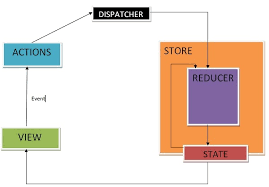
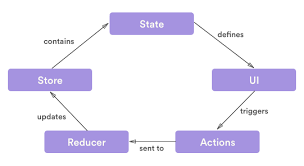
**Questions**

1. **What is redux. Explain its flow**

Redux is a predictable state container for JavaScript apps. As the application grows, it becomes difficult to keep it organized and maintain data flow. Redux solves this problem by managing application’s state with a single global object called Store. Redux fundamental principles help in maintaining consistency throughout your application, which makes debugging and testing easier.



/src/index.js

import React from 'react'

import { render } from 'react-dom'

import { Provider } from 'react-redux'

import { createStore } from 'redux';

import reducer from '../src/reducer/index'

import App from '../src/App'

import './index.css';

const store = createStore(

reducer,

window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ &&

window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()

)

render(

<Provider store = {store}>

<App />

</Provider>, document.getElementById('root')

)

====================

/src/app.js

import React, { Component } from 'react';

import './App.css';

import Counter from '../src/container/appContainer';

class App extends Component {

render() {

return (

<div className = "App">

<header className = "App-header">

<Counter/>

</header>

</div>

);

}

}

export default App;

==========================

/container/counterContainer.js

import { connect } from 'react-redux'

import Counter from '../component/counter'

import { increment, decrement, reset } from '../actions';

const mapStateToProps = (state) => {

return {

counter: state

};

};

const mapDispatchToProps = (dispatch) => {

return {

increment: () => dispatch(increment()),

decrement: () => dispatch(decrement()),

reset: () => dispatch(reset())

};

};

export default connect(mapStateToProps, mapDispatchToProps)(Counter);

==============================

Given below is the react component responsible for view part −

/component/counter.js

import React, { Component } from 'react';

class Counter extends Component {

render() {

const {counter,increment,decrement,reset} = this.props;

return (

<div className = "App">

<div>{counter}</div>

<div>

<button onClick = {increment}>INCREMENT BY 1</button>

</div>

<div>

<button onClick = {decrement}>DECREMENT BY 1</button>

</div>

<button onClick = {reset}>RESET</button>

</div>

);

}

}

export default Counter;

==============================

The following are the action creators responsible for creating an action −

/actions/index.js

export function increment() {

return {

type: 'INCREMENT'

}

}

export function decrement() {

return {

type: 'DECREMENT'

}

}

export function reset() {

return { type: 'RESET' }

}

==============================

Below, we have shown line of code for reducer file which is responsible for updating the state in Redux.

reducer/index.js

const reducer = (state = 0, action) => {

switch (action.type) {

case 'INCREMENT': return state + 1

case 'DECREMENT': return state - 1

case 'RESET' : return 0

default: return state

}

}

export default reducer;

1. **What is the need of "refs" in React JS?**

Refs provide a way to access DOM nodes or React elements created in the render method.

There are a few good use cases for refs:

* Managing focus, text selection, or media playback.
* Triggering imperative animations.
* Integrating with third-party DOM libraries.

Refs are a function provided by React to access the DOM element and the React element that you might have created on your own. They are used in cases where we want to change the value of a child component, without making use of props and all. They also provide us with good functionality as we can use callbacks with them.

// using ref

class CustomTextInput extends React.Component {

constructor(props) {

super(props);

// create a ref to store the textInput DOM element

this.textInput = React.createRef();

this.focusTextInput = this.focusTextInput.bind(this);

}

focusTextInput() {

// Explicitly focus the text input using the raw DOM API

// Note: we're accessing "current" to get the DOM node

this.textInput.current.focus();

}

render() {

// tell React that we want to associate the <input> ref

// with the `textInput` that we created in the constructor

return (

<div>

<input

type="text"

ref={this.textInput} />

<input

type="button"

value="Focus the text input"

onClick={this.focusTextInput}

/>

</div>

);

}

}

1. **How would you implement two-way data-binding in React JS?**

In React, data flows one way: from owner to child. We think that this makes your app’s code easier to understand. You can think of it as “one-way data binding.”

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you’ll often want to update some React state when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a “change” event, read from your data source (usually the DOM) and call setState() on one of your components.

import React, { Component } from 'react';

import logo from './logo.svg';

import ChildComponent from './ChildComponent';

class ParentComponent extends Component {

constructor(props) {

super(props)

this.state = {

color: 'Select a color'

};

this.changeColor = this.changeColor.bind(this);

}

changeColor(newColor){

this.setState({

color: newColor

});

}

render() {

const { color } = this.state;

return (

<div className="parent-component">

<header className="header">

<img src={logo} className="react-logo" alt="logo" />

<h4>Parent Component</h4>

<p>{color}</p>

</header>

<ChildComponent changeColor={this.changeColor}/>

</div>

);

}

}

export default ParentComponent;

import React from 'react';

export default function ChildComponent(props) {

const handleClick = e => props.changeColor(e.target.value);

return (

<div className="child-component">

<h4>Child Component</h4>

<button value="Red" onClick={handleClick}>

Red

</button>

<button value="Blue" onClick={handleClick}>

Blue

</button>

<button value="White" onClick={handleClick}>

White

</button>

</div>

);

}

1. **Difference between "props" and "state" in React JS**

**State**:

1. states are mutable.
2. states are associated with the individual components can't be used by other components.
3. states are initialized on component mount.
4. states are used for rendering dynamic changes within component.
5. states are used to manage the internal environment of a component means the data changes inside the component

**props**:

1. props are immutable.
2. you can pass props between components.
3. props are mostly used to communicate between components.
4. You can pass from parent to child directly. For passing from child to parent you need use concept of lifting up states.
5. **Use case for "shouldComponentUpdate" lifecycle method**

shouldComponentUpdate(nextProps, nextState)

import React, { Component } from "react";

class Counter1 extends Component {

constructor(props){

super(props);

this.state:{

value:0

};

this.onClick = this.onClick.bind(this);

}

shouldComponentUpdate(nextProps) {

// Rendering the component only if

// passed props value is changed

if (nextProps.value !== this.props.value) {

return true;

} else {

return false;

}

}

render() {

return (

<div>

<h2>Counter 1:</h2>

<h3>{this.props.value}</h3>

<button onClick={this.props.onClick}>Add</button>

</div>

);

}

}

export default Counter1;

1. **What is a "synthetic event" in React JS"?**

A synthetic event is a cross-browser wrapper around the browser’s native event. It has the same interface as the browser’s native event, including stopPropagation() and preventDefault(), except the events work identically across all browsers.It achieves high performance by automatically using event delegation

Every SyntheticEvent object has the following attributes:

boolean bubbles

boolean cancelable

DOMEventTarget currentTarget

boolean defaultPrevented

number eventPhase

boolean isTrusted

DOMEvent nativeEvent

void preventDefault()

boolean isDefaultPrevented()

void stopPropagation()

boolean isPropagationStopped()

void persist()

DOMEventTarget target

number timeStamp

string type

function Example() {

return (

<input

onFocus={(e) => {

console.log('Focused on input');

}}

placeholder="onFocus is triggered when you click this input."

/>

)

}

1. **What is the use of "ReactDOM.render()"?**

The ReactDOM.render() function takes two arguments, HTML code and an HTML element.

The purpose of the function is to display the specified HTML code inside the specified HTML element.

**Syntax - ReactDOM.render(element, container[, callback])**

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

1. **What is the role of a middleware in Redux?**

Redux middleware provides a third-party extension point between dispatching an action, and the moment it reaches the reducer. People use Redux middleware for logging, crash reporting, talking to an asynchronous API, routing, and more.

import { createStore, applyMiddleware } from 'redux'  
import rootReducer from './reducer'  
import { print1, print2, print3 } from './exampleAddons/middleware'  
  
const middlewareEnhancer = applyMiddleware(print1, print2, print3)  
  
// Pass enhancer as the second arg, since there's no preloadedState  
const store = createStore(rootReducer, middlewareEnhancer)  
  
export default store

Redux Thunk is a middleware. Redux Thunk allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met.

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import { createStore, applyMiddleware } from 'redux';

import thunk from 'redux-thunk';

import './index.css';

import rootReducer from './reducers';

import App from './App';

import \* as serviceWorker from './serviceWorker';

const store = createStore(rootReducer, applyMiddleware(thunk));

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

1. **What is the significance of JSX in React JS?**

JSX stands for JavaScript XML. It is simply a syntax extension of JavaScript. It allows us to directly write HTML in React (within JavaScript code)

It is faster than normal JavaScript as it performs optimizations while translating to regular JavaScript. Instead of separating the markup and logic in separated files, React uses components for this purpose.

**Advantages of JSX:**

* JSX makes it easier to write or add HTML in React.
* JSX can easily convert HTML tags to react elements.
* It is faster than regular JavaScript.
* JSX allows us to put HTML elements in DOM without using appendChild() or createElement() method.
* It is type-safe, and most of the errors can be found at compilation time.

const name = 'Josh Perez';

const element = <h1>Hello, {name}</h1>;

ReactDOM.render(

element,

document.getElementById('root')

);

1. **What is conditional rendering in React JS?**

Conditional rendering is a term to describe the ability to render different user interface (UI) markup if a condition is true or false. In React, it allows us to render different elements or components based on a condition.

1. **How to use conditional rendering in JSX?**

render() {

const isLoggedIn = this.state.isLoggedIn;

return (

<div>

{isLoggedIn

? <LogoutButton onClick={this.handleLogoutClick} />

: <LoginButton onClick={this.handleLoginClick} />

}

</div>

);

}

class App extends Component {

render() {

let {isLoggedIn} = this.state;

return (

<div className="App">

<h1>

This is a Demo showing several ways to implement Conditional Rendering in React.

</h1>

{

if(isLoggedIn){

return <button>Logout</button>

} else{

return <button>Login</button>

}

}

</div>

);

}

}

1. **Component has a table with 3 columns, pid, pname, no of visits. If you were to write a unit test to check if the last column in the table contains a number, what would be your approach to write such a test?**

describe('test MyComponent', () => {

const wrapper = mount(<MyComponent />);

const table = wrapper.find('table');

const col3 = table.find('.classname-of-last-column')

it('Check if number’, () => {

expect(col3). text().not.toBeNaN();

});

});

1. **What is the use of React.Fragment?**

A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM

render() {

return (

<React.Fragment>

<ChildA />

<ChildB />

<ChildC />

</React.Fragment>

);

}

**Shorthand Fragment:** use of ‘<>’ and ‘</>’ instead of the ‘React.Fragment’.

<>

<h2>Child-1</h2>

<p> Child-2</p>

</>

1. **State has array, how to display it in a table within a component?**

function MyTable() {

const initState = [

{ id: 1, name: "bread", quantitiy: 50, location: "cupboard" },

{ id: 2, name: "milk", quantitiy: 20, location: "fridge" },

{ id: 3, name: "water", quantitiy: 10, location: "fridge" }

];

const [state, setState] = React.useState(initState);

return (

<table>

<tr key={"header"}>

{Object.keys(state[0]).map((key) => (

<th>{key}</th>

))}

</tr>

{state.map((item) => (

<tr key={item.id}>

{Object.values(item).map((val) => (

<td>{val}</td>

))}

</tr>

))}

</table>

);

}

ReactDOM.render(<MyTable />, document.getElementById("target"));

1. **How to debug a React JS app in VS Code?**
2. Download the latest release of VS Code and install our Chrome debugger
3. Create your React app using create-react-app
4. Use the following config for your launch.jsonfile to configure the VS Code debugger and put it inside .vscode in your root folder.

{

"version": "0.2.0",

"configurations": [

{

"name": "Chrome",

"type": "chrome",

"request": "launch",

"url": "http://localhost:3000",

"webRoot": "${workspaceRoot}/src"

}

]

}

1. Start your React app by running npm start in your favorite terminal
2. Start debugging in VS Code by pressing F5or by clicking the green debug icon
3. **What is the use of Switch component in React routing?**

* Switch components are used to render the default components once the app rendered, and it will switch between routes as needed
* The <Switch /> component will only render the first route that matches/includes the path. Once it finds the first route that matches the path, it will not look for any other matches.
* Not only that, it allows for nested routes to work properly, which is something that <Router /> will not be able to handle.

ReactDOM.render((

<Switch>

<Route exact path="/" component={Home} />

<Route path="/login" component={Login} />

<Route path="/explore" component={Explore} />

</Switch>),

document.getElementById('root')

);

1. **What is "match" in react routing?**

A match object contains information about how a <Route path> matched the URL. match objects contain the following properties:

* params - (object) Key/value pairs parsed from the URL corresponding to the dynamic segments of the path
* isExact - (boolean) true if the entire URL was matched (no trailing characters)
* path - (string) The path pattern used to match. Useful for building nested <Route>s
* url - (string) The matched portion of the URL. Useful for building nested <Link>s

The match object contains information on how the <Route> path matches the current URL. It includes the url, path, isExact, and params properties.

<Route path="/user"

render={({ match }) => {

console.log(match);

return (

<div> Inside User route </div>

);

}}/>

1. **What is "prop drilling" in React JS?**

* Prop drilling is the process in a React app where props are passed from one part of a tree to another by going through other parts that do not need the data
* Prop drilling is basically a situation when the same data is being sent at almost every level due to requirements in the final level. but only help in passing it through the tree

import React, { useState } from "react";

function Parent() {

const [fName, setfName] = useState("firstName");

const [lName, setlName] = useState("LastName");

return (

<>

<div>This is a Parent component</div>

<br />

<ChildA fName={fName} lName={lName} />

</>

);

}

function ChildA({ fName, lName }) {

return (

<>

This is ChildA Component.

<br />

<ChildB fName={fName} lName={lName} />

</>

);

}

function ChildB({ fName, lName }) {

return (

<>

This is ChildB Component.

<br />

<ChildC fName={fName} lName={lName} />

</>

);

}

function ChildC({ fName, lName }) {

return (

<>

This is ChildC component.

<br />

<h3> Data from Parent component is as follows:</h3>

<h4>{fName}</h4>

<h4>{lName}</h4>

</>

);

}

export default Parent;

The problem with Prop Drilling is that whenever data from the Parent component will be needed, it would have to come from each level, Regardless of the fact that it is not needed there and simply needed in last.

A better alternative to this is using **useContext** hook.

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

let context = React.createContext(null);

function Parent() {

const [fName, setfName] = useState("firstName");

const [lName, setlName] = useState("LastName");

return (

<context.Provider value={{ fName, lName }}>

<div>This is a Parent component</div>

<br />

<ChildA />

</context.Provider>

);

}

function ChildA() {

return (

<>

This is ChildA Component.

<br />

<ChildB />

</>

);

}

function ChildB() {

return (

<>

This is ChildB Component.

<br />

<ChildC />

</>

);

}

function ChildC() {

const { fName, lName } = useContext(context);

return (

<>

This is ChildC component.

<br />

<h3> Data from Parent component is as follows:</h3>

<h4>{fName}</h4>

<h4>{lName}</h4>

</>

);

}

export default Parent;

1. **Benefits of using Redux in React as compared to local state?**

Redux allows the users to manage the state of the application in a single place and keep changes in the app more predictable and traceable

**Advantages of using Redux:**

**1. Centralized state management system i.e. Store:** React state is stored locally within a component. To share this state with other components in the application, props are passed to child components, or callbacks are used for parent components. Redux state, on the other hand, is stored globally in the store. All the components of the entire application can easily access the data directly. This centralizes all data and makes it very easy for a component to get the state it requires. So while developing large, complex applications with many components, the Redux store is highly preferred.

**2. Performance Optimizations:** By default, whenever a component is updated, React re-renders all the components inside that part of the component tree. In such a case when the data for a given component hasn’t changed, these re-renders are wasted (cause the UI output displayed on the screen would remain the same). Redux store helps in improving the performance by skipping such unnecessary re-renders and ensuring that a given component re-renders only when its data has actually changed.

**3. Pure reducer functions:** A pure function is defined as any function that doesn’t alter input data, doesn’t depend on the external state, and can consistently provide the same output for the same input. As opposed to React, Redux depends on such pure functions. It takes a given state (object) and passes it to each reducer in a loop. In case of any data changes, a new object is returned from the reducer (re-rendering takes place). However, the old object is returned if there are no changes (no re-rendering).

**4. Storing long-term data:** Since data stored in redux persists until page refresh, it is widely used to store long-term data that is required while the user navigates the application, such as, data loaded from an API, data submitted through a form, etc. On the other hand, React is suitable for storing short-term data that is likely to change quickly (form inputs, toggles, etc.)

**5. Time-travel Debugging:** In React, it becomes a tedious task to track the state of the application during the debugging process. Redux makes debugging the application an easy process. Since it represents the entire state of an application at any given point in time, it is widely used for time-travel debugging. It can even send complete error reports to the server!

**6. Great supportive community** Since redux has a large community of users, it becomes easier to learn about best practices, get help when stuck, reuse your knowledge across different applications. Also, there are a number of extensions for redux that help in simplifying the code logic and improving the performance.

1. **What is a promises in ES6?**

Promises are a way to implement async programming in JavaScript(ES6). A Promise will become a container for future value.

Like if you order any food on any site to deliver it to your place that order record will be the promise and the food will be the value of that promise.

JavaScript uses callbacks, among other things. However, callbacks suffer from problems like **Callback Hell/Pyramid of Doom. Promises** are a pattern that greatly simplifies asynchronous programming by making the code look synchronous and avoid problems associated with callbacks.

|  |
| --- |
| const myPromise = new Promise((resolve, reject) => {      if (Math.random() > 0) {          resolve('Hello, I am positive number!');      }      reject(new Error('I failed some times'));  }) |

<script>

function add\_positivenos\_async(n1, n2) {

let p = new Promise(function (resolve, reject) {

if (n1 >= 0 && n2 >= 0) {

//do some complex time consuming work

resolve(n1 + n2)

} else

reject('NOT\_Postive\_Number\_Passed')

})

return p;

}

//Promise.race(iterable)

Promise.race([add\_positivenos\_async(10,20),add\_positivenos\_async(30,40)])

.then(function(resolveValue){

console.log('one of them is done')

console.log(resolveValue)

}).catch(function(err){

console.log("Error",err)

})

console.log('end')

</script>

1. **What is the use of webpack?**

Webpack is a popular module bundling system built on top of Node. js. It can handle not only combination and minification of JavaScript and CSS files, but also other assets such as image files (spriting) through the use of plugins.

Webpack is a static module bundler for JavaScript applications. ... This enables you to take a fully dynamic application and package it into static files, which you can then upload and deploy to your server.

Webpack is a command line tool to create bundles of assets (code and files). Webpack doesn't run on the server or the browser. Webpack takes all your javascript files and any other assets and transforms then into one huge file. This big file can then be sent by the server to a client's browser.10

1. **What is the Babel in React JS?**

Babel is a very famous transpiler that basically allows us to use future JavaScript in today's browsers. In simple words, it can convert the latest version of JavaScript code into the one that the browser understands.

Babel is a JavaScript compiler that includes the ability to compile JSX into regular JavaScript.

1. **Why are callbacks required in JS?**

A callback is a function passed as an argument to another function.

Callbacks make sure that a function is not going to run before a task is completed but will run right after the task has completed. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors.

The benefit of using a callback function is that you can wait for the result of a previous function call and then execute another function call.

EX1:

const message = function() {

console.log("This message is shown after 3 seconds");

}

setTimeout(message, 3000);

EX2:

setTimeout(() => {

console.log("This message is shown after 3 seconds");

}, 3000);

Ex3:

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

function myCalculator(num1, num2, myCallback) {

let sum = num1 + num2;

myCallback(sum);

}

myCalculator(5, 5, myDisplayer);

Nesting many asynchronous functions inside callbacks is known as the **pyramid of doom** or **the callback hell:**

asyncFunction(function(){

asyncFunction(function(){

asyncFunction(function(){

asyncFunction(function(){

asyncFunction(function(){

....

});

});

});

});

});

1. **Some new features of ES6 which were not in JS?**

* The let keyword
* The const keyword
* Arrow Functions
* For/of
* Map Objects
* Set Objects
* Classes
* Promises
* Symbol
* Default Parameters
* Function Rest Parameter
* String.includes()
* String.startsWith()
* String.endsWith()
* Array.from()
* Array keys()
* Array find()
* Array findIndex()
* New Math Methods
* New Number Properties
* New Number Methods
* New Global Methods

1. Default Parameters in ES6
2. Template Literals in ES6
3. Multi-line Strings in ES6
4. Destructuring Assignment in ES6
5. Enhanced Object Literals in ES6
6. Arrow Functions in ES6
7. Promises in ES6
8. Block-Scoped Constructs Let and Const
9. Classes in ES6
10. Modules in ES6
11. **Difference between controlled and uncontrolled components?**

**A Controlled Component** is one that takes its current value through props and notifies changes through callbacks like onChange. A parent component "controls" it by handling the callback and managing its own state and passing the new values as props to the controlled component. You could also call this a "dumb component".

**A Uncontrolled Component** is one that stores its own state internally, and you query the DOM using a ref to find its current value when you need it. This is a bit more like traditional HTML.

// Controlled:

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

// Uncontrolled:

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

// Use `inputRef.current.value` to read the current value of <input>

|  |  |  |
| --- | --- | --- |
| **SN** | **Controlled** | **Uncontrolled** |
| **1.** | It does not maintain its internal state. | It maintains its internal states. |
| **2.** | Here, data is controlled by the parent component. | Here, data is controlled by the DOM itself. |
| **3.** | It accepts its current value as a prop. | It uses a ref for their current values. |
| **4.** | It allows validation control. | It does not allow validation control. |
| **5.** | It has better control over the form elements and data. | It has limited control over the form elements and data. |

1. **How to configure nested routing in React JS?**

**Ex1**

import React from "react";

import {

BrowserRouter as Router,

Switch,

Route,

Link,

useParams,

useRouteMatch

} from "react-router-dom";

export default function NestingExample() {

return (

<Router>

<div>

<ul>

<li>

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

</li>

<li>

<Link to="/topics">Topics</Link>

</li>

</ul>

<hr />

<Switch>

<Route exact path="/">

<Home />

</Route>

<Route path="/topics">

<Topics />

</Route>

</Switch>

</div>

</Router>

);

}

function Home() {

return (

<div>

<h2>Home</h2>

</div>

);

}

function Topics() {

// The `path` lets us build <Route> paths that are

// relative to the parent route, while the `url` lets

// us build relative links.

let { path, url } = useRouteMatch();

return (

<div>

<h2>Topics</h2>

<ul>

<li>

<Link to={`${url}/rendering`}>Rendering with React</Link>

</li>

<li>

<Link to={`${url}/components`}>Components</Link>

</li>

<li>

<Link to={`${url}/props-v-state`}>Props v. State</Link>

</li>

</ul>

<Switch>

<Route exact path={path}>

<h3>Please select a topic.</h3>

</Route>

<Route path={`${path}/:topicId`}>

<Topic />

</Route>

</Switch>

</div>

);

}

function Topic() {

// The <Route> that rendered this component has a

// path of `/topics/:topicId`. The `:topicId` portion

// of the URL indicates a placeholder that we can

// get from `useParams()`.

let { topicId } = useParams();

return (

<div>

<h3>{topicId}</h3>

</div>

);

}

1. **React JS Component lifecycle method order.**

**Mounting**

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

* **constructor**()

method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

* **static getDerivedStateFromProps**()

method is called right before rendering the element(s) in the DOM.

This is the natural place to set the state object based on the initial props.

* **render**()

method is required, and is the method that actually outputs the HTML to the DOM.

* **componentDidMount**()

method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

static getDerivedStateFromProps(props, state) {

return {favoritecolor: props.favcol };

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

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

**Updating**

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

* **static getDerivedStateFromProps()**

This is the first method that is called when a component gets updated.

* **shouldComponentUpdate()**

can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is true.

* **render**()

re-render the HTML to the DOM, with the new changes.

* **getSnapshotBeforeUpdate**()

method you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.

If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

* **componentDidUpdate**()

method is called after the component is updated in the DOM.

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

getSnapshotBeforeUpdate(prevProps, prevState) {

document.getElementById("div1").innerHTML =

"Before the update, the favorite was " + prevState.favoritecolor;

}

componentDidUpdate() {

document.getElementById("div2").innerHTML =

"The updated favorite is " + this.state.favoritecolor;

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<div id="div1"></div>

<div id="div2"></div>

</div>

);

}

}

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

**Unmounting**

This method is called when a component is being removed from the DOM:

* **componentWillUnmount**()

method is called when the component is about to be removed from the DOM.

lass Child extends React.Component {

componentWillUnmount() {

alert("The component named Header is about to be unmounted.");

}

render() {

return (

<h1>Hello World!</h1>

);

}

}

1. **What is Virtual DOM in React?**

**Virtual DOM:** React uses Virtual DOM exists which is like a lightweight copy of the actual DOM(a virtual representation of the DOM). So for every object that exists in the original DOM, there is an object for that in React Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document. Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen. So each time there is a change in the state of our application, virtual DOM gets updated first instead of the real DOM.

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

**How Virtual DOM helps React:** In react, everything is treated as a component be it a functional component or class component. A component can contain a state. Each time we change something in our JSX file or let’s put it in simple terms, whenever the state of any component is changed react updates it’s Virtual DOM tree

React maintains two Virtual DOM at each time, one contains the updated Virtual DOM and one which is just the pre-update version of this updated Virtual DOM. Now it compares the pre-update version with the updated Virtual DOM and figures out what exactly has changed in the DOM like which components have been changed. This process of comparing the current Virtual DOM tree with the previous one is known as **‘diffing’.**

Once React finds out what exactly has changed then it updated those objects only, on real DOM. React uses something called as batch updates to update the real DOM. It just mean that the changes to the real DOM are sent in batches instead of sending any update for a single change in the state of a component. We have seen that the re-rendering of the UI is the most expensive part and React manages to do this most efficiently by ensuring that the Real DOM receives batch updates to re-render the UI. This entire proces of transforming changes to the real DOM is called **Reconciliation**

1. **Parent comp passes a number via props to a child comp. Child must calculate the cube and return back to parent comp. In React how to do this?**

import React from 'react';

class Parent extends React.Component{

constructor(props){

super(props);

this.state = {

data: null

}

}

handleCallback = (childData) =>{

this.setState({data: childData})

}

render(){

const {data} = this.state;

return(

<div>

<Child parentCallback = {this.handleCallback}/>

{data}

</div>

)

}

}

class Child extends React.Component{

onTrigger = (event) => {

this.props.parentCallback("Data from child");

event.preventDefault();

}

render(){

return(

<div>

<form onSubmit = {this.onTrigger}>

<input type = "submit" value = "Submit"/>

</form>

</div>

)

}

}

export default Parent;

**Using Hooks :**

import { useCallback, useState } from "react";

const Counter = ({ parentCallback }) => {

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

return (

<button

onClick={() => {

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

parentCallback(count + 1);

}}

>

increment

</button>

);

};

export default function App() {

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

const callback = useCallback((count) => {

setCount(count);

}, []);

return (

<div className="App">

<Counter parentCallback={callback} />

<h2>count {count}</h2>

</div>

);

}

1. **Use of enzyme in React JS Unit Testing?**

Enzyme provides a mechanism to mount and traverse React.js component trees. This will help us get access to its own properties and state as well as its children props in order to run our assertions.

Enzyme offers two basic functions for component mounting: **shallow and mount**. The **shallow function loads in memory only the root component** whereas **mount loads the full DOM tree.**

**Shallow:**

import React from 'react';

import { expect } from 'chai';

import { shallow } from 'enzyme';

import sinon from 'sinon';

import MyComponent from './MyComponent';

import Foo from './Foo';

describe('<MyComponent />', () => {

it('renders three <Foo /> components', () => {

const wrapper = shallow(<MyComponent />);

expect(wrapper.find(Foo)).to.have.lengthOf(3);

});

it('renders an `.icon-star`', () => {

const wrapper = shallow(<MyComponent />);

expect(wrapper.find('.icon-star')).to.have.lengthOf(1);

});

it('renders children when passed in', () => {

const wrapper = shallow((

<MyComponent>

<div className="unique" />

</MyComponent>

));

expect(wrapper.contains(<div className="unique" />)).to.equal(true);

});

**Mount:**

import React from 'react';

import sinon from 'sinon';

import { expect } from 'chai';

import { mount } from 'enzyme';

import Foo from './Foo';

describe('<Foo />', () => {

it('allows us to set props', () => {

const wrapper = mount(<Foo bar="baz" />);

expect(wrapper.props().bar).to.equal('baz');

wrapper.setProps({ bar: 'foo' });

expect(wrapper.props().bar).to.equal('foo');

});

it('simulates click events', () => {

const onButtonClick = sinon.spy();

const wrapper = mount((

<Foo onButtonClick={onButtonClick} />

));

wrapper.find('button').simulate('click');

expect(onButtonClick).to.have.property('callCount', 1);

});

1. **What is the use of the "public" folder in a react js app?**

The public folder contains static files such as index.html, javascript library files, images, and other assets, etc. which you don’t want to be processed by webpack. Files in this folder are copied and pasted as they are directly into the build folder. Only files inside the `public` folder can be referenced from the HTML.

1. **What is the use of "package.json" in a react app?**

In order to add dependency packages to your project, you need to create package. json file. This file is used to store the metadata associated with that project as well as to store the list of dependency packages

This file keeps track of all the different kinds of dependencies you are using in your project in JSON format.

1. **Difference between Link and NavLink in react routing?**

The Link component is used to navigate the different routes on the site. But NavLink is used to add the style attributes to the active routes.

Well actually, the main difference between these two's is a class attribute. When we use the NavLink as a tag, it automatically inherit an active class when clicked. On the other hand, the Link tag does now have an active class when clicked.

**When should I use the NavLink?**

Just as the name implies 'NavLink', we use it mostly on navigation bars. This is because the active class permits us to define our custom styling in the App.css stylesheet. As such, we can use it to style our active buttons which in notify the use on which page he/she is currently on.

**When should I use the Link?**

The Link tag can be used where we want to do just some routing with no special effect. For instance; we can use the Link tag for scroll-to-top button, add to card buttons, submit button and more.

import '../App.css';

import {NavLink} from 'react-router-dom';

let Nav =()=>{

return (

<div>

<nav>

<div className ='logo'>

<p>Logo</p>

</div>

<div>

<ul>

<li><NavLink to = '/home'>Home</NavLink></li>

<li><NavLink to = '/about'>About</NavLink></li>

<li><NavLink to = '/contact'>Contact</NavLink></li>

</ul>

</div>

</nav>

</div>

)

} export default Nav

1. **What is the use of snapshot testing in React JS?**

Snapshot tests are a very useful tool whenever you want to make sure your UI does not change unexpectedly. A typical snapshot test case renders a UI component, takes a snapshot, then compares it to a reference snapshot file stored alongside the test.

Snapshot testing allows you to ensure your output continues to behave as expected. This is useful because as you revisit your code to make updates over time, there is an increased likelihood that those changes may cause something to break.

import React from 'react';

import renderer from 'react-test-renderer';

import Items from './Items';

it('renders correctly when there are no items', () => {

const tree = renderer.create(<Items />).toJSON();

expect(tree).toMatchSnapshot();

});

1. **What is the use of "describe" in test?**

describe breaks your test suite into components. Depending on your test strategy, you might have a describe for each function in your class, each module of your plugin, or each user-facing piece of functionality.

**describe(name, fn)** creates a block that groups together several related tests.

You shouldn't be able to subdivide tests further-- if you feel like you need to, use describe instead.

describe('Circle class', function() {

describe('area is calculated when', function() {

it('sets the radius', function() { ... });

it('sets the diameter', function() { ... });

it('sets the circumference', function() { ... });

});

});

1. **How to get code coverage in JEST unit testing?**

shows you the percent coverage for the following code stats by file:

* Statements
* Branches
* Functions
* Lines

**npx jest --coverage**

In order to generate this report, all you have to do is run **npm run coverage** or **npm run test:coverage**

1. **How to define global styles in React JS?**

**Method 1: Custom Components**

Since React is component-driven, the most intuitive way is to create custom components (e.g. custom text fields, custom buttons, etc.), define the styles within each component, and reuse these components throughout the app.

**Method 2: Global Stylesheet**

Create a single stylesheet (e.g. in the constants folder) and manage all styles from inside there.

**Method 3: createGlobalStyle from styled-components**

Here, we’ll use the **createGlobalStyle** function from **styled-components** and add some global styles:

// globalStyles.js

import { createGlobalStyle } from 'styled-components';

const GlobalStyle = createGlobalStyle`

body {

margin: 0;

padding: 0;

background: teal;

font-family: Open-Sans, Helvetica, Sans-Serif;

}

`;

export default GlobalStyle;

1. **What is mapDispatchToProps in redux?**

mapDispatchToProps is used for dispatching actions to the store. **mapStateToProps** is a function that you would use to provide the store data to your component, whereas **mapDispatchToProps** is something that you will use to provide the action creators as props to your component.

With mapDispatchToProps every action creator wrapped into a dispatch call so they may be invoked directly, will be merged into the component’s props.

function mapStateToProps(state) {

return { todos: state.todos }

}

function mapDispatchToProps(dispatch) {

return {  
 // dispatching plain actions  
 increment: () => dispatch({ type: 'INCREMENT' }),  
 decrement: () => dispatch({ type: 'DECREMENT' }),  
 reset: () => dispatch({ type: 'RESET' }),  
 }

}

export default connect(mapStateToProps, mapDispatchToProps)(Todos);

1. **Talk about ES6 promises**

Promises are a way to implement async programming in JavaScript(ES6). A Promise will become a container for future value.

Like if you order any food on any site to deliver it to your place that order record will be the promise and the food will be the value of that promise.

JavaScript uses callbacks, among other things. However, callbacks suffer from problems like **Callback Hell/Pyramid of Doom. Promises** are a pattern that greatly simplifies asynchronous programming by making the code look synchronous and avoid problems associated with callbacks.

function readFilePromise() {

    return new Promise(function(resolve, reject) {

        fs.readFile('config.json',

            function(error, text) {

                if (error) {

                    reject('Error while reading file')

                } else {

                    resolve(text)

                }

            });

    })

}

//multiple then block can have single catch block

readFilePromise()

    .then(function(res) {

        console.log('Res ', JSON.parse(res))

        return promise

    })

    .then()

    .then()

    .catch(function(error) {

        console.log('error ', error)

    })

|  |
| --- |
| const myPromise = new Promise((resolve, reject) => {      if (Math.random() > 0) {          resolve('Hello, I am positive number!');      }      reject(new Error('I failed some times'));  }) |

<script>

function add\_positivenos\_async(n1, n2) {

let p = new Promise(function (resolve, reject) {

if (n1 >= 0 && n2 >= 0) {

//do some complex time consuming work

resolve(n1 + n2)

} else

reject('NOT\_Postive\_Number\_Passed')

})

return p;

}

//Promise.race(iterable)

Promise.race([add\_positivenos\_async(10,20),add\_positivenos\_async(30,40)])

.then(function(resolveValue){

console.log('one of them is done')

console.log(resolveValue)

}).catch(function(err){

console.log("Error",err)

})

console.log('end')

</script>

1. **Differences between then() and catch() in Promises. When to use what?**

**Callbacks to Promises**: There are two types of callbacks which are used for handling promises .then() and .catch(). It can be used for handling promises in case of fulfillment (promise is kept) or rejection (promise is broken).

**.then():** Invoked when a promise is kept or broken. It can be chained to handle the fulfillment or rejection of a promise. It takes in two functions as parameters. The first one is invoked if the promise is fulfilled and the second one(optional) is invoked if the promise is rejected.

**v**ar promise = new Promise(function(resolve, reject) {

resolve('Hello, I am a Promise!');

})

promise.then(function(promise\_kept\_message) {

console.log(promise\_kept\_message);

}, function(error) {

// This function is invoked this time

// as the Promise is rejected.

console.log(error); })

**.catch()** can be used for handling the errors(if any). It takes only one function as a parameter which is used to handle the errors (if any).

const myPromise = new Promise((resolve, reject) => {

if (Math.random() > 0) {

console.log('resolving the promise ...');

resolve('Hello, Positive :)');

}

reject(new Error('No place for Negative here :('));

});

const Fulfilled = (fulfilledValue) => console.log(fulfilledValue);

const Rejected = (error) => console.log(error);

myPromise.then(Fulfilled, Rejected);

myPromise.then((fulfilledValue) => {

console.log(fulfilledValue);

}}).catch(err => console.log(err));

The main difference between the forms promise.then(success, error) and promise.then(success).catch(error) is that in case if success callback returns a rejected promise, then only the second form is going to catch that rejection.

function funA(input, ms = 500, shouldReject = false) {

    return new Promise((resolve, reject) => {

        setTimeout(() => {

            if (shouldReject) reject(`Error! [funA] Promise Rejected`)

            resolve(input)

        }, ms);

    })

}

function funB(input, ms = 500, shouldReject = false) {

    return new Promise((resolve, reject) => {

        setTimeout(() => {

            if (shouldReject) reject(`Error! [funA] Promise Rejected`)

            resolve(input)

        }, ms);

    })

}

function funC(input, ms = 500, shouldReject = false) {

    return new Promise((resolve, reject) => {

        setTimeout(() => {

            if (shouldReject) reject(`Error! [funC] Promise Rejected`)

            resolve(input)

        }, ms);

    })

}

funA('john')

    .then((resA) => {

        console.log(resA)

        return funB(`${resA} funB variation`)

    }).then((resB) => {

        console.log(resB)

        return funC(`${resB} funC variation`)

    }).then((resC) => {

        console.log(resC)

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

//Promise all

Promise.all([funA('Ram'), funB('Jane', 1000, false), funC('Jany')])

    .then(([res1, res2, res3]) => {

        console.log(res1, res2, res3)

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

    //Ram Jane Jany

Promise.allSettled([funA('Ram'), funB('Jane', 1000, false), funC('Jany')])

    .then((arr) => {

        console.log(arr)

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

    // [

    //     { status: 'fulfilled', value: 'Ram' },

    //     { status: 'fulfilled', value: 'Jane' },

    //     { status: 'fulfilled', value: 'Jany' }

    // ]

1. **What are pure components?**

Pure Components in React are the components which do not re-renders when the value of state and props has been updated with the same values. If the value of the previous state or props and the new state or props is the same, the component is not re-rendered.

ReactJS has provided us a Pure Component. If we extend a class with Pure Component, there is no need for **shouldComponentUpdate**() Lifecycle Method. ReactJS Pure Component Class compares current state and props with new props and states to decide whether the React component should re-render itself or Not.

In simple words, If the previous value of state or props and the new value of state or props is the same, the component will not re-render itself. Since Pure Components restricts the re-rendering when there is no use of re-rendering of the component.

import React from ‘react’;

export default class Test extends React.PureComponent{

render(){

return <h1>Welcome to GeeksforGeeks</h1>;

}

}

1. **What are hooks in React? useDispatch & useSelector hooks?**

Hooks are a new addition in React 16.8. They let developers 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

**useState() hook,** which allows functional components to have a dedicated state of their own,

import React, { useState } from 'react';

function App() {

const [click, setClick] = useState(0);

// using array destructuring here

// to assign initial value 0

// to click and a reference to the function

// that updates click to setClick

return (

<div>

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

<button onClick={() => setClick(click + 1)}>

Click me

</button>

</div>

);

}

export default App;

and the **useEffect() hook**, which allows functional components to manipulate DOM elements before each render (almost like one gets to do it in lifecycle functions).

**useEffect Hook** as **componentDidMount, componentDidUpdate**, and **componentWillUnmount** combined.

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

function HookCounterOne() {

const [count, setCount] = useState(0)

useEffect(() => {

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

return (

<div>

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

Click {count} times </button>

</div>

)

}

export default HookCounterOne

1. **What are the keys in react? What is the use of it?**

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

import React from 'react';

import ReactDOM from 'react-dom';

// Component to be extracted

function MenuItems(props)

{

const item = props.item;

return(

<li>

{item}

</li>

);

}

// Component that will return an

// unordered list

function Navmenu(props)

{

const list = props.menuitems;

const updatedList = list.map((listItems)=>{

return (

);

});

return(

<ul>{updatedList}</ul>);

}

const menuItems = [1, 2, 3, 4, 5];

ReactDOM.render(

,

document.getElementById('root')

);

1. **What is Higher order components?Explain the use case where we can make use of HOC?**

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React’s compositional nature.

Concretely, a higher-order component is a function that takes a component and returns a new component.

// HOC.js

import React, {Component} from 'react';

export default function Hoc(HocComponent, data){

return class extends Component{

constructor(props) {

super(props);

this.state = {

data: data

};

}

render(){

return (

<HocComponent data={this.state.data} {...this.props} />

);

}

}

}

// UserList.js

import React, { Component } from 'react';

import TableRow from './TableRow';

class UserList extends Component {

constructor(props) {

super(props);

}

tabRow(){

if(this.props.data instanceof Array){

return this.props.data.map(function(object, i){

return <TableRow obj={object} key={i} />;

})

}

}

render() {

return (

<div className="container">

<table className="table table-striped">

<thead>

<tr>

<td>ID</td>

<td>Name</td>

</tr>

</thead>

<tbody>

{this.tabRow()}

</tbody>

</table>

</div>

);

}

}

export default UserList;

// App.js

import React, { Component } from 'react';

import StockList from './StockList';

import UserList from './UserList';

import Hoc from './HOC';

const StocksData = [

{

id: 1,

name: 'TCS'

},

{

id: 2,

name: 'Infosys'

},

{

id: 3,

name: 'Reliance'

}

];

const UsersData = [

{

id: 1,

name: 'Krunal'

},

{

id: 2,

name: 'Ankit'

},

{

id: 3,

name: 'Rushabh'

}

];

const Stocks = Hoc(

StockList,

StocksData

);

const Users = Hoc(

UserList,

UsersData

);

class App extends Component {

render() {

return (

<div>

<Users></Users>

</div>

)

}

}

export default App;

1. **What is the purpose of switch,link,BrowserRouter component in the React Router DOM?**
2. **Explain mapSateToProps() in redux?**

mapDispatchToProps is used for dispatching actions to the store. **mapStateToProps** is a function that you would use to provide the store data to your component, whereas **mapDispatchToProps** is something that you will use to provide the action creators as props to your component.

With mapDispatchToProps every action creator wrapped into a dispatch call so they may be invoked directly, will be merged into the component’s props.

function mapStateToProps(state) {

return { todos: state.todos }

}

function mapDispatchToProps(dispatch) {

return {  
 // dispatching plain actions  
 increment: () => dispatch({ type: 'INCREMENT' }),  
 decrement: () => dispatch({ type: 'DECREMENT' }),  
 reset: () => dispatch({ type: 'RESET' }),  
 }

}

export default connect(mapStateToProps, mapDispatchToProps)(Todos);

1. **error boundaries in react?**
2. **StateFul and StateLess Components?**
3. **how to update state's - By using setState method**

***setState({ stateName : updatedStateValue })***

***// OR***

***setState((prevState) => ({***

***stateName: prevState.stateName + 1***

***}))***

class App extends Component {

constructor(props){

super(props)

// Set initial state

this.state = {greeting :

'Click the button to receive greetings'}

// Binding this keyword

this.updateState = this.updateState.bind(this)

}

updateState(){

// Changing state

this.setState({greeting :

'GeeksForGeeks welcomes you !!'})

}

1. **why we can't update state's directly?**

NEVER mutate this.state directly, as calling setState() afterwards may replace the mutation you made. Treat this.state as if it were immutable.

setState() does not immediately mutate this.state but creates a pending state transition. Accessing this.state after calling this method can potentially return the existing value.

There is no guarantee of synchronous operation of calls to setState and calls may be batched for performance gains.

setState() will always trigger a re-render unless conditional rendering logic is implemented in shouldComponentUpdate(). If mutable objects are being used and the logic cannot be implemented in shouldComponentUpdate(), calling setState() only when the new state differs from the previous state will avoid unnecessary re-renders.

Basically, if you modify this.state directly, you create a situation where those modifications might get overwritten.

1. **Synchronous and Asynchronous?**

**Synchronous JavaScript:** As the name suggests synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.

<script>

document.write("Hi"); // First

document.write("<br>");

document.write("Mayukh") ;// Second

document.write("<br>");

document.write("How are you"); // Third

</script>

**Asynchronous JavaScript:** Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one.

<script>

document.write("Hi");

document.write("<br>");

setTimeout(() => {

document.write("Let us see what happens");

}, 2000);

document.write("<br>");

document.write("End");

document.write("<br>");

</script>

1. **how to prevent rerendering in react?**

**shouldComponentUpdate(nextProps, nextState)**

import React, { Component } from "react";

class Counter1 extends Component {

shouldComponentUpdate(nextProps) {

// Rendering the component only if

// passed props value is changed

if (nextProps.value !== this.props.value) {

return true;

} else {

return false;

}

}

render() {

return (

<div>

<h2>Counter 1:</h2>

<h3>{this.props.value}</h3>

<button onClick={this.props.onClick}>Add</button>

</div>

);

}

}

export default Counter1;

1. **Why do we use arrow function in react?**

**Arrows prevent this bugs :**Arrow functions don’t redefine the value of this within their function body. This makes it a lot easier to predict their behavior when passed as callbacks, and prevents bugs caused by use of this within callbacks.

With a standard function expression or declaration in Javascript, the definition of 'this' is dependent on where the function was called.

As a result in order to ensure that 'this' is acting on the correct object or class you may need to use the bind method to ensure the 'this' keyword maintains the correct reference.

it doesn’t change the context of this keyword, so you don’t need to bind at all:

handleClick = () => {

this.setState({name: "Jack"})

}

render(){

return (

<div>

<p>My name is {this.state.name}</p>

<button onClick={this.handleClick}>

Change name

</button>

</div>

1. **When to use component vs element?**

**React Elements**

* A React Element is what gets returned from components. It’s an object that virtually describes the DOM nodes that a component represents.
* With a function component, this element is the object that the function returns.
* With a class component, the element is the object that the component’s render function returns. R
* React elements are not what we see in the browser. They are just objects in memory and we can’t change anything about them.
* React elements can have other type properties other than native HTML elements.

const element = React.createElement(

'div',

{id: 'login-btn'},

'Login'

)

**React Components**

* A component is a function or a Class which optionally accepts input and returns a React element.
* A React Component is a template. A blueprint. A global definition. This can be either a function or a class (with a render function).
* If react sees a class or a function as the first argument, it will check to see what element it renders, given the corresponding props and will continue to do this until there are no more createElement invocations which have a class or a function as their first argument.
* When React sees an element with a function or class type, it will consult with that component to know which element it should return, given the corresponding props.

1. **Context API?**

The React Context API is a way for a React app to effectively produce global variables that can be passed around. This is the alternative to "prop drilling" or moving props from grandparent to child to parent, and so on. Context is also touted as an easier, lighter approach to state management using Redux.

Context provides a way to pass data through the component tree without having to pass props down manually at every level

// Context lets us pass a value deep into the component tree// without explicitly threading it through every component.// Create a context for the current theme (with "light" as the default).

const ThemeContext = React.createContext('light');

class App extends React.Component {

render() {

// Use a Provider to pass the current theme to the tree below. // Any component can read it, no matter how deep it is. // In this example, we're passing "dark" as the current value. return (

<ThemeContext.Provider value="dark"> <Toolbar />

</ThemeContext.Provider>

);

}

}

// A component in the middle doesn't have to// pass the theme down explicitly anymore.function Toolbar() {

return (

<div>

<ThemedButton />

</div>

);

}

class ThemedButton extends React.Component {

// Assign a contextType to read the current theme context. // React will find the closest theme Provider above and use its value. // In this example, the current theme is "dark".

static contextType = ThemeContext;

render() {

return <Button theme={this.context} />; }

}

1. **Async Await?**

The purpose of async / await is to simplify the syntax necessary to consume promise-based APIs. The behavior of async / await is similar to combining generators and promises. Async functions always return a promise

async function hello() {

return await Promise.resolve("Hello");

};

hello().then(alert);

1. **What is let var const?**

**Var**

* Scope essentially means where these variables are available for use. var declarations are globally scoped or function/locally scoped.
* The scope is global when a var variable is declared outside a function. This means that any variable that is declared with var outside a function block is available for use in the whole window.
* var is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.
* **var variables can be re-declared and updated :** This means that we can do this within the same scope and won't get an error.
* **Hoisting of var :** Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

**Let**

* **let is block scoped :** A block is a chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block.
* **let can be updated but not re-declared :** Just like var, a variable declared with let can be updated within its scope. Unlike var, a let variable cannot be re-declared within its scope
* **Hoisting of let :** if you try to use a let variable before declaration, you'll get a Reference Error.

**Const**

* Variables declared with the const maintain constant values.
* **const cannot be updated or re-declared :** This means that the value of a variable declared with const remains the same within its scope. It cannot be updated or re-declared
* while we cannot do this:

const greeting = {

message: "say Hi",

times: 4

}

greeting = {

words: "Hello",

number: "five"

} // error: Assignment to constant variable.

* we can do this:

greeting.message = "say Hello instead";