React. It allows developers to declaratively define routes and handle navigation in a single-page application.

Here's an example of how to use React Router to define routes and handle navigation:

**index.js (Working VS code)**

import React from 'react';

import ReactDOM from 'react-dom'

import { BrowserRouter as Router, Routes, Switch, Route, Link } from 'react-router-dom';

import Home from './Home'

import About from './About'

import Contact from './Contact'

function App() {

return (

<Router>

<div>

<nav>

<ul>

<li>

<Link to="/">Home</Link>

</li>

<li>

<Link to="/about">About</Link>

</li>

<li>

<Link to="/contact">Contact</Link>

</li>

</ul>

</nav>

<Switch>

<Route path="/about">

<About />

</Route>

<Route path="/contact">

<Contact />

</Route>

<Route path="/">

<Home />

</Route>

</Switch>

</div>

</Router>

);

}

ReactDOM.render(<App/> ,document.getElementById('root'));

------------------------

**Home.js**

import React from 'react';

function Home() {

return (

<div>

<h1>Home</h1>

</div>

);

}

export default Home

-----------

**About.js**

import React from 'react';

function About() {

return (

<div>

<h1>About</h1>

</div>

);

}

export default About

-----------

**Contact.js**

import React from 'react';

function Contact() {

return (

<div>

<h1>Contact</h1>

</div>

);

}

export default Contact

-----------

In this example, we've imported several components from React Router, including BrowserRouter, Switch, Route, and Link. We use BrowserRouter to wrap our application and provide routing functionality, and Link to create navigation links. The Switch component ensures that only one route is matched at a time.

We define three routes using the Route component, with the path prop specifying the URL path for each route. When a user clicks on a navigation link, React Router will update the URL and render the appropriate component for the matched route.

Overall, React Router provides a simple and powerful way to handle navigation in a React application.

-------------------------------------------------------------------------------------------------------

**10) What is ReactJS?**

ReactJS is an open-source JavaScript library used for building user interfaces. It was developed by Facebook and is widely used in web development.

Here's an example of how ReactJS can be used to create a simple component that displays a greeting message:

index.js (Working Vs Code)

import React from 'react';

class Greeting extends React.Component {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

export default Greeting;

--------------

In this example, we've created a Greeting component that takes a name prop as input and displays a greeting message with that name.

To use this component in another part of our code, we can simply import it and render it like this:

**index.js**

import React from 'react';

import Greeting from './Greeting';

import ReactDOM from 'react-dom';

function App() {

return (

<div>

<Greeting />

</div>

);

}

ReactDOM.render(<App/> ,document.getElementById('root'));

--------------------------------

In this example, we've imported the Greeting component and used it twice with different names. When we run this code, we'll see a web page that displays:

**Greeting.js:**

import React from 'react';

function Greeting() {

return (

<div>

<h1>Hello, Alice!</h1>

<h3>Hello, Bob!</h3>

</div>

);

}

export default Greeting

-------------------------------

This is just a simple example, but it demonstrates the basic idea behind ReactJS: we create components that take input and produce output, and we can compose these components to build more complex user interfaces.

-------------------------------------------------------------------------------------------------------

**11) What is different between Virtual DOM and real DOM?**

In React, the Virtual DOM (VDOM) is a lightweight copy of the real DOM, which is a hierarchical representation of an HTML document. The VDOM is used by React to efficiently update the real DOM when changes are made to the UI.

The key difference between the Virtual DOM and the real DOM is that the Virtual DOM is a lightweight and fast representation of the actual DOM, while the real DOM is a complex and heavy structure that takes a lot of time to update.

React Virtual DOM and real DOM coding

Sure, here's an example that demonstrates the difference between the Virtual DOM and the real DOM in React:

**index.js (Working VS code )**

import React, { useState } from 'react';

import ReactDOM from 'react-dom'

// JSX code representing a component

const App = () => {

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

const handleClick = () => {

setCount(count + 1);

};

return (

<div>

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

<button onClick={handleClick}>Increment</button>

</div>

);

};

// Render the App component to the Real DOM

ReactDOM.render(<App />, document.getElementById('root'));

----------------------------------

In React, the Virtual DOM (VDOM) is a lightweight representation of the actual DOM (Real DOM). The Virtual DOM is a concept and technique used by React to optimize rendering performance and provide a simpler way to update the user interface.

Here's an example of how React works with the Virtual DOM and Real DOM:

React uses JSX syntax to define components and their structure. In the example above, the App component renders a <div> element containing a <p> element to display the count and a <button> element to increment the count.

When ReactDOM.render(<App />, document.getElementById('root')) is called, React creates a Virtual DOM representation of the App component.

React then compares the Virtual DOM with the Real DOM to identify the differences (referred to as the "diffing" process).

React efficiently updates only the necessary parts of the Real DOM based on the differences it found in the Virtual DOM. This allows React to minimize the number of updates and improve performance.

In our example, when the button is clicked, React updates the Virtual DOM by incrementing the count. It then performs the diffing process to determine the changes.

Finally, React updates the Real DOM by applying the necessary changes, specifically updating the <p> element to display the new count.

The use of the Virtual DOM allows React to optimize rendering and minimize direct interactions with the Real DOM, which can be a costly operation. By efficiently updating only the necessary parts, React achieves better performance and a smoother user experience.

Additionally, React handles events, state changes, and component re-rendering efficiently by leveraging the Virtual DOM and its diffing algorithm. This approach simplifies development by abstracting away many low-level DOM manipulations and providing a declarative way to build user interfaces.

-------------------------------------------------------------------------------------------------------

**12) Tell me about all Hooks?**

Hooks is a new feature (React 16.8) that lets you use state and other React features without writing a class.

Hooks are functions that let you “hook into” React state and lifecycle features from function components. Hooks don't work inside classes — they let you use React without classes. ... You can also create your own Hooks to reuse stateful behavior between different components.

**1.useState Hook :** It allows you to add state to your functional components. Using the useState hook inside a function component, you can create a piece of state without switching to class components. useState is a hook that lets you add state to a functional component

**2.useEffect Hook:**  The useEffect Hook allows you to perform side effects in your components.

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

**3.useRef Hook:** The useRef is a hook that allows to directly create a reference to the DOM element in the functional component. Syntax: const refContainer = useRef(initialValue); The useRef returns a mutable ref object. This object has a property called.

**4.useCallback Hook:**  The useCallback hook is used when you have a component in which the child is rerendering again and again without need. Pass an inline callback and an array of dependencies. useCallback will return a memoized version of the callback that only changes if one of the dependencies has changed.

**5.useMemo Hook:**  The useMemo is a hook used in the functional component of react that returns a memoized value. In Computer Science, memoization is a concept used in general when we don't need to recompute the function with a given argument for the next time as it returns the cached result.

**6.useContext Hook:** “useContext” hook is used to create common data that can be accessed throughout the component hierarchy without passing the props down manually to each level. Context defined will be available to all the child components without involving “props”.

**7.useReducer Hook:** The useReducer() hook in React lets you separate the state management from the rendering logic of the component. const [state, dispatch] = useReducer(reducer, initialState) accepts 2 argument: the reducer function and the initial state

---------------------------------------------------

React Hooks are a feature introduced in React version 16.8 that allow functional components to use state and other React features without having to use a class-based component. Here are some of the most commonly used Hooks:

**useState Hook:** The useState Hook is used to manage state in a functional component. It takes an initial state value and returns an array with two elements: the current state value and a function to update the state.

**Example: 1**

**Index.js(Working in VS)**

import React, { useState } from 'react';

import ReactDOM from 'react-dom'

function Counter() {

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

function handleClick() {

setCount(count + 1);

}

return (

<div>

<p>You clicked {count} times</p>

<button onClick={handleClick}>Click me</button>

</div>

);

}

ReactDOM.render(<Counter/> ,document.getElementById('root'));

--------------------------

**Example: 2**

**index.js**

import React, { useState } from 'react';

import ReactDOM from 'react-dom'

const UseStateCounter = () => {

const [value, setValue] = useState(0);

const reset = () =>{

setValue(0);

};

return (

<>

<section style={{margin: '4rem 0'}} >

<h2>regular counter</h2>

<h1>{value}</h1>

<button className='btn' onClick={() => setValue

(value - 1) }>

decrease

</button>

<button className='btn' onClick={(reset)}>

reset

</button>

<button className='btn' onClick={() => setValue

(value +1)}>

increase

</button>

</section>

</>

);

};

ReactDOM.render(<UseStateCounter/> ,document.getElementById('root'));

----------------

**useEffect Hook:** The useEffect Hook is used to manage side effects in a functional component. It takes a callback function that will be executed after every render.

**Example:1**

**index.js (Working VS Code)**

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

import ReactDOM from 'react-dom';

function Counter() {

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

useEffect(() => {

document.title = `You clicked ${count} times`;

});

function handleClick() {

setCount(count + 1);

}

return (

<div>

<p>You clicked {count} times</p>

<button onClick={handleClick}>Click me</button>

</div>

);

}

ReactDOM.render(<Counter/> ,document.getElementById('root'));

----------------------

**Exmple 2**

**Index.js (Working in VS code)**

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

import ReactDOM from 'react-dom'

// by default runs after every re-render

// cleanup function

// second parameter

const UseEffectBasics = () => {

const [value, setValue] = useState(0);

useEffect(()=>{

console.log('call useEffect');

if(value >0){

document.title=`New Messages(${value})`;

}

});

return(

<div>

<h1>{value}</h1>

<button className='btn' onClick={()=> setValue(value+1)}>

click me

</button>

</div>

);

};

ReactDOM.render(<UseEffectBasics/> ,document.getElementById('root'));

----------------------

**useContext Hook:** The useContext Hook is used to access a context object created by the React.createContext function.

Example:

**index.js(ParentComponent) (Working in VS code)**

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

import ReactDOM from 'react-dom'

import ChildComponent from './ChildComponent'

export const MyContext = createContext();

function ParentComponent() {

const [name, setName] = useState('John');

return (

<MyContext.Provider value={name}>

<ChildComponent />

</MyContext.Provider>

);

}

ReactDOM.render(<ParentComponent/> ,document.getElementById('root'));

--------------

**Child component:**

**ChildComponent.js (Working in VS code)**

import React, { useContext } from 'react';

import { MyContext } from './index';

function ChildComponent() {

const name = useContext(MyContext);

return (

<div>

<p>

<b></b> Hello, {name}!</p>

</div>

);

}

export default ChildComponent;

-------------------

**useReducer Hook:** The useReducer Hook is used to manage state in a more complex way than useState. It takes a reducer function and an initial state value and returns an array with the current state value and a dispatch function to update the state.

Example:

**Index.js (Working in VScode)**

**Example -1**

import React,{useReducer} from 'react';

import ReactDOM from 'react-dom'

// Reducer function

const reducer = (state, action) => {

switch (action.type) {

case 'INCREMENT':

return { count: state.count + 1 };

case 'DECREMENT':

return { count: state.count - 1 };

case 'RESET':

return { count: 0 };

default:

throw new Error('Unhandled action type');

}

};

// Component using useReducer

const Counter = () => {

const initialState = { count: 0 };

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

const handleIncrement = () => {

dispatch({ type: 'INCREMENT' });

};

const handleDecrement = () => {

dispatch({ type: 'DECREMENT' });

};

const handleReset = () => {

dispatch({ type: 'RESET' });

};

return (

<div>

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

<button onClick={handleIncrement}>Increment</button>

<button onClick={handleDecrement}>Decrement</button>

<button onClick={handleReset}>Reset</button>

</div>

);

};

ReactDOM.render(<Counter/> ,document.getElementById('root'));

----------------

**Example -2**

**Index.js**

import React, { useReducer } from 'react';

import ReactDOM from 'react-dom'

const initialState = { count: 0 };

function reducer(state, action) {

switch (action.type) {

case 'increment':

return { count: state.count + 1 };

case 'decrement':

return { count: state.count - 1 };

default:

throw new Error();

}

}

function Counter() {

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

function handleIncrement() {

dispatch({ type: 'increment' });

}

function handleDecrement() {

dispatch({ type: 'decrement' });

}

return (

<div>

<p>You clicked {state.count} times</p>

<button onClick={handleIncrement}>+</button>

<button onClick={handleDecrement}>-</button>

</div>

);

}

ReactDOM.render(<Counter/> ,document.getElementById('root'));

-------------------------

**5.useRef Hook:**

The useRef hook is a built-in hook in React that allows you to create a mutable value that persists across renders without triggering a re-render. It is commonly used to access DOM elements or to store mutable values that need to persist across renders. Here's an example of how to use the useRef hook in React:

**index.js (Working in VS Code)**

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

import ReactDOM from 'react-dom';

// preserves value

// DOES NOT trigger re-render

// target DOM nodes/elements

const UseRefBasics = () => {

const refContainer = useRef(null);

const divContainer = useRef(null) ;

const handleSubmit= (e) =>{

e.preventDefault();

console.log(refContainer.current.value);

console.log(divContainer.current);

};

useEffect(()=>{

console.log(refContainer.current);

refContainer.current.focus();

});

return(

<>

<form className='form' onSubmit={handleSubmit}>

<div>

<input type='text' ref={refContainer}/>

</div>

<button type='submit'>submit</button>

</form>

<div ref = {divContainer} >Hello World</div>

</>

);

};

ReactDOM.render(<UseRefBasics/> ,document.getElementById('root'));

-------------------------------

**6.useCallback Hook:**

The useCallback hook is used to memoize a callback function so that it is not recreated on each render unless its dependencies change. This can be beneficial for performance optimization, especially when passing callbacks to child components that rely on referential equality. Here's an example of how to use the useCallback hook in React

**index.js (working in VS code)**

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

import ReactDOM from 'react-dom'

const Counter = () => {

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

const increment = useCallback(() => {

setCount(count + 1);

}, [count]);

return (

<div>

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

<button onClick={increment}>Increment</button>

</div>

);

};

ReactDOM.render(<Counter/> ,document.getElementById('root'));

---------------

**7.useMemo Hook:**

The useMemo hook is used to memoize a value, computing it only when its dependencies change. It allows you to optimize expensive calculations or operations by caching the result and avoiding unnecessary recalculations. Here's an example of how to use the useMemo hook in React:

**index.js (Working in VS Code)**

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

import ReactDOM from 'react-dom'

const Counter = () => {

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

const doubledCount = useMemo(() => {

return count \* 2;

}, [count]);

return (

<div>

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

<p>Doubled Count: {doubledCount}</p>

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

</div>

);

};

ReactDOM.render(<Counter/> ,document.getElementById('root'));

-------------------------------------------------------------------------------------------------------

**13) If you want use UnMount means How to write?**

In ReactJS, unmounting refers to the process of removing a component from the DOM. This typically happens when a component is no longer needed or when a parent component is updated and the child component is no longer required.

To unmount a component in ReactJS, you can use the ReactDOM.unmountComponentAtNode() method. This method takes a single argument, which is the DOM node where the component is mounted. When called, this method will remove the component from the DOM and clean up any event listeners or other resources associated with the component.

Here's an example of how to use ReactDOM.unmountComponentAtNode() to unmount a component:

index.js (Some parts only working. later check)

import React from 'react';

import ReactDOM from 'react-dom';

class MyComponent extends React.Component {

render() {

return <h1>Hello, world!</h1>;

}

}

// Mount the component to a DOM node

const container = document.getElementById('root');

ReactDOM.render(<MyComponent />, container);

// Later, unmount the component

ReactDOM.unmountComponentAtNode(container);

In this example, we define a MyComponent class that renders a simple heading. We then mount the component to a DOM node using ReactDOM.render(), and finally we use ReactDOM.unmountComponentAtNode() to unmount the component from the same DOM node. This will remove the heading from the page and clean up any resources associated with the component.

-------------------------------------------------------------------------------------------------------

**14) What is different between useMemo and useCallBack?**

Both useMemo and useCallback are hooks in React that can help optimize the performance of your application by reducing unnecessary re-renders of components. However, they are used in slightly different scenarios.

useMemo is used to memoize a value. It takes a function and an array of dependencies and returns a memoized value. The function is only re-executed if any of the dependencies change. The memoized value is only recalculated if the dependencies have changed.

Here's an example of useMemo:

**index.js (working VS Studio)**

import React, { useMemo } from 'react';

import ReactDOM from 'react-dom'

function MyComponent({ a, b }) {

const result = useMemo(() => {

console.log('calculating result');

return 10 + 10;

}, [a, b]);

return <div>Result: {result}</div>;

}

ReactDOM.render(<MyComponent/> ,document.getElementById('root'));

------------

In this example, useMemo is used to calculate the result of adding a and b. The function is only re-executed if a or b change. This can help optimize performance by avoiding unnecessary calculations.

useCallback is used to memoize a function. It takes a function and an array of dependencies and returns a memoized version of the function. The memoized function is only recreated if any of the dependencies change.

Here's an example of useCallback:

**index.js (Working VS Code )**

import React, { useCallback } from 'react';

import ReactDOM from 'react-dom';

function MyComponent({ onClick }) {

const handleClick = useCallback(() => {

console.log('button clicked');

onClick();

}, [onClick]);

return <button onClick={handleClick}>Click me</button>;

}

function handleClick() {

console.log('onClick function called');

}

ReactDOM.render(<MyComponent onClick={handleClick} />, document.getElementById('root'));

In this example, useCallback is used to memoize the handleClick function. The function is only recreated if the onClick prop changes. This can help optimize performance by avoiding unnecessary function re-creations.

In summary, useMemo is used to memoize a value, while useCallback is used to memoize a function. They both take a function and an array of dependencies, and the function is only re-executed or recreated if any of the dependencies change.

-------------------------------------------------------------------------------------------------------

**15) Difference between controlcomponent and uncontrolled component ?**

In React, there are two main approaches to handle form inputs: controlled components and uncontrolled components. Let's see examples of both:

**Controlled Components:**

Controlled components are inputs whose value is controlled by React state. The value of the input is passed as a prop, and onChange event handler is used to update the state. Here's an example:

Here's an example of a controlled input:

**index.js (working VS code)**

import React, { useState } from 'react';

import ReactDOM from 'react-dom';

const ControlledInput = () => {

const [value, setValue] = useState('');

const handleChange = (event) => {

setValue(event.target.value);

};

return (

<input

type="text"

value={value}

onChange={handleChange}

/>

);

};

ReactDOM.render(<ControlledInput/>, document.getElementById('root'));

In this example, the ControlledInput component renders an input element. The value of the input is controlled by the value state, and any changes to the input trigger the handleChange function, which updates the value state accordingly.

--------------------------------------------------

**Uncontrolled Components:**

Uncontrolled components allow the form input to manage its own state internally without using React state. We can obtain the input value through the ref attribute. Here's an example:

**index.js (working VS Code)**

import React, { useRef } from 'react';

import ReactDOM from 'react-dom';

const UncontrolledInput = () => {

const inputRef = useRef();

const handleClick = () => {

console.log('Input value:', inputRef.current.value);

};

return (

<div>

<input type="text" ref={inputRef} />

<button onClick={handleClick}>Get Value</button>

</div>

);

};

ReactDOM.render(<UncontrolledInput/>, document.getElementById('root'));

--------------

In this example, the UncontrolledInput component renders an input element. The inputRef is created using the useRef hook to obtain a reference to the input element. When the "Get Value" button is clicked, the handleClick function logs the current value of the input by accessing inputRef.current.value.

Uncontrolled components are useful in scenarios where you don't need to manage the input value through React state, or when working with third-party libraries that require direct access to the DOM element.

Both approaches have their use cases, and the choice depends on the specific requirements of your application. Controlled components offer more control and validation, while uncontrolled components can be simpler to set up in certain scenarios

-------------------------------------------------------------------------------------------------

**16) Is there performance issued on controlled and uncontrolled inputs?**

There can be a performance difference between controlled and uncontrolled inputs in React, but it depends on the specific use case.

Controlled inputs can have better performance in situations where you need to re-render the input frequently, because the input value is stored in the component's state and is updated through an onChange event handler. This means that every time the input value changes, the component re-renders, but this also means that the input value is always up-to-date with the component's state.

Uncontrolled inputs can have better performance in situations where you don't need to re-render the input frequently, because the input value is not stored in the component's state, but rather accessed through a ref to the DOM node. This means that the component doesn't re-render when the input value changes, which can be more performant in some situations.

However, in most cases, the performance difference between controlled and uncontrolled inputs is negligible. It's more important to choose the approach that fits the requirements of your specific use case and is more maintainable and easier to reason about in your codebase

-------------------------------------------------------------------------------------------------------

**17) Suppose I have a component and I want prevent that component from re-rendering . How we can do that? Do we have any function? To Prevent it?**

In React, there are several ways to prevent a component from re-rendering. Here are a few approaches:

**Use shouldComponentUpdate:** You can implement the shouldComponentUpdate lifecycle method to control whether or not a component should re-render. By default, shouldComponentUpdate returns true, which means that the component will re-render whenever its props or state change. You can override this method to perform a custom comparison between the old and new props or state, and return false if the component does not need to re-render

.

**index.js (working -some need check)**

import React from 'react';

import ReactDOM from 'react-dom'

class MyComponent extends React.Component {

shouldComponentUpdate(nextProps, nextState) {

// Compare nextProps and nextState with the current props and state

// Return false if the component does not need to re-render

}

render() {

// Render the component

}

}

ReactDOM.render(<MyComponent/> ,document.getElementById('root'));

--------------------------

**Use React.memo:** You can wrap your component with the React.memo higher-order component to memoize the component and prevent it from re-rendering unnecessarily. React.memo compares the old and new props of the component and only re-renders if there are changes.

const MyComponent = React.memo(props => {

// Render the component

});

-------------------------

**Use PureComponent:** You can extend the PureComponent base class instead of Component to automatically perform a shallow comparison of the props and state and prevent unnecessary re-renders.

class MyComponent extends React.PureComponent {

render() {

// Render the component

}

}

---------------------

These are some ways to prevent a component from re-rendering in React. However, keep in mind that preventing a component from re-rendering can have unintended consequences, such as stale data or UI inconsistencies. You should use these techniques judiciously and test your components thoroughly to ensure that they behave as expected

---------------------------------------------------------------------------------------------------

**18) What is strict mode in component and why we are use it? Which situation we can do it?**

In React, Strict Mode is a development-only feature that can be used to highlight potential problems in your application's code. It performs additional checks and warnings to help you write better code and avoid common mistakes.

When you wrap a component (or a group of components) in <React.StrictMode>, React will perform the following checks:

Identifying components with unsafe lifecycles

Warning about legacy string ref API usage

Detecting unexpected side effects

Detecting and warning about legacy context API usage

Warning about deprecated findDOMNode usage

These checks are designed to help you catch common issues during development, such as using deprecated APIs, or relying on unsafe lifecycle methods.

To enable Strict Mode, you simply need to wrap your component (or your entire application) with the <React.StrictMode> component:

**index.js (working VS code)**

import React from 'react';

import ReactDOM from 'react-dom';

import Modal from './Modal';

ReactDOM.render(

<React.StrictMode>

<Modal />

</React.StrictMode>,

document.getElementById('root')

);

-------

**Modal.js**

import React from 'react';

const Modal =() =>{

return <div>

<h2>Welcome to StrictMode </h2>

</div>

}

export default Modal;

Strict Mode is recommended for use during development, but should not be used in production environments. It is particularly useful in larger codebases, where it can help catch issues that may otherwise go unnoticed.

-------------------------------------------------------------------------------------------------------

**19) In React , We have parent and child component right. I need to pass the data from parent component to child OR child to parent . So how we can do that. And If we there is no relationship between two component so then how we can pass the data?**

In React, when two components have no direct relationship, we can still pass data between them using a technique called "prop drilling". Prop drilling involves passing data down from a parent component to a child component through one or more intermediary components that don't use the data themselves.

Here's an example to illustrate how prop drilling works:

**example-1**

**index.js (working VScode)**

import React, { useState } from 'react';

import ReactDOM from 'react-dom'

function ParentComponent() {

const [message, setMessage] = useState('Hello from parent!');

return (

<div>

<ChildComponent message={message} />

</div>

);

}

function ChildComponent(props) {

return (

<div>

<GrandChildComponent message={props.message} />

</div>

);

}

function GrandChildComponent(props) {

return (

<div>

<p>{props.message}</p>

</div>

);

}

ReactDOM.render(<ParentComponent/> ,document.getElementById('root'));

-------------------------------------------------------

**example-2**

**index.js**

import React from 'react';

import ReactDOM from 'react-dom'

// Grandchild component

const GrandchildComponent = ({ message }) => {

return <p>{message}</p>;

};

// Child component

const ChildComponent = ({ message }) => {

return <GrandchildComponent message={message} />;

};

// Parent component

const ParentComponent = () => {

const message = "Hello from parent!";

return <ChildComponent message={message} />;

};

// export default ParentComponent;

ReactDOM.render(<ParentComponent/> ,document.getElementById('root'));

----------------------

In this example, we have three components: ParentComponent, ChildComponent, and GrandChildComponent.

The ParentComponent holds the state of a message, and passes it down to the ChildComponent via a prop named message.

The ChildComponent then passes the message prop down to the GrandChildComponent.

Finally, the GrandChildComponent displays the message prop in a paragraph element.

By using prop drilling, we can pass data from the ParentComponent all the way down to the GrandChildComponent, even though they have no direct relationship.

While prop drilling can work for simple cases, it can become unwieldy and difficult to maintain in larger applications. In these cases, it may be better to use a state management library like Redux or React Context to handle the flow of data between components.

-------------------------------------------------------------------------------------------------------

**20) What is Higher Order Component or what is HOC in React?**

In React, a Higher-Order Component (HOC) is a function that takes a component as input and returns a new component with enhanced functionality. HOCs are a way of sharing common functionality across multiple components without duplicating code.

***(Working but some change need for fulfill)***

**Example 1:**

**index.js (working -from codevolution)**

import React from 'react';

import ReactDOM from 'react-dom'

import ClickCounter from './ClickCounter'

import HoverCounter from './HoverCounter'

function App(){

return(

<div>

<ClickCounter />

<HoverCounter />

</div>

)

}

ReactDOM.render(<App /> ,document.getElementById('root'));

-------------------------

**ClickCounter.js**

import React , { useState }from 'react';

import UpdatedComponent from './withCounter'

function ClickCounter(props){

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

// const incrementCount = () =>{

// setCount(count + 1)

// }

const {count, incrementCount} = props;

return(

<div>

<button onClick={incrementCount} >

Clicked {count} times

</button>

</div>

)

}

export default UpdatedComponent (ClickCounter)

--------------------------------

**HoverCounter.js**

import React,{useState} from 'react';

import UpdatedComponent from './withCounter'

function ClickCounter(props){

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

// const incrementCount = () =>{

// setCount(count + 1)

// }

const {count, incrementCount} = props;

return(

<div>

<h2

onMouseOver={incrementCount}>

Hovered {count} times

</h2>

</div>

)

}

export default UpdatedComponent (ClickCounter)

--------------------------------

**withCounter.js**

import React,{useState} from 'react';

const UpdatedComponent = OriginalComponent => {

function NewComponent(){

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

const incrementCount = () => {

setCount(count + 1)

}

return(

<OriginalComponent

count={count}

incrementCount={incrementCount}

/>

)

// return <OriginalComponent name="sakthivlel"/>

}

return NewComponent

}

export default UpdatedComponent

-----------------------

In React, a Higher-Order Component (HOC) is a function that takes a component as input and returns a new component with enhanced functionality. HOCs are a way of sharing common functionality across multiple components without duplicating code.

Here's an example of a Higher-Order Component that adds a "click count" feature to any component:

**Example 2:**

**withClickCount.js (notworking)(no need this code )**

import React, { useState } from 'react';

function withClickCount(WrappedComponent) {

return function ClickCount(props) {

const [clickCount, setClickCount] = useState(0);

function handleClick() {

setClickCount(clickCount + 1);

}

return (

<WrappedComponent

{...props}

clickCount={clickCount}

handleClick={handleClick}

/>

);

};

}

In this example, we define a Higher-Order Component withClickCount that takes a component as input and returns a new component that includes a clickCount prop and a handleClick function. The clickCount state variable keeps track of the number of times the component has been clicked, and the handleClick function increments the count when the component is clicked.

Here's how we can use this Higher-Order Component to enhance a simple button component:

**index.js (notworkking)**

import React from 'react';

import withClickCount from './withClickCount';

function Button({ label, handleClick }) {

return <button onClick={handleClick}>{label}</button>;

}

const ClickCountButton = withClickCount(Button);

function App() {

return (

<div>

<ClickCountButton label="Click me" />

</div>

);

}

-------------

In this example, we use the withClickCount Higher-Order Component to create a new component ClickCountButton that includes the clickCount prop and handleClick function. We then use the ClickCountButton in our app and pass it a label prop as normal.

By using Higher-Order Components, we can easily add common functionality to multiple components without duplicating code. This helps to keep our code DRY (Don't Repeat Yourself) and makes it easier to maintain and refactor in the future.

-------------------------------------------------------------------------------------------------------

**21) Why we are use arrow function in reactJs?**

Arrow functions in React can be useful in several ways:

Defining event handlers: Arrow functions are often used to define event handlers in React. This is because arrow functions automatically bind the this keyword to the component instance, so you don't need to manually bind it in the constructor or use the bind method. For example:

for classcomponenent:

index.js

import React from 'react';

import ReactDOM from 'react-dom'

class MyComponent extends React.Component {

handleClick = () => {

console.log("hello welcome")

}

render() {

return (

<button onClick={this.handleClick}>Click me</button>

);

}

}

ReactDOM.render(<MyComponent/>, document.getElementById("root"))

-----------------------

for functional componenent

index.js

import React from 'react';

import ReactDOM from 'react-dom'

const MyComponent = () =>{

const handleClick = () => {

console.log("hello welcome")

}

return(

<div>

<button onClick={handleClick}>Click me</button>

</div>

)

}

ReactDOM.render(<MyComponent/>, document.getElementById("root"))

-------------------

In this example, we define the handleClick event handler as an arrow function, which automatically binds this to the MyComponent instance. We can then pass this function directly as the onClick prop of the button element.

Defining component methods: Arrow functions can also be used to define methods on the component class. This is similar to defining regular class methods, but with a more concise syntax. For example:

import React from 'react';

class MyComponent extends React.Component {

myMethod = () => {

// method logic here

}

render() {

return (

<div>My component</div>

);

}

}

export default MyComponent;

-------------------------

In this example, we define the myMethod method as an arrow function, which can be called from anywhere within the component class using this.myMethod(). Note that this syntax is equivalent to defining the method using the function keyword.

Simplifying code: Arrow functions can help simplify your code by reducing the amount of boilerplate required for certain operations. For example, you can use arrow functions to map over an array and render a list of items:

import React from 'react';

function MyComponent() {

const items = ['apple', 'banana', 'cherry'];

return (

<ul>

{items.map(item => <li>{item}</li>)}

</ul>

);

}

export default MyComponent;

----

In this example, we use an arrow function in the map method to render a list of li elements for each item in the items array. This can be a more concise and readable way to write this code compared to using a traditional function syntax

-------------------------------------------------------------------------------------------------------

**22) Do you know about render props ? what is it?**

In React, a render prop is a technique for sharing code between components using a prop whose value is a function. The function takes some data as input and returns some JSX that can be rendered by the component that receives the prop.

Here is an example of how to use render props in React:

**index.js (working in vscode)**

import React from 'react';

import ReactDom from 'react-dom';

class MouseTracker extends React.Component {

constructor(props) {

super(props);

this.state = {

x: 0,

y: 0,

};

}

handleMouseMove = event => {

this.setState({

x: event.clientX,

y: event.clientY,

});

};

render() {

return (

<div style={{ height: '100%' }} onMouseMove={this.handleMouseMove}>

{this.props.render(this.state)}

</div>

);

}

}

function App() {

return (

<div style={{ height: '100%' }}>

<MouseTracker render={mouse => (

<p>The current mouse position is ({mouse.x}, {mouse.y})</p>

)} />

</div>

);

}

ReactDom.render(<App/>, document.getElementById('root')) ;

-------------------

In this example, we define a MouseTracker component that listens for mouse movement events and passes the current mouse position as an object to a render prop function provided by the parent component. The parent component, App, renders the MouseTracker component and provides a function as the render prop. The render function takes the mouse object as an argument and returns a p element displaying the current mouse position.

By using a render prop, we can share the mouse position data between the MouseTracker and App components without having to use complex state management or prop drilling techniques. This can help make our code more reusable and maintainable.

-------------------------------------------------------------------------------------------------------

**23) Can you tell me different type component in React? (ok)**

In React, there are several types of components that can be used to create a user interface:

**Functional components:** These are the simplest type of components and are just JavaScript functions that return JSX. They can be used for simple UI elements that don't have any state or lifecycle methods. Here's an example:

function Greeting(props) {

return <h1>Hello, {props.name}!</h1>;

}

**Class components:** These are JavaScript classes that extend the React.Component class and implement a render() method. They can be used for more complex UI elements that need to manage state or have lifecycle methods. Here's an example:

class Counter extends React.Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

handleClick = () => {

this.setState({ count: this.state.count + 1 });

};

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={this.handleClick}>Increment</button>

</div>

);

}

}

**Higher-order components (HOCs):** These are functions that take a component as input and return a new component with additional functionality. They can be used to reuse code and add behavior to existing components. Here's an example:

function withLogging(WrappedComponent) {

return class extends React.Component {

componentDidMount() {

console.log(`Component ${WrappedComponent.name} mounted.`);

}

render() {

return <WrappedComponent {...this.props} />;

}

};

}

const LoggedInButton = withLogging(Button);

Stateless functional components: These are functional components that don't have any state or lifecycle methods. They are similar to regular functional components, but can be used to improve performance in some cases. Here's an example:

const Greeting = ({ name }) => <h1>Hello, {name}!</h1>;

Overall, the choice of which type of component to use depends on the specific requirements of the UI element and the preferences of the developer.

-------------------------------------------------------------------------------------------------------

**24) Have you use pure Component?**

React PureComponent is a class component that inherits from React.Component and provides an optimized way to handle updates. It is used when the component's render method produces the same result given the same props and state.

The PureComponent implements the shouldComponentUpdate lifecycle method, which performs a shallow comparison of the component's props and state to determine if a re-render is necessary. If the props and state have not changed, the component will not re-render, which can improve performance.

Here's an example of how to use PureComponent in React:

**index.js (Working in VScode)**

import React, { PureComponent } from 'react';

import ReactDOM from 'react-dom';

class MyComponent extends PureComponent {

constructor(props) {

super(props);

this.state = { count: 0 };

}

handleClick = () => {

this.setState({ count: this.state.count + 1 });

};

render() {

console.log('Render MyComponent');

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={this.handleClick}>Increment</button>

</div>

);

}

}

ReactDOM.render(<MyComponent />, document.getElementById("root"));

--------

In the example above, the MyComponent class extends PureComponent and has a count state that gets incremented when the button is clicked. The render method displays the count value.

If you run the above code and click the button, you'll see in the console log that the render method is called on every click, even though the count state hasn't changed. This is because the default shouldComponentUpdate method of the React.Component base class always returns true.

However, if you change the MyComponent class to extend PureComponent, you'll notice that the render method is only called on the first click and subsequent clicks don't trigger a re-render. This is because the shouldComponentUpdate method of the PureComponent class performs a shallow comparison of the count state and determines that it hasn't changed.

-------------------------------------------------------------------------------------------------------

**25) Inside the package.json there is dependancy and dev.dependancy? (ok)**

In a React project, the package.json file contains information about the project dependencies and dev dependencies. Dependencies are external packages that are required to run the application, while dev dependencies are packages that are only needed during development and testing.

Here's an example of what the dependencies and dev dependencies sections of a package.json file might look like:

json

Copy code

{

"name": "my-react-app",

"version": "1.0.0",

"dependencies": {

"react": "^17.0.2",

"react-dom": "^17.0.2",

"react-scripts": "4.0.3"

},

"devDependencies": {

"eslint": "^7.27.0",

"prettier": "^2.3.2"

}

}

In this example, the dependencies section includes three packages that are required for the React app to run: react, react-dom, and react-scripts. The ^ symbol before the version numbers means that the app can use any version of the package that is greater than or equal to the specified version.

The devDependencies section includes two packages that are only needed during development and testing: eslint and prettier. These packages are used for code linting and formatting.

When the app is installed using npm install, both the dependencies and dev dependencies will be installed. However, the dev dependencies will not be included in the production build of the app.

-------------------------------------------------------------------------------------------------------

**26) npm install axios --save , npm install packagename . what is the different between --save and without save? (ok)**

react npm install axios --save , npm install packagename with example

Sure, here are some examples of using npm install to install packages in a React project:

Installing a package and saving it as a dependency in package.json using the --save flag:

css

Copy code

npm install axios --save

This command installs the axios package and saves it as a dependency in the package.json file of the project. This means that when someone else clones the project and runs npm install, axios will be installed automatically.

Installing a package without saving it as a dependency:

Copy code

npm install moment

This command installs the moment package without saving it as a dependency in the package.json file of the project. This means that if someone else clones the project and runs npm install, moment will not be installed automatically. Instead, they will need to manually install it using npm install moment.

Note that in newer versions of npm, the --save flag is no longer necessary, as packages are automatically saved as dependencies by default. So, you can simply run:

Copy code

npm install axios

or

Copy code

npm install moment

and the package will be installed and saved as a dependency in package.json automatically.

---------------------

what is the different between --save and without save

In older versions of npm, when we install a package without the --save option, the package is installed locally for development purposes only, and not saved as a dependency in the package.json file. This meant that if we were to move the project to a different machine or share it with someone else, they would not be able to install the same packages as us without also manually installing them.

However, starting from npm version 5.0, the --save option is now the default behavior when we install a package using npm install. This means that the package is automatically added as a dependency in the package.json file.

Therefore, the only difference between using npm install axios and npm install axios --save is that the former only installs the package locally for development purposes, while the latter installs the package and saves it as a dependency in the package.json file.

-------------------------------------------------------------------------------------------------------

**27) How React works internally? (ok)**

React works internally by maintaining a virtual DOM (Document Object Model) which is a lightweight copy of the actual DOM. The virtual DOM updates the actual DOM by only making changes to the parts that have been updated. This helps improve performance by minimizing unnecessary updates to the DOM.

Here is an example of how React works internally:

1).When a React component is rendered, a virtual DOM is created in memory.

2).The virtual DOM is then updated with any changes to the component's state or props.

3).React then compares the updated virtual DOM with the previous version of the virtual DOM to identify the differences.

4).React then updates only the parts of the actual DOM that have changed, based on the differences in the virtual DOM.

5).Finally, React re-renders the component with the updated virtual DOM.

This process helps improve the performance of the application by minimizing the number of updates to the actual DOM.

-------------------------------------------------------------------------------------------------------

**28) What is use of the KEY for the component?**

Example 1).A “key” is a special string attribute you need to include when creating lists of elements in React. Keys are used to React to identify which items in the list are changed, updated, or deleted. In other words, we can say that keys are used to give an identity to the elements in the lists.

Example 2).

In React, the "key" is a special attribute that is used to uniquely identify and track components in a list. When rendering a list of components, React needs to be able to efficiently determine which components have changed, been added, or been removed.

The "key" attribute is added to each child component within the list and should be set to a unique identifier, such as an ID or index value. This helps React to track each component and its state, even as the list changes over time.

Here's an example of how to use "key" in a list of items:

**index.js (Working in VScode)**

import React from 'react';

import ReactDOM from 'react-dom';

const items = [

{ id: 1, name: 'Item 1' },

{ id: 2, name: 'Item 2' },

{ id: 3, name: 'Item 3' }

];

const List = () => {

return (

<ul>

{items.map(item => (

<li key={item.id}>{item.name}</li>

))}

</ul>

);

};

ReactDOM.render(<List />, document.getElementById("root"));

----

In this example, each item in the "items" array has a unique "id" property. When rendering the list of items using the "map" method, we assign the "key" attribute to the "id" value of each item. This ensures that React can efficiently track changes to the list as it is updated over time

-------------------------------------------------------------------------------------------------------

**29) Could you please tell me what is the controlled component in ReactJS and when do use?**

In ReactJS, a controlled component is a form element whose value is controlled by React. It means the React component that renders a form also controls what happens in that form on subsequent user input.

Controlled components are used when you want to achieve a behavior where all form input data is handled by a React component, and that component then decides what to do with the data. This is useful when you want to enforce input validation rules or apply some other form of data transformation before the data is submitted.

Here is an example of a controlled component in ReactJS:

**index.js (ok)**

import React, { useState } from 'react';

import ReactDOM from 'react-dom';

function LoginForm() {

const [username, setUsername] = useState('');

const [password, setPassword] = useState('');

const handleUsernameChange = (event) => {

setUsername(event.target.value);

};

const handlePasswordChange = (event) => {

setPassword(event.target.value);

};

const handleSubmit = (event) => {

event.preventDefault();

// handle form submission with username and password

};

return (

<form onSubmit={handleSubmit}>

<label>

Username:

<input type="text" value={username} onChange={handleUsernameChange} />

</label>

<label>

Password:

<input type="password" value={password} onChange={handlePasswordChange} />

</label>

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

</form>

);

}

ReactDOM.render(<LoginForm />, document.getElementById("root"));

-------

In this example, the LoginForm component has two controlled inputs - the username and password fields. The component's state manages the values of these fields. Whenever the user types something into either field, the handleUsernameChange or handlePasswordChange function updates the corresponding state value.

The value prop of the input fields is set to the corresponding state value. This means that the component always knows the current value of the input fields.

When the form is submitted, the handleSubmit function is called, which can access the current values of the username and password fields from the component's state

-------------------------------------------------------------------------------------------------------

**30) Do you know what the pure component is over the normal component?**

In React, a component can either be a normal component or a pure component.

A normal component is a regular React component that extends the React.Component class. It may have its own state or receive props from its parent components, and it may re-render whenever its state or props change.

On the other hand, a pure component is a special type of React component that extends the React.PureComponent class. It is similar to a normal component, but it has some performance optimizations built-in to prevent unnecessary re-renders.

The main difference between a normal component and a pure component is how they handle the shouldComponentUpdate lifecycle method. In a normal component, shouldComponentUpdate is called on every update, and it defaults to always returning true. In a pure component, shouldComponentUpdate is automatically implemented to compare the component's props and state to determine if it needs to re-render. This can help to reduce the number of unnecessary re-renders in your application, especially for large and complex components.

Here's an example of a normal component:

**index.js (Working VS Code)**

import React from 'react';

import ReactDOM from 'react-dom';

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.state = {

count: 0

};

}

handleClick() {

this.setState({ count: this.state.count + 1 });

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={() => this.handleClick()}>Increment</button>

</div>

);

}

}

ReactDOM.render(<MyComponent />, document.getElementById("root"));

-------------

And here's an example of a pure component:

index.js (Working)

import React from 'react';

import ReactDOM from 'react-dom';

class MyPureComponent extends React.PureComponent {

constructor(props) {

super(props);

this.state = {

count: 0

};

}

handleClick() {

this.setState({ count: this.state.count + 1 });

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={() => this.handleClick()}>Increment</button>

</div>

);

}

}

ReactDOM.render(<MyPureComponent />, document.getElementById("root"));

----

In this example, the MyPureComponent is functionally identical to the MyComponent, except that it extends React.PureComponent instead of React.Component. This means that the shouldComponentUpdate method is automatically implemented to compare the component's props and state, and the component will only re-render when necessary.

-------------------------------------------------------------------------------------------------------

**31) Which one is fast, functional component or class based component? (ok)**

React Which one is fast, functional component or class based component with example

Both functional and class components can be performant in React, but in general, functional components are considered to be faster because they have a simpler implementation and do not have the overhead of the class component's instance.

In a class component, React has to create a new instance of the component class each time it is rendered. This involves creating a new object, setting up the instance properties, and calling the constructor. In contrast, a functional component is just a simple function, which means it has less overhead.

Additionally, functional components can take advantage of the React Hooks API to manage state and side effects, which can lead to more optimized and maintainable code.

Here is an example of a functional component and a class component that both render the same output:

**Functional Component:**

**index.js(working in VS code)**

import React, { useState } from 'react';

import ReactDOM from 'react-dom';

function Counter() {

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

const handleIncrement = () => {

setCount(count + 1);

};

return (

<div>

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

<button onClick={handleIncrement}>Increment</button>

</div>

);

}

ReactDOM.render(<Counter />, document.getElementById("root"));

----------------------

**Class Component:**

**index.js (Working in VS code)**

import React from 'react';

import ReactDOM from 'react-dom';

class Counter extends React.Component {

constructor(props) {

super(props);

this.state = {

count: 0,

};

this.handleIncrement = this.handleIncrement.bind(this);

}

handleIncrement() {

this.setState({ count: this.state.count + 1 });

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={this.handleIncrement}>Increment</button>

</div>

);

}

}

ReactDOM.render(<Counter />, document.getElementById("root"));

-----

Both of these components will render a simple counter with an increment button. However, the functional component uses the useState hook to manage state, whereas the class component uses the setState method to update state. In terms of performance, the functional component is likely to be faster due to the simpler implementation and less overhead.

-------------------------------------------------------------------------------------------------------

**32) In React useRef hook and ref keyword ? What is significant of both?**

In React, the useRef hook and the ref keyword serve different purposes.

The ref keyword is used to reference a React element or component. It can be used to get a reference to the actual DOM element of a component or to get a reference to a component instance. ref is commonly used for manipulating the DOM, setting focus to an element, or triggering animations.

Here's an example of using ref keyword to get the value of an input field:

**index.js(working vs code )**

import React, { useRef } from 'react';

import ReactDOM from 'react-dom';

function InputField() {

const inputRef = useRef(null);

function handleClick() {

const inputValue = inputRef.current.value;

console.log(inputValue);

}

return (

<div>

<input type="text" ref={inputRef} />

<button onClick={handleClick}>Get Value</button>

</div>

);

}

ReactDOM.render(<InputField />, document.getElementById("root"));

------

The useRef hook, on the other hand, is used to hold a mutable reference to a value that persists between renders. This can be useful for cases like storing the previous value of a prop or state, or to access the current state value inside an event handler. useRef can also be used to store a reference to a DOM element, similar to using the ref keyword.

Here's an example of using useRef to store the previous value of a prop:

index.js (Working VS code)

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

import ReactDOM from 'react-dom';

function Counter() {

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

const [prevCount, setPrevCount] = useState(null);

const countRef = useRef(null);

useEffect(() => {

if (countRef.current !== null) {

setPrevCount(countRef.current);

}

countRef.current = count;

}, [count]);

return (

<div>

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

{prevCount !== null && <p>Previous Count: {prevCount}</p>}

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

</div>

);

}

ReactDOM.render(<Counter />, document.getElementById('root'));

-----

In summary, while ref is used to reference a React element or component, useRef is used to hold a mutable reference to a value that persists between renders.

-------------------------------------------------------------------------------------------------------

**33) How to implement key and ref that code you need to write?**

Sure, here are examples of how to implement key and ref in React:

key example:

**index.js (Working in VS code)**

iimport React from 'react';

import ReactDOM from 'react-dom';

const MyComponent = () => {

const data = [

{ id: 1, name: 'John' },

{ id: 2, name: 'Jane' },

{ id: 3, name: 'Bob' }

];

const listItems = data.map((item) =>

<li key={item.id}>{item.name}</li>

);

return (

<ul>{listItems}</ul>

);

};

ReactDOM.render(<MyComponent />, document.getElementById('root'));

---------

In the example above, key is used to help React identify each list item in the data array uniquely. By using a unique identifier for each list item, React can optimize its rendering performance by only updating the necessary items that have changed.

ref example:

index.js(Working in vs code)

import React, { useRef } from 'react';

import ReactDOM from 'react-dom';

const MyComponent = () => {

const inputRef = useRef(null);

const handleSubmit = () => {

const inputValue = inputRef.current.value;

console.log(inputValue);

};

return (

<div>

<input type="text" ref={inputRef} />

<button onClick={handleSubmit}>Submit</button>

</div>

);

};

ReactDOM.render(<MyComponent />, document.getElementById('root'));

In the example above, Inside the MyComponent function, we declare a variable called inputRef using the useRef hook. This allows us to create a reference to the input field in the DOM.

We also define a handleSubmit function that will be called when the submit button is clicked. In the handleSubmit function, we access the value entered in the input field using inputRef.current.value and log it to the console.

The return statement contains JSX code that renders an input field and a button. The ref={inputRef} attribute is used to associate the inputRef with the input field, allowing us to access its value later. The onClick event handler is set to the handleSubmit function.

-------------------------------------------------------------------------------------------------------

**34)What is JSX in React?**

JSX (JavaScript XML) is an extension to the JavaScript language syntax that allows us to write HTML-like code in our JavaScript files. It is used in React to describe the UI elements.

Here's an example of JSX:

**index.js (Working in VS Code)**

import React from 'react';

import ReactDOM from 'react-dom';

const App = () => {

const name = 'John';

return (

<div>

<h1>Hello, {name}!</h1>

<p>Welcome to my React app.</p>

</div>

);

};

ReactDOM.render(<App />, document.getElementById("root"));

----------

In this example, we are using JSX to create a functional component called App. The component returns a <div> element that contains a <h1> element with the text "Hello, John!" (where "John" is a variable defined earlier in the component) and a <p> element with the text "Welcome to my React app.".

Note that while this code looks like HTML, it is actually JavaScript. When the code is compiled, the JSX is converted to regular JavaScript function calls that create React elements.

-------------------------------------------------------------------------------------------------------

**35) What is props used for in React?**

In React, props (short for properties) are used to pass data from one component to another. They are a way of communicating between components, allowing you to customize and configure components based on the values you pass as props.

For example, consider a parent component that renders a child component:

index.js (working in VS code)

import React from 'react';

import ReactDom from 'react-dom';

import Child from './Child'

function Parent() {

const name = "John";

return (

<div>

<h1>Welcome, {name}!</h1>

<Child age={30} />

</div>

);

}

ReactDom.render(<Parent />, document.getElementById("root"));

--------------

Child.js (Working vs code )

import React from 'react'

function Child(props) {

return <p>You are {props.age} years old.</p>;

}

export default Child

--------------------

In this example, the parent component Parent renders a child component Child and passes the age prop to it with a value of 30. The child component receives the prop as an argument and displays it with the message "You are [age] years old."

Props are read-only, meaning that the child component cannot modify the values passed to it as props. Instead, it can use the values to render its output or trigger events, which are then handled by the parent component.

----------------------------------------------------------------------------------------------

**(Not Answer)**

**36) Lazy loading?**

**37) What is should component update in ReactJS?**

**38) Can you please explain me the whole process how data flow in react from one layer to another?**

**39) How to prevent re-rendering in functional component as well as class based component?**

**40) Do you know the dependency Injection?**

**41) What is Redux Architecture. How it is use in React?**

**42) What is Redux?**

**43) What is basic flow method of Redux? Like if you want have something, If you**

**want state change, Just give me brief part? How to trigger state ?**

**44)What are the differences between redux-saga and redux-thunk?**

**45) Have you heard about Redux Saga , Redux Thunk? What is basic flow , If you**

**want call the API to Reducer?**

**46) Can you Multiple reducer in Redux?**

**47) Where is state is immutable in Redux?**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**>>>>>>>>>>>>>>>>>>>>> >>End <<<<<<<<<<<<<<<<<<<<<<<<<<**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_