

React Basics

1. What is ReactJS?

ReactJS is a JavaScript library for building user interfaces, primarily developed and maintained by Facebook. It allows developers to create large web applications that can update and render efficiently in response to data changes.

javascript
Copy code

```
// Example of a simple React component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

2. Explain the key features of React.

- **Component-Based:** Build encapsulated components that manage their own state.
- **Declarative:** Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.
- **Virtual DOM:** React creates a virtual DOM to improve performance.

Unidirectional Data Flow: Data flows in one direction, making it easier to debug.

javascript
Copy code

```
// Example: Component-based and declarative
class App extends React.Component {
  render() {
    return <h1>Hello, world!</h1>;
  }
}
```

3. What are the differences between React and Angular?

- **Library vs Framework:** React is a library, while Angular is a full-fledged framework.
- **Data Binding:** React uses one-way data binding, Angular uses two-way data binding.
- **DOM:** React uses a virtual DOM, Angular manipulates the real DOM directly.
- **Learning Curve:** React has a simpler learning curve compared to Angular.

4. Explain the virtual DOM in React.

The virtual DOM is a lightweight copy of the real DOM. React keeps a virtual representation of the UI in memory, which it synchronizes with the real DOM using a process called reconciliation.

javascript
Copy code

```
// Example: Updating the virtual DOM
const element = <h1>Hello, world</h1>;
ReactDOM.render(element,
  document.getElementById('root'));
```

5. How does React achieve better performance with the virtual DOM?

- React achieves better performance by using the virtual DOM to batch updates and perform efficient diffing and reconciliation, minimizing direct manipulations to the real DOM.

6. What is JSX? Why is it used in React?

JSX is a syntax extension for JavaScript that looks similar to XML or HTML. It is used in React to describe what the UI should look like.

javascript
Copy code

```
// Example: JSX syntax
const element = <h1>Hello, world!</h1>;
```

7. How do you create components in React?

Components in React can be created using functions or classes.

javascript
Copy code

```
// Functional component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

```
// Class component
class Welcome extends React.Component {
  render() {
    return <h1>Hello,
{this.props.name}</h1>;
  }
}
```

8. What are props in React?

Props are inputs to a React component. They are passed to the component similarly to function arguments.

javascript
Copy code

```
// Example: Using props
function Greeting(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

9. Explain the concept of state in React.

State is a built-in object that holds property values that belong to the component. When the state object changes, the component re-renders.

javascript
Copy code

```
// Example: Using state in a class component
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }

  render() {
    return <h1>It is
{this.state.date.toLocaleTimeString()}</h1>;
  }
}
```

10. What is the difference between state and props?

- **Props:** Passed to the component (similar to function parameters), immutable.
- **State:** Managed within the component (similar to variables declared inside a function), mutable.

Components and Props

How do you create functional components in React?

javascript

Copy code

```
// Example: Functional component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

11.

How do you create class components in React?

javascript
Copy code

```
// Example: Class component
class Welcome extends React.Component {
  render() {
    return <h1>Hello,
    {this.props.name}</h1>;
  }
}
```

12.

13. What are pure components in React?

Pure components are components that only re-render when their props or state change. They perform a shallow comparison of props and state to determine if re-rendering is necessary.

javascript
Copy code

```
// Example: PureComponent
class MyComponent extends React.PureComponent {
  render() {
    return <h1>Hello,
    {this.props.name}</h1>;
  }
}
```

14. What are higher-order components (HOC) in React?

HOCs are functions that take a component and return a new component with added functionality.

javascript
Copy code

```
// Example: Higher-order component
function withGreeting(WrappedComponent) {
  return class extends React.Component {
    render() {
      return <WrappedComponent
      greeting="Hello" {...this.props} />;
    }
  };
}
```

How do you pass props to components in React?

javascript
Copy code

```
// Example: Passing props
function App() {
  return <Welcome name="Sara" />;
}

function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

15.

16. How do you handle props and state changes in React components?

- **Props:** Handled by the parent component and passed down to the child component.

State: Managed within the component, changes are handled with `setState` or `useState` hook.

javascript
Copy code

```
// Example: Handling state changes
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

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

  render() {
    return (
      <div>
        <p>Count:
        {this.state.count}</p>
        <button
        onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

17. Explain the lifecycle methods of React components.

- Lifecycle methods allow you to hook into certain points of a component's life.
 - **Mounting:** `constructor()`, `componentDidMount()`
 - **Updating:** `shouldComponentUpdate()`, `componentDidUpdate()`
 - **Unmounting:** `componentWillUnmount()`

javascript
Copy code

```
// Example: Lifecycle methods
class MyComponent extends React.Component {
  componentDidMount() {
    console.log('Component mounted');
  }

  componentDidUpdate() {
    console.log('Component updated');
  }

  componentWillUnmount() {
    console.log('Component will
    unmount');
  }

  render() {
    return <div>Hello</div>;
  }
}
```

18. What is the significance of `shouldComponentUpdate` method?

`shouldComponentUpdate` allows you to optimize performance by preventing unnecessary re-renders. It returns `true` by default, but you can override it to return `false` if the component does not need to update.

javascript
Copy code

```
// Example: shouldComponentUpdate
class MyComponent extends React.Component {
  shouldComponentUpdate(nextProps,
    nextState) {
    return nextProps.someValue !==
    this.props.someValue;
  }

  render() {
    return
    <div>{this.props.someValue}</div>;
  }
}
```

○

How do you implement default props in a React component?

javascript
Copy code

```
// Example: Default props
class Greeting extends React.Component {
  static defaultProps = {
    name: 'Guest'
  };

  render() {
    return <h1>Hello,
    {this.props.name}</h1>;
  }
}
```

19.

20. What are controlled and uncontrolled components in React?

Controlled Components: Form inputs whose values are controlled by React state.

javascript
Copy code

```
// Example: Controlled component
class ControlledInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: '' };
  }

  handleChange = (event) => {
    this.setState({ value:
    event.target.value });
  }

  render() {
    return <input
    value={this.state.value}
    onChange={this.handleChange} />;
  }
}
```

○

Uncontrolled Components: Form inputs that maintain their own state.

javascript

Copy code

```
// Example: Uncontrolled component
class UncontrolledInput extends
React.Component {
  constructor(props) {
    super(props);
    this.inputRef = React.createRef();
  }

  handleSubmit = (event) => {
    event.preventDefault();
    alert('Input value: ' +
    this.inputRef.current.value);
  }

  render() {
    return (
      <form
      onSubmit={this.handleSubmit}>
        <input type="text"
        ref={this.inputRef} />
        <button
        type="submit">Submit</button>
      </form>
    );
  }
}
```

○

State Management

21. What are the different ways to manage state in React?

- **Local state:** Managed within a component using `useState` or `setState`.
- **Global state:** Managed across multiple components using Context API or libraries like Redux.
- **Derived state:** State derived from props or other state values.
- **URL state:** Managed through React Router for route parameters and query strings.

22. Explain the `useState` hook in React.

`useState` is a hook that allows you to add state to functional components.

javascript
Copy code

```
// Example: useState hook
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() =>
        setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

○

23. What is the `useEffect` hook used for in React?

`useEffect` is a hook for performing side effects in functional components. It runs after the component renders.

javascript
Copy code

```
// Example: useEffect hook
```

```
function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => {
      setCount(prevCount => prevCount +
1);
    }, 1000);

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

  return <div>Timer: {count}</div>;
}
```

24. How do you manage global state in React applications?

Use Context API or state management libraries like Redux or MobX.

javascript
Copy code

```
// Example: Using Context API for global
state
const AppContext = React.createContext();
```

```
function AppProvider({ children }) {
  const [state, setState] = useState({
    user: null });

  return (
    <AppContext.Provider value={[state,
setState]}>
      {children}
    </AppContext.Provider>
  );
}
```

```
function Component() {
  const [state, setState] =
useContext(AppContext);
  return <div>User: {state.user}</div>;
}
```

25. What are the differences between Redux and Context API for state management?

- **Redux:** Centralized store, more boilerplate, powerful debugging tools.

Context API: Simpler, less boilerplate, no middleware, better for smaller applications.

javascript
Copy code

```
// Redux example
import { createStore } from 'redux';

const initialState = { count: 0 };

function reducer(state = initialState,
action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1
};
    default:
      return state;
  }
}
```

```
const store = createStore(reducer);
```

26. Explain the Redux architecture and its core principles.

- **Single Source of Truth:** The state is stored in a single, centralized store.
- **State is Read-Only:** The state can only be changed by emitting actions.

Changes are Made with Pure Functions: Reducers specify how the state changes in response to actions.

javascript
Copy code

```
// Example: Redux store, actions, and
reducers
const initialState = { count: 0 };
```

```
function reducer(state = initialState,
action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1
};
    default:
      return state;
  }
}
```

```
const store = createStore(reducer);
```

```
store.dispatch({ type: 'INCREMENT' });
```

27. How do you connect Redux with a React application?

Use the **Provider** component from **react-redux** to wrap your app and the **connect** function or **useSelector** and **useDispatch** hooks to connect components to the Redux store.

javascript
Copy code

```
// Example: Connecting Redux with React
import { Provider, connect } from
'react-redux';
```

```
function mapStateToProps(state) {
  return { count: state.count };
}
```

```
function mapDispatchToProps(dispatch) {
  return {
    increment: () => dispatch({ type:
'INCREMENT' })
  };
}
```

```
function Counter({ count, increment }) {
  return (
    <div>
      <p>Count: {count}</p>
      <button
onClick={increment}>Increment</button>
    </div>
  );
}
```

```
const ConnectedCounter =
connect(mapStateToProps,
mapDispatchToProps)(Counter);
```

```
function App() {
  return (
```

```

    <Provider store={store}>
      <ConnectedCounter />
    </Provider>
  );
}

```

28. What is an action creator in Redux?

An action creator is a function that returns an action object.

javascript
Copy code

```

// Example: Action creator
function increment() {
  return { type: 'INCREMENT' };
}

```

29. Explain reducers in Redux.

Reducers are pure functions that take the current state and an action as arguments and return a new state.

javascript
Copy code

```

// Example: Reducer
const initialState = { count: 0 };

function reducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    default:
      return state;
  }
}

```

30. How does Redux store work?

The Redux store holds the state of the application, allows access to state via `getState()`, dispatches actions via `dispatch(action)`, and registers listeners via `subscribe(listener)`.

javascript
Copy code

```

// Example: Redux store
const store = createStore(reducer);

store.subscribe(() =>
  console.log(store.getState()));

store.dispatch({ type: 'INCREMENT' });

```

Hooks

31. What are hooks in React? Why were they introduced?

Hooks are functions that let you use state and other React features in functional components. They were introduced to provide a way to use state and lifecycle methods without writing class components.

javascript
Copy code

```

// Example: useState and useEffect hooks
function Counter() {
  const [count, setCount] = useState(0);

```

```

  useEffect(() => {
    document.title = `You clicked
    ${count} times`;
  }, [count]);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() =>
        setCount(count + 1)}>Click me</button>
    </div>
  );
}

```

32. Explain the `useState` hook in detail.

`useState` is a hook that allows you to add state to functional components. It returns an array with two values: the current state and a function to update it.

javascript
Copy code

```

// Example: useState hook
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() =>
        setCount(count + 1)}>Click me</button>
    </div>
  );
}

```

33. Explain the `useEffect` hook in detail.

`useEffect` is a hook that performs side effects in functional components. It runs after the render and can optionally clean up after itself.

javascript
Copy code

```

// Example: useEffect hook
function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => {
      setCount(prevCount => prevCount + 1);
    }, 1000);

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

  return <div>Timer: {count}</div>;
}

```

34. How do you create custom hooks in React?

Custom hooks are functions that use other hooks to encapsulate and reuse stateful logic.

javascript
Copy code

```

// Example: Custom hook
function useCounter(initialCount = 0) {
  const [count, setCount] =
    useState(initialCount);

```

```

    const increment = () => setCount(count + 1);
    const decrement = () => setCount(count - 1);

    return { count, increment, decrement };
}

```

```

function Counter() {
  const { count, increment, decrement } = useCounter();

```

```

  return (
    <div>
      <p>Count: {count}</p>
      <button
onClick={increment}>Increment</button>
      <button
onClick={decrement}>Decrement</button>
    </div>
  );
}

```

-
- 35. **What are the rules of hooks in React?**
 - Only call hooks at the top level.

Only call hooks from React function components or custom hooks.

```

javascript
Copy code
// Example: Valid use of hooks
function Counter() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked
${count} times`;
  }, [count]);
  return <button onClick={() =>
setCount(count + 1)}>Click me</button>;
}

```

-
- 36. **How do you optimize performance with hooks?**
 - Use `useMemo` to memoize expensive calculations.
 - Use `useCallback` to memoize event handlers.

Use `React.memo` to prevent unnecessary re-renders of functional components.

```

javascript
Copy code
// Example: useMemo and useCallback hooks
function ExpensiveComponent({ compute, count }) {
  const result = useMemo(() =>
compute(count), [compute, count]);

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

```

```

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

  const compute = useCallback((num) => {
    // Expensive computation here
    return num * 2;
  }, []);

  return (
    <div>

```

```

      <ExpensiveComponent
compute={compute} count={count} />
      <button onClick={() =>
setCount(count + 1)}>Increment</button>
    </div>
  );
}

```

○

Routing and Navigation

- 37. **How do you implement routing in a React application?**

Use `react-router-dom` for declarative routing.

```

javascript
Copy code
// Example: Basic routing with
react-router-dom
import { BrowserRouter as Router, Route,
Switch, Link } from 'react-router-dom';

```

```

function Home() {
  return <h2>Home</h2>;
}

```

```

function About() {
  return <h2>About</h2>;
}

```

```

function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link>
        <Link
to="/about">About</Link>
      </nav>
      <Switch>
        <Route exact path="/"
component={Home} />
        <Route path="/about"
component={About} />
      </Switch>
    </Router>
  );
}

```

○

- 38. **Explain the role of `react-router` in React applications.**

- `react-router` provides a declarative way to handle routing in React applications, allowing for dynamic routing, nested routes, and route parameters.

- 39. **How do you handle dynamic routing in React?**

Use route parameters and `useParams` hook from `react-router-dom`.

```

javascript
Copy code
// Example: Dynamic routing with
react-router-dom
import { useParams } from 'react-router-dom';

function UserProfile() {
  const { userId } = useParams();
  return <div>User ID: {userId}</div>;
}

```

```
function App() {
  return (
    <Router>
      <Switch>
        <Route path="/user/:userId"
component={UserProfile} />
      </Switch>
    </Router>
  );
}
```

40. What are nested routes in React-router?

Nested routes allow you to render child routes within parent routes.

javascript
Copy code

```
// Example: Nested routes with
react-router-dom
function Dashboard() {
  return (
    <div>
      <h2>Dashboard</h2>
      <Switch>
        <Route
path="/dashboard/overview"
component={Overview} />
        <Route
path="/dashboard/stats" component={Stats} />
      </Switch>
    </div>
  );
}
```

```
function App() {
  return (
    <Router>
      <Switch>
        <Route path="/dashboard"
component={Dashboard} />
      </Switch>
    </Router>
  );
}
```

41. How do you pass parameters in the URL using React-router?

Use route parameters with the `useParams` hook.

javascript
Copy code

```
// Example: Passing parameters in the URL
function UserProfile() {
  const { userId } = useParams();
  return <div>User ID: {userId}</div>;
}

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/user/:userId"
component={UserProfile} />
      </Switch>
    </Router>
  );
}
```

Forms and Data Handling (continued)

41. How do you pass parameters in the URL using React-router?

Use route parameters with the `useParams` hook.

javascript
Copy code

```
// Example: Passing parameters in the URL
import { BrowserRouter as Router, Route,
Switch, useParams } from 'react-router-dom';
```

```
function UserProfile() {
  const { userId } = useParams();
  return <div>User ID: {userId}</div>;
}
```

```
function App() {
  return (
    <Router>
      <Switch>
        <Route path="/user/:userId"
component={UserProfile} />
      </Switch>
    </Router>
  );
}
```

•

Component Lifecycle

48. Explain the lifecycle of a React component.

- **Mounting:** The component is being inserted into the DOM.
 - `constructor()`
 - `static getDerivedStateFromProps()`
 - `render()`
 - `componentDidMount()`
- **Updating:** The component is being re-rendered as a result of changes to its props or state.
 - `static getDerivedStateFromProps()`
 - `shouldComponentUpdate()`
 - `render()`
 - `getSnapshotBeforeUpdate()`
 - `componentDidUpdate()`
- **Unmounting:** The component is being removed from the DOM.
 - `componentWillUnmount()`
- **Error Handling:** Called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.
 - `static getDerivedStateFromError()`
 - `componentDidCatch()`

javascript
Copy code

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
```



```

    static
    getDerivedStateFromProps(nextProps,
    prevState) {
        // Return an object to update state,
        or null to update nothing
        return null;
    }

    componentDidMount() {
        // Perform any initial data loading
    }

    shouldComponentUpdate(nextProps,
    nextState) {
        // Return false to prevent
        re-rendering
        return true;
    }

    getSnapshotBeforeUpdate(prevProps,
    prevState) {
        // Capture some information from the
        DOM before it is potentially changed
        return null;
    }

    componentDidUpdate(prevProps, prevState,
    snapshot) {
        // Perform any post-update tasks
    }

    componentWillUnmount() {
        // Clean up any resources
    }

    componentDidCatch(error, info) {
        // Handle errors in the component
        this.setState({ hasError: true });
    }

    render() {
        if (this.state.hasError) {
            return <div>Error occurred</div>;
        }
        return <div>My Component</div>;
    }
}

```

49.

50. What are the deprecated lifecycle methods in React 16?

- The following lifecycle methods are deprecated and should not be used:
 - `componentWillMount()`
 - `componentWillReceiveProps()`
 - `componentWillUpdate()`

51. Instead, use:

- `componentDidMount()` for `componentWillMount()`
- `static getDerivedStateFromProps()` for `componentWillReceiveProps()`
- `getSnapshotBeforeUpdate()` for `componentWillUpdate()`

javascript
Copy code

```

// Deprecated lifecycle methods (avoid using
these)
class MyComponent extends React.Component {

```

```

    componentWillMount() {
        // Use componentDidMount instead
    }

    componentWillReceiveProps(nextProps) {
        // Use static
        getDerivedStateFromProps instead
    }

    componentWillUpdate(nextProps, nextState)
    {
        // Use getSnapshotBeforeUpdate
        instead
    }

    render() {
        return <div>My Component</div>;
    }
}

```

52.

53. How do you handle side effects in React components?

Use the `useEffect` hook for handling side effects in functional components. This includes tasks such as data fetching, subscriptions, or manually changing the DOM.

javascript
Copy code

```

// Example: Handling side effects with
useEffect
function MyComponent() {
    useEffect(() => {
        // Perform a side effect
        document.title = "New Title";

        // Cleanup function (optional)
        return () => {
            document.title = "Original
            Title";
        };
    }, []); // Empty dependency array ensures
    it runs only once

    return <div>My Component</div>;
}

```

○

Performance Optimization

51. How do you optimize performance in React applications?

- **Use `React.memo`:** Memoize functional components to prevent unnecessary re-renders.
- **Use `shouldComponentUpdate` or `PureComponent`:** Optimize class components by preventing unnecessary updates.
- **Use the `useMemo` and `useCallback` hooks:** Memoize values and functions.
- **Code splitting:** Use dynamic imports to split code.
- **Virtualize long lists:** Use libraries like `react-window` or `react-virtualized`.

javascript
Copy code

```

// Example: Using React.memo

```



```
const MyComponent = React.memo(function
MyComponent(props) {
  return <div>{props.value}</div>;
});

// Example: Using useMemo
function MyComponent({ items }) {
  const memoizedItems = useMemo(() =>
items.filter(item => item.active), [items]);
  return <div>{memoizedItems.length}</div>;
}

// Example: Using useCallback
function ParentComponent() {
  const [count, setCount] = useState(0);

  const handleIncrement = useCallback(() =>
{
  setCount(count + 1);
}, [count]);

  return <ChildComponent
onIncrement={handleIncrement} />;
}
```

52.

53. **Explain memoization in React.**

- Memoization is a performance optimization technique to store the results of expensive function calls and return the cached result when the same inputs occur again.
- In React, `React.memo` memoizes functional components, `useMemo` memoizes values, and `useCallback` memoizes functions.

javascript
Copy code

```
// Example: Using React.memo
const MyComponent = React.memo(function
MyComponent(props) {
  return <div>{props.value}</div>;
});

// Example: Using useMemo
function MyComponent({ items }) {
  const memoizedItems = useMemo(() =>
items.filter(item => item.active), [items]);
  return <div>{memoizedItems.length}</div>;
}

// Example: Using useCallback
function ParentComponent() {
  const [count, setCount] = useState(0);

  const handleIncrement = useCallback(() =>
{
  setCount(count + 1);
}, [count]);

  return <ChildComponent
onIncrement={handleIncrement} />;
}
```

54.

55. **What are the performance best practices for React applications?**

- **Avoid inline functions in render:** Use `useCallback` to memoize them.
- **Use production build:** Ensure the app is built for production.

- **Minimize re-renders:** Use `React.memo`, `shouldComponentUpdate`, or `PureComponent`.
- **Code splitting:** Dynamically import components.
- **Use virtualization for long lists:** Use libraries like `react-window` or `react-virtualized`.

javascript
Copy code

```
// Example: Avoiding inline functions
function MyComponent({ onClick }) {
  return <button onClick={onClick}>Click
Me</button>;
}

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

  const handleClick = useCallback(() => {
    setCount(count + 1);
  }, [count]);

  return <MyComponent onClick={handleClick}
/>;
}
```

56.

57. **How do you profile React applications for performance improvements?**

- Use the React DevTools Profiler to measure the performance of React components.
- Use the built-in performance tools in browsers like Chrome's Performance panel.

javascript
Copy code

```
// Example: Using React DevTools Profiler
// 1. Open React DevTools.
// 2. Navigate to the Profiler tab.
// 3. Click on "Record" and perform actions in your app.
// 4. Analyze the recorded profile for performance bottlenecks.
```

58.

Context API

55. **What is Context API in React?**

- Context API provides a way to pass data through the component tree without having to pass props down manually at every level.
- It's used for sharing state, themes, or other common properties across components.

javascript
Copy code

```
// Example: Creating and consuming context
const MyContext = React.createContext();

function MyProvider({ children }) {
  const value = "some value";
  return <MyContext.Provider
value={value}>{children}</MyContext.Provider>
;
}
```

```
function MyComponent() {
  const context = useContext(MyContext);
  return <div>{context}</div>;
}
```

56.

57. **How do you create and consume context in React?**

Creating context:

javascript
Copy code

```
const MyContext =
  React.createContext(defaultValue);
```

○

Providing context:

javascript
Copy code

```
function MyProvider({ children }) {
  const value = "some value";
  return <MyContext.Provider
    value={value}>{children}</MyContext.Provider>
  ;
}
```

○

Consuming context:

javascript
Copy code

```
function MyComponent() {
  const context = useContext(MyContext);
  return <div>{context}</div>;
}
```

○

58. **When would you use Context API over props drilling?**

- Use Context API when you need to share state or data across multiple levels of the component tree, making the code cleaner and avoiding the "props drilling" problem.
- Example: Sharing a theme or user authentication status across many components.

javascript
Copy code

```
// Example: Using Context API to avoid props drilling
const ThemeContext =
  React.createContext('light');
```

```
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

```
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
```

```
function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button style={{ background: theme
    === 'dark' ? '#333' : '#fff'
  }}>Button</button>;
}
```

59.

Testing

58. **How do you test React components?**

- Use libraries like Jest for testing JavaScript and React Testing Library for testing React components.
- Write unit tests for individual components and integration tests for component interactions.

javascript
Copy code

```
// Example: Testing a React component with
  React Testing Library
import { render, screen } from
  '@testing-library/react';
import MyComponent from './MyComponent';
```

```
test('renders MyComponent with correct text',
  () => {
    render(<MyComponent />);
    const linkElement =
      screen.getByText(/some text/i);
    expect(linkElement).toBeInTheDocument();
  });
```

59.

60. **What are the popular testing libraries for React?**

- **Jest:** A JavaScript testing framework used for unit tests, snapshot tests, and more.
- **React Testing Library:** A library for testing React components by querying the DOM.
- **Enzyme:** A JavaScript testing utility for React, useful for shallow rendering and DOM manipulation.

javascript
Copy code

```
// Example: Setting up Jest and React Testing
  Library
// Install Jest and React Testing Library
// npm install --save-dev jest
@testing-library/react
```

```
// Add a test script in package.json
"scripts": {
  "test": "jest"
}
```

```
// Write a test file (MyComponent.test.js)
import { render, screen } from
  '@testing-library/react';
import MyComponent from './MyComponent';
```

```
test('renders MyComponent with correct text',
  () => {
    render(<MyComponent />);
    const linkElement =
      screen.getByText(/some text/i);
    expect(linkElement).toBeInTheDocument();
  });
```

- 61.
62. **How do you perform unit testing on React components?**
- Write unit tests to test individual components or functions in isolation.
 - Use Jest for running tests and assertions, and React Testing Library for rendering components and querying the DOM.

javascript
Copy code

```
// Example: Unit testing a React component
import { render, screen } from
 '@testing-library/react';
import MyComponent from './MyComponent';

test('renders MyComponent with correct text',
 () => {
   render(<MyComponent />);
   const linkElement =
 screen.getByText(/some text/i);
   expect(linkElement).toBeInTheDocument();
 });

// Example: Unit testing a function
function add(a, b) {
  return a + b;
}

test('adds 1 + 2 to equal 3', () => {
  expect(add(1, 2)).toBe(3);
});
```

63.

Styling in React

61. **What are the different ways to style components in React?**
- **CSS:** Standard CSS files.
 - **Inline styles:** Using the `style` attribute.
 - **CSS-in-JS:** Libraries like styled-components or emotion.
 - **CSS Modules:** Scoped CSS classes.

javascript
Copy code

```
// Example: Inline styles
function MyComponent() {
  return <div style={{ color: 'red' }}>Styled with inline styles</div>;
}

// Example: CSS-in-JS with styled-components
import styled from 'styled-components';

const StyledDiv = styled.div`
  color: red;
`;

function MyComponent() {
  return <StyledDiv>Styled with
  styled-components</StyledDiv>;
}
```

62.

63. **Explain CSS Modules in React.**
- CSS Modules are a way to scope CSS locally to avoid conflicts between different components.
 - In CSS Modules, class names are automatically scoped locally by default.

css

Copy code

```
/* Example: styles.module.css */
.header {
  color: blue;
}

javascript
Copy code
// Example: Using CSS Modules in a component
import styles from './styles.module.css';
```

```
function MyComponent() {
  return <div
    className={styles.header}>Styled with CSS
    Modules</div>;
}
```

64.

65. **How do you use styled-components in React?**
- styled-components is a library that allows you to write CSS-in-JS.
 - It allows you to define styled components with tagged template literals.

javascript
Copy code

```
// Example: Using styled-components
import styled from 'styled-components';

const StyledButton = styled.button`
  background-color: blue;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;

  &:hover {
    background-color: darkblue;
  }
`;

function MyComponent() {
  return <StyledButton>Styled
  Button</StyledButton>;
}
```

66.

Error Handling

64. **How do you handle errors in React applications?**
- Use Error Boundaries to catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI.

javascript
Copy code

```
// Example: Error Boundary component
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
```

```

        console.error("Error caught by Error
Boundary:", error, errorInfo);
    }

    render() {
        if (this.state.hasError) {
            return <h1>Something went
wrong.</h1>;
        }
        return this.props.children;
    }
}

function MyComponent() {
    return (
        <ErrorBoundary>
            <ComponentThatMayError />
        </ErrorBoundary>
    );
}

```

65.

66. What are Error Boundaries in React?

- Error Boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.
- They catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

```

javascript
Copy code
// Example: Implementing an Error Boundary
class ErrorBoundary extends React.Component {
    constructor(props) {
        super(props);
        this.state = { hasError: false };
    }

    static getDerivedStateFromError(error) {
        return { hasError: true };
    }

    componentDidCatch(error, errorInfo) {
        console.error("Error caught by Error
Boundary:", error, errorInfo);
    }

    render() {
        if (this.state.hasError) {
            return <h1>Something went
wrong.</h1>;
        }
        return this.props.children;
    }
}

```

67.

Integrations

66. How do you integrate React with other libraries or frameworks?

- Integrate React with other libraries or frameworks by using standard import/export statements and embedding them within React components.
- Example: Integrating a charting library like D3.js with React.

```

javascript
Copy code
// Example: Integrating D3.js with React
import React, { useRef, useEffect } from
'react';
import * as d3 from 'd3';

function MyChart() {
    const ref = useRef();

    useEffect(() => {
        const svg = d3.select(ref.current)
            .append('svg')
            .attr('width', 500)
            .attr('height', 500);

        svg.append('circle')
            .attr('cx', 250)
            .attr('cy', 250)
            .attr('r', 50)
            .attr('fill', 'blue');
    }, []);

    return <div ref={ref}></div>;
}

```

67.

68. How do you use React with TypeScript?

- Use TypeScript with React by installing the necessary TypeScript packages and setting up a `tsconfig.json` file.
- Define types for props and state in your components.

```

typescript
Copy code
// Example: Using TypeScript with React
import React from 'react';

interface MyComponentProps {
    message: string;
}

const MyComponent: React.FC<MyComponentProps>
= ({ message }) => {
    return <div>{message}</div>;
};

export default MyComponent;

```

69.

70. How do you integrate React with GraphQL?

- Use the Apollo Client to integrate React with GraphQL.
- Set up an ApolloProvider and use the `useQuery` hook to fetch data.

```

javascript
Copy code
// Example: Integrating React with GraphQL
using Apollo Client
import React from 'react';
import { ApolloProvider, ApolloClient,
InMemoryCache, useQuery, gql } from
'@apollo/client';

const client = new ApolloClient({
    uri: 'https://example.com/graphql',
    cache: new InMemoryCache()
});

const GET_DATA = gql`

```

```

    query GetData {
      data {
        id
        value
      }
    }
  }
`;

function MyComponent() {
  const { loading, error, data } =
    useQuery(GET_DATA);

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error :(</p>;

  return (
    <div>
      {data.data.map(({ id, value }) =>
        (
          <div key={id}>{value}</div>
        ))}
    </div>
  );
}

function App() {
  return (
    <ApolloProvider client={client}>
      <MyComponent />
    </ApolloProvider>
  );
}

```

71.

State Management

69. How do you manage global state in a React application?

- Use context and the `useReducer` hook for a built-in solution.
- Use state management libraries like Redux, MobX, or Zustand for more complex state management needs.

javascript
Copy code

```

// Example: Managing global state with
context and useReducer
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();
  }
}

const CountContext = React.createContext();

function CountProvider({ children }) {
  const [state, dispatch] =
    useReducer(reducer, initialState);
  return (

```

```

    <CountContext.Provider value={{
      state, dispatch }}>
      {children}
    </CountContext.Provider>
  );
}

function Counter() {
  const { state, dispatch } =
    useContext(CountContext);
  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({
        type: 'increment' })}>Increment</button>
      <button onClick={() => dispatch({
        type: 'decrement' })}>Decrement</button>
    </div>
  );
}

function App() {
  return (
    <CountProvider>
      <Counter />
    </CountProvider>
  );
}

```

70.

71. Explain Redux and its core principles.

- Redux is a state management library for JavaScript applications.
- **Core principles:**
 - **Single source of truth:** The state of your whole application is stored in an object tree within a single store.
 - **State is read-only:** The only way to change the state is to emit an action, an object describing what happened.
 - **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write pure reducers.

javascript
Copy code

```

// Example: Basic Redux setup
import { createStore } from 'redux';

// Initial state
const initialState = { count: 0 };

// Reducer function
function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

// Create Redux store
const store = createStore(counterReducer);

```

```
// Action creators
const increment = () => ({ type: 'increment'
});
const decrement = () => ({ type: 'decrement'
});

// Dispatch actions
store.dispatch(increment());
```

72. store.dispatch(decrement());
73. **Explain Redux and its core principles.**
- Redux is a state management library for JavaScript applications.
 - **Core principles:**
 - **Single source of truth:** The state of your whole application is stored in an object tree within a single store.
 - **State is read-only:** The only way to change the state is to emit an action, an object describing what happened.
 - **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write pure reducers.

```
javascript
Copy code
// Example: Basic Redux setup
import { createStore } from 'redux';

// Initial state
const initialState = { count: 0 };

// Reducer function
function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

// Create Redux store
const store = createStore(counterReducer);

// Action creators
const increment = () => ({ type: 'increment'
});
const decrement = () => ({ type: 'decrement'
});

// Dispatch actions
store.dispatch(increment());
store.dispatch(decrement());
```

- 74.
75. **What is the useReducer hook and when would you use it?**
- The useReducer hook is an alternative to useState for managing complex state logic.

- Use it when you have state logic that involves multiple sub-values or when the next state depends on the previous one.

```
javascript
Copy code
// Example: Using useReducer hook
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:
      return state;
  }
}

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

// Dispatch actions
dispatch({ type: 'increment' });
dispatch({ type: 'decrement' });
```

76.

40

continue from 71

```
ChatGPT
javascript
Copy code
// Example: Handling forms in React
function MyForm() {
  const [name, setName] = useState('');

  const handleChange = (event) => {
    setName(event.target.value);
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`A name was submitted: ${name}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text"
          value={name} onChange={handleChange} />
      </label>
      <button
        type="submit">Submit</button>
    </form>
  );
}
```

React Router

79. ****What is React Router?****
- React Router is a standard library for routing in React.
 - It enables the navigation among views of various components in a React application, allows changing the browser URL, and keeps the UI in sync with the URL.

```
````javascript
// Example: Basic React Router setup
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

function App() {
 return (
 <Router>
```

## Security

## 71. What are some security best practices in React applications?

- **Sanitize Inputs:** Always sanitize and validate user inputs to prevent injection attacks.
- **Avoid Dangerous APIs:** Avoid using `dangerouslySetInnerHTML` to prevent XSS attacks. Use libraries like `DOMPurify` to sanitize HTML content if necessary.
- **Use HTTPS:** Ensure your application uses HTTPS to encrypt data between the client and server.
- **Content Security Policy (CSP):** Implement a CSP to mitigate XSS and data injection attacks.
- **Security Headers:** Use security headers like `X-Content-Type-Options`, `X-Frame-Options`, `Strict-Transport-Security`, and `X-XSS-Protection`.
- **Environment Variables:** Store sensitive data (e.g., API keys) in environment variables, not in the source code.

## 72. How do you prevent XSS attacks in React?

- **Avoid `dangerouslySetInnerHTML`:** Do not use `dangerouslySetInnerHTML` unless absolutely necessary and sanitize input.
- **Escape User Inputs:** Always escape any user inputs before rendering them in the UI.
- **Use Trusted Libraries:** Use libraries like `DOMPurify` to sanitize HTML content.
- **Content Security Policy (CSP):** Implement CSP to reduce the risk of XSS attacks.

javascript  
Copy code

```
// Example: Using DOMPurify to sanitize HTML
import DOMPurify from 'dompurify';
```

```
function MyComponent({ htmlContent }) {
 const sanitizedContent =
 DOMPurify.sanitize(htmlContent);
 return <div dangerouslySetInnerHTML={{
 __html: sanitizedContent }} />;
}
```

73.

## Deployment

## 73. How do you deploy a React application?

- **Build the Application:** Use `npm run build` to create a production-ready build of your application.
- **Deploy to a Web Server:** Upload the build files to a web server or cloud service provider (e.g., AWS, Heroku, Netlify, Vercel).
- **Configure the Server:** Ensure the server serves the `index.html` file for all routes to handle client-side routing.
- **Continuous Integration/Continuous Deployment (CI/CD):** Use CI/CD tools to automate the deployment process.

## 74. What are the considerations for deploying a React application?

- **Environment Configuration:** Use environment variables for configuration

```
<Switch>
 <Route path="/about">
 <About />
 </Route>
 <Route path="/users">
 <Users />
 </Route>
 <Route path="/">
 <Home />
 </Route>
</Switch>
</Router>
);
}

function Home() {
 return <h2>Home</h2>;
}

function About() {
 return <h2>About</h2>;
}

function Users() {
 return <h2>Users</h2>;
}
...

```

80. \*\*How do you perform programmatic navigation in React Router?\*\*

- Use the `'useHistory'` hook from React Router to programmatically navigate to a different route.

```
```javascript
// Example: Programmatic navigation using useHistory
import { useHistory } from
'react-router-dom';

```

```
function MyComponent() {
  let history = useHistory();

  function handleClick() {
    history.push("/new-route");
  }

  return (
    <button type="button"
      onClick={handleClick}>
      Go to new route
    </button>
  );
}
...

```

81. **What is the purpose of the Switch component in React Router?**

- The `'Switch'` component is used to group `'Route'` components and ensures that only one route is rendered at a time.
- It renders the first child `'Route'` or `'Redirect'` that matches the location.

```
```javascript

```

40

continue from questions 71



settings (e.g., API endpoints, feature flags).

- **Static Asset Handling:** Ensure proper handling of static assets like images, fonts, and styles.
- **Caching and Performance:** Implement caching strategies and performance optimizations (e.g., code splitting, lazy loading).
- **Security:** Ensure security best practices are followed (e.g., HTTPS, CSP, secure headers).
- **SEO and Metadata:** Ensure proper SEO and metadata configuration (e.g., meta tags, Open Graph tags).

## Advanced Concepts

### 75. What are portals in React?

- **Portals** provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.
- Useful for rendering modals, tooltips, or any content that needs to visually break out of its container.

javascript  
Copy code

```
// Example: Creating a portal
import ReactDOM from 'react-dom';

function Modal({ children }) {
 return ReactDOM.createPortal(
 <div className="modal">
 {children}
 </div>,
 document.getElementById('modal-root')
);
}

function App() {
 return (
 <div>
 <h1>My App</h1>
 <Modal>
 <p>This is a modal!</p>
 </Modal>
 </div>
);
}
```

76.

### 77. How do you handle state management without Redux?

- **Context API and `useReducer`:** Use React's Context API combined with the `useReducer` hook to manage global state.
- **Hooks:** Use custom hooks to encapsulate and manage state logic.
- **Third-party Libraries:** Use other state management libraries like MobX, Recoil, or Zustand.

javascript  
Copy code

```
// Example: Managing state with Context API and useReducer
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();
 }
}

const CountContext = React.createContext();

function CountProvider({ children }) {
 const [state, dispatch] =
 useReducer(reducer, initialState);
 return (
 <CountContext.Provider value={{
 state, dispatch }}>
 {children}
 </CountContext.Provider>
);
}

function Counter() {
 const { state, dispatch } =
 useContext(CountContext);
 return (
 <div>
 <p>Count: {state.count}</p>
 <button onClick={() => dispatch({
 type: 'increment' })}>Increment</button>
 <button onClick={() => dispatch({
 type: 'decrement' })}>Decrement</button>
 </div>
);
}

function App() {
 return (
 <CountProvider>
 <Counter />
 </CountProvider>
);
}
```

78.

### 79. What are lazy loading and code splitting in React?

- **Lazy Loading:** Technique to defer loading of non-critical resources at the initial load time. Improves performance by loading components only when they are needed.
- **Code Splitting:** Breaking up the codebase into smaller chunks that can be loaded on demand. Usually implemented using `React.lazy` and `Suspense`.

javascript  
Copy code

```
// Example: Implementing lazy loading and code splitting
import React, { Suspense, lazy } from 'react';

const LazyComponent = lazy(() =>
 import('./LazyComponent'));

function App() {
 return (
 <div>
```

```

 <h1>My App</h1>
 <Suspense
fallback={<div>Loading...</div>}>
 <LazyComponent />
 </Suspense>
 </div>
);
 }

```

80.

#### 81. How do you handle animations in React?

- **CSS Animations:** Use CSS animations and transitions for simple animations.
- **React Transition Group:** A library that provides components for managing component transition effects.
- **Framer Motion:** A popular animation library for React that provides declarative animations and gestures.

javascript  
Copy code

```

// Example: Using React Transition Group
import { CSSTransition, TransitionGroup }
from 'react-transition-group';

function TodoList({ items }) {
 return (
 <TransitionGroup>
 {items.map((item) => (
 <CSSTransition key={item.id}
timeout={500} classNames="fade">
 <div>{item.text}</div>
 </CSSTransition>
))}
 </TransitionGroup>
);
}

```

82.

### Real-world Scenarios

#### 79. How do you handle authentication in a React application?

- **Authentication Libraries:** Use libraries like Firebase, Auth0, or JWT for authentication.
- **Context API:** Manage authentication state using Context API and hooks.
- **Protected Routes:** Implement protected routes that require authentication.

javascript  
Copy code

```

// Example: Protected Route
import { Route, Redirect } from
'react-router-dom';

function PrivateRoute({ component: Component,
...rest }) {
 const isAuthenticated = /* logic to
determine if authenticated */;
 return (
 <Route
 {...rest}
 render={((props) =>
 isAuthenticated ? (
 <Component {...props} />
) : (
 <Redirect to="/login" />
)
)}
 >
)
}

```

```

 />
);
}

```

80.

#### 81. How do you handle routing transitions in React?

- **React Transition Group:** Use [React Transition Group](#) to animate route transitions.
- **Custom Animations:** Implement custom animations using CSS or animation libraries.

javascript  
Copy code

```

// Example: Route transitions with React
Transition Group
import { Switch, Route, useLocation } from
'react-router-dom';
import { CSSTransition, TransitionGroup }
from 'react-transition-group';

function App() {
 const location = useLocation();
 return (
 <TransitionGroup>
 <CSSTransition key={location.key}
classNames="fade" timeout={300}>
 <Switch location={location}>
 <Route exact path="/"
component={Home} />
 <Route path="/about"
component={About} />
 <Route path="/contact"
component={Contact} />
 </Switch>
 </CSSTransition>
 </TransitionGroup>
);
}

```

82.

### React Native

#### 81. What is React Native?

- React Native is a framework for building native mobile applications using React.
- It allows developers to use React to build mobile apps with native components for iOS and Android.

#### 82. How does React Native differ from React?

- **Rendering:** React Native uses native components, while React uses HTML and the browser's DOM.
- **Styling:** React Native uses [StyleSheet](#) and JavaScript objects for styling, whereas React typically uses CSS.
- **Navigation:** React Native uses libraries like [react-navigation](#), while React uses [react-router](#).

javascript  
Copy code

```

// Example: Basic React Native component
import React from 'react';
import { View, Text, StyleSheet } from
'react-native';

function App() {
 return (
 <View style={styles.container}>

```

```

 <Text>Hello, React Native!</Text>
 </View>
);
 }

const styles = StyleSheet.create({
 container: {
 flex: 1,
 justifyContent: 'center',
 alignItems: 'center',
 },
});

export default App;

```

83.

#### 84. How do you create a mobile application using React Native?

- **Setup Environment:** Install Node.js, React Native CLI, Android Studio, and Xcode (for iOS).
- **Create Project:** Use `react-native init` to create a new React Native project.
- **Develop and Test:** Write React Native code and test it on Android and iOS emulators or real devices.
- **Build and Deploy:** Build the app for release and deploy it to app stores.

bash

Copy code

```

npx react-native init MyApp
cd MyApp
npx react-native run-android # or npx
react-native run-ios

```

85.

### Performance Monitoring and Optimization

#### 84. How do you monitor performance in a React application?

- **React Developer Tools:** Use the React DevTools to inspect the component tree and measure performance.
- **Browser Performance Tools:** Use browser performance tools like Chrome DevTools for detailed performance analysis.
- **Third-party Monitoring Services:** Integrate services like New Relic, Sentry, or Datadog to monitor performance in production.

#### 85. What tools would you use to identify performance bottlenecks in React?

- **React Profiler:** Use the React Profiler API to measure performance.
- **Chrome DevTools:** Utilize Chrome DevTools for performance profiling and analysis.
- **Web Vitals:** Use the Web Vitals library to measure key performance metrics.
- **Lighthouse:** Use Lighthouse to audit performance, accessibility, and best practices.

### Accessibility

#### 86. How do you ensure accessibility in React applications?

- **Semantic HTML:** Use semantic HTML elements to improve accessibility.

- **ARIA Roles:** Use ARIA roles and attributes to provide additional context to screen readers.
- **Focus Management:** Ensure proper focus management, especially for interactive elements.
- **Testing Tools:** Use tools like aXe, Lighthouse, and screen readers to test accessibility.

javascript

Copy code

```

// Example: Using ARIA attributes
function AccessibleButton() {
 return (
 <button aria-label="Close"
 onClick={() => console.log('Closed')}>
 x
 </button>
);
}

```

87.

#### 88. What are ARIA roles in React?

- ARIA (Accessible Rich Internet Applications) roles provide extra information to assistive technologies about the purpose and behavior of elements.
- They are used to enhance the accessibility of web applications by defining roles, states, and properties.

javascript

Copy code

```

// Example: Using ARIA roles
function Navigation() {
 return (
 <nav aria-label="Main navigation">

 Home
 About
 Contact

 </nav>
);
}

```

89.

### Redux Middleware

#### 88. What is Redux middleware? Give examples.

- Redux middleware provides a way to interact with dispatched actions before they reach the reducer.
- Examples: `redux-thunk`, `redux-saga`, and `redux-logger`.

javascript

Copy code

```

// Example: Applying middleware in Redux
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

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

```

89.

90. **How do you write custom middleware for Redux?**

- Custom middleware is a function that receives `store` and returns a function that takes `next`, which in turn returns a function that takes `action`.

javascript  
Copy code

```
// Example: Custom logging middleware
const loggerMiddleware = store => next =>
action => {
 console.log('Dispatching:', action);
 let result = next(action);
 console.log('Next state:',
store.getState());
 return result;
};
```

```
const store = createStore(rootReducer,
applyMiddleware(loggerMiddleware));
```

91.

**Redux Thunk**

90. **What is Redux Thunk used for?**

- Redux Thunk is middleware that allows you to write action creators that return a function instead of an action.
- It is commonly used for handling asynchronous logic like API calls.

javascript  
Copy code

```
// Example: Redux Thunk for async actions
import { createStore, applyMiddleware } from
'redux';
import thunk from 'redux-thunk';
```

```
const fetchUser = userId => {
 return async dispatch => {
 const response = await
fetch(`/api/user/${userId}`);
 const user = await response.json();
 dispatch({ type: 'USER_LOADED',
payload: user });
 };
};
```

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

91.

92. **How do you handle asynchronous actions in Redux with Redux Thunk?**

- Create an action creator that returns a function.
- Inside the function, perform the asynchronous operation and dispatch actions based on the result.

javascript  
Copy code

```
// Example: Handling async actions with Redux
Thunk
const fetchData = () => {
 return async dispatch => {
 dispatch({ type: 'FETCH_DATA_REQUEST'
});
 try {
```

```
const response = await
fetch('/api/data');
 const data = await
response.json();
 dispatch({ type:
'FETCH_DATA_SUCCESS', payload: data });
 } catch (error) {
 dispatch({ type:
'FETCH_DATA_FAILURE', error });
 }
};
};
```

93.

**Server-side Rendering (SSR)**

92. **What is Server-side Rendering (SSR) in React?**

- SSR is the process of rendering React components on the server and sending the HTML to the client.
- It improves performance and SEO by delivering fully rendered pages on the first request.

93. **How do you implement SSR in a React application?**

- Use a framework like Next.js that provides built-in support for SSR.
- Alternatively, set up your own SSR using Node.js and libraries like Express and ReactDOMServer.

javascript  
Copy code

```
// Example: Basic SSR setup with Express
import express from 'express';
import React from 'react';
import ReactDOMServer from
'react-dom/server';
import App from './App';
```

```
const server = express();
```

```
server.get('*', (req, res) => {
 const appString =
ReactDOMServer.renderToString(<App />);
 res.send(`
 <html>
 <head><title>SSR React
App</title></head>
 <body>
 <div
id="root">${appString}</div>
 <script
src="/bundle.js"></script>
 </body>
 </html>
 `);
});
```

```
server.listen(3000, () => {
 console.log('Server is running on port
3000');
});
```

94.

**Context and useContext Hook**

94. **What is the useContext hook in React?**

- The `useContext` hook allows you to consume a context value in a functional component.
- It simplifies accessing the context compared to using the `Context.Consumer` component.

javascript  
Copy code

```
// Example: Using useContext hook
const ThemeContext =
 React.createContext('light');

function ThemedComponent() {
 const theme = useContext(ThemeContext);
 return <div style={{ background: theme
=== 'dark' ? '#333' : '#FFF' }}>Themed
Component</div>;
}
```

95.

96. **How do you use context to pass data through the component tree in React?**

- Create a context using `React.createContext`.
- Provide the context value using `Context.Provider`.
- Consume the context value using `useContext` or `Context.Consumer`.

javascript  
Copy code

```
// Example: Passing data through context
const UserContext = React.createContext();

function App() {
 const user = { name: 'John', age: 30 };
 return (
 <UserContext.Provider value={user}>
 <UserProfile />
 </UserContext.Provider>
);
}

function UserProfile() {
 const user = useContext(UserContext);
 return <div>User: {user.name}</div>;
}
```

97.

## useMemo and useCallback Hooks

96. **What is the `useMemo` hook in React and when do you use it?**

- `useMemo` is a hook that memoizes a computed value, recomputing it only when its dependencies change.
- It is used to optimize performance by avoiding expensive calculations on every render.

javascript  
Copy code

```
// Example: Using useMemo
const expensiveCalculation = (num) => {
 // expensive computation
 return num * 2;
};

function MyComponent({ number }) {
 const memoizedValue = useMemo(() =>
expensiveCalculation(number), [number]);
```

```
 return <div>Result:
{memoizedValue}</div>;
 }
```

97.

98. **What is the `useCallback` hook in React and when do you use it?**

- `useCallback` is a hook that returns a memoized version of the callback function.
- It is used to prevent unnecessary re-creations of the function, which can improve performance, especially when passing callbacks to child components.

javascript  
Copy code

```
// Example: Using useCallback
const MyComponent = ({ onClick }) => {
 const handleClick = useCallback(() => {
 onClick();
 }, [onClick]);

 return <button onClick={
```

99.

4o

continue

ChatGPT  
javascript  
Copy code

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

 // Example usage
 function App() {
 const handleClick = () => {
 console.log('Button clicked!');
 };

 return (
 <div>
 <MyComponent
onClick={handleClick} />
 </div>
);
 }
 ...
```

## ### React Concurrent Mode

98. **\*\*What is React Concurrent Mode and what problem does it solve?\*\***

- React Concurrent Mode is an experimental feature that aims to improve the responsiveness and user experience of React applications by allowing React to interrupt rendering work to handle higher-priority updates (like user interactions) without blocking the browser's main thread.
- It addresses the problem of janky and unresponsive user interfaces by enabling smoother interactions and animations.

```
```javascript
```

```
// Example: Concurrent mode placeholder
function App() {
  return <h1>My App</h1>;
}
```

```
}
...

```

99. ****How does React Concurrent Mode affect the rendering pipeline?****

- Concurrent Mode enables React to schedule and prioritize updates, allowing more important updates (e.g., user interactions) to interrupt less important ones (e.g., rendering offscreen components).
- It helps in rendering more efficiently and maintaining responsiveness even when there are multiple updates happening simultaneously.

```
```javascript
// Example: Concurrent mode placeholder
function App() {
 return <h1>My App</h1>;
}
...

```

### ### React Suspense

100. **\*\*What is React Suspense and how does it improve the user experience?\*\***

- React Suspense is another experimental feature that allows components to "suspend" rendering while waiting for some asynchronous operation to complete (e.g., data fetching).
- It improves the user experience by providing a better way to handle loading states and seamlessly integrate data fetching with component rendering.

```
```javascript
// Example: Using React Suspense for data fetching
import { Suspense } from 'react';

function UserProfile() {
  // Suspense wraps components that may need to wait for async data
  return (
    <Suspense
      fallback=<{<div>Loading...</div>>
        <UserData />
      </Suspense>
    >
  );
}

function UserData() {
  // Simulate fetching user data
  const userData = fetchUserData();

  return <div>User:
{userData.name}</div>;
}
...

```

These examples and explanations cover a wide range of topics in React, from basic concepts to more advanced features and best practices. Each example demonstrates how to implement various functionalities in a React application, helping you understand how to apply these concepts in your own projects. If you have any specific questions or need further clarification on any topic, feel free to ask!