React Interview Questions and Answers for Freshers

Basic React Concepts

1. What is React?

React is an open-source JavaScript library developed by Facebook for building user interfaces, particularly single-page applications. It allows developers to create reusable UI components and efficiently update and render components when data changes.

2. What are the key features of React?

- **Virtual DOM**: React creates a lightweight representation of the actual DOM for improved performance
- **Component-Based Architecture**: UI is broken down into reusable components
- **Unidirectional Data Flow**: Data flows in a single direction, making the code more predictable
- **JSX**: A syntax extension that allows writing HTML-like code in JavaScript
- **High Performance**: Efficient rendering and updates through its unique rendering mechanism

3. What is JSX?

JSX (JavaScript XML) is a syntax extension for JavaScript recommended by React. It allows you to write HTML structures in the same file as JavaScript code. Example:

```
const element = <h1>Hello, React!</h1>;
```

JSX gets transformed into regular JavaScript function calls by Babel.

4. Explain the difference between React and React Native

- **React**: A JavaScript library for building web interfaces
- **React Native**: A framework for building mobile applications using React
- React uses web components, while React Native uses native mobile components
- React renders to the browser's DOM, React Native renders to mobile platforms' native views

5. What are Components in React?

Components are the building blocks of React applications. They are reusable pieces of code that return React elements describing what should appear on the

screen. There are two types:

1. **Functional Components**:

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

2. **Class Components**:

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

6. What is the difference between State and Props?

- **Props (Properties)**:
  - Read-only data passed from parent to child component
  - Immutable
  - Used to communicate between components
- **State**:
  - Mutable data managed within the component
  - Can be changed using `setState()`
  - Represents the component's internal data

7. What is a Virtual DOM?

The Virtual DOM is a lightweight JavaScript representation of the actual DOM. When state changes in a React component, React first updates the Virtual DOM, compares it with the previous version, and then efficiently updates only the changed parts in the real DOM.

8. What are React Hooks?

Hooks are functions that let you use state and other React features in functional components. Some important hooks include: - `useState()`: Adds state to functional components - `useEffect()`: Performs side effects in functional components - `useContext()`: Consumes context in a functional component - `useReducer()`: Manages complex state logic - `useCallback()`: Memoizes callback functions - `useMemo()`: Memoizes computed values

9. Explain the useEffect Hook

`useEffect` allows you to perform side effects in functional components. It runs after every render and can be used for: - Data fetching - Directly updating the DOM - Setting up subscriptions - Timers

```
useEffect(() => {
  // Side effect code
  return () => {
    // Cleanup code (optional)
  };
}, [/* dependency array */]);
```

10. What is React Router?

React Router is a standard routing library for React that enables navigation among views in a React Application. It allows changing the browser URL and keeping the UI in sync with the URL.

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

function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </BrowserRouter>
  );
}
```

11. What is Redux?

Redux is a predictable state container for JavaScript apps, commonly used with React. It helps manage the entire application's state in a single immutable store.

Key concepts: - **Store**: Holds the whole state tree - **Actions**: Plain JavaScript objects that describe what happened - **Reducers**: Functions that specify how the application's state changes

12. What is the difference between == and ===?

- == (Loose Equality): Compares value after type coercion
- === (Strict Equality): Compares value and type without coercion

13. Explain Controlled vs Uncontrolled Components

- **Controlled Components**: Form data is handled by React state

```
function ControlledInput() {
  const [value, setValue] = useState('');
  return (
```

```
    <input
      value={value}
      onChange={(e) => setValue(e.target.value)}
    />
  );
}
```

- **Uncontrolled Components**: Form data is handled by the DOM itself

```
function UncontrolledInput() {
  const inputRef = useRef(null);
  return <input ref={inputRef} />;
}
```

14. What is Prop Drilling?

Prop drilling occurs when you pass props through multiple levels of components that don't need those props themselves.

Solution: Use Context API or State Management libraries like Redux.

15. Explain React Fragments

Fragments let you group multiple elements without adding an extra DOM node:

```
function Example() {
  return (
    <>
      <ChildA />
      <ChildB />
      <ChildC />
    </>
  );
}
```

16. What is the Difference Between `createElement` and `cloneElement`?

- `createElement()`: Creates a new React element
- `cloneElement()`: Clones an existing element and allows modification

17. What are Higher-Order Components (HOC)?

A Higher-Order Component is a function that takes a component and returns a new component with additional props or behavior:

```
function withLogging(WrappedComponent) {
  return class extends React.Component {
    componentDidMount() {
      console.log('Component mounted');
```

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

18. What is the Purpose of `key` Prop in Lists?

The `key` prop helps React identify which items have changed, been added, or been removed in a list:

```
function TodoList({ todos }) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>{todo.text}</li>
      ))}
    </ul>
  );
}
```

19. Explain React Lifecycle Methods

For Class Components: - **Mounting**: - `constructor()` - `render()` - `componentDidMount()`

- **Updating**:
  - `shouldComponentUpdate()`
  - `render()`
  - `componentDidUpdate()`
- **Unmounting**:
  - `componentWillUnmount()`

20. What is the Difference Between `useMemo` and `useCallback`?

- `useMemo`: Memoizes computed values
- `useCallback`: Memoizes entire function

```
// useMemo
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

// useCallback
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

... (continue with more questions to reach 100)

Advanced React Concepts

21. What is Code Splitting?

Code splitting allows you to split your application's JavaScript into smaller chunks, improving load time:

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <React.Suspense fallback={<div>Loading...</div>}>
      <OtherComponent />
    </React.Suspense>
  );
}
```

22. Explain Error Boundaries

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree:

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

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

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

    return this.props.children;
  }
}
```

(The document continues with more questions to reach 100 total…)

23. What is Prop Types and Type Checking?

Prop Types is a way to check the types of props in React:

```
import PropTypes from 'prop-types';
```

```jsx
function Greeting({ name }) {
  return <h1>Hello, {name}</h1>;
}

Greeting.propTypes = {
  name: PropTypes.string.isRequired
};
```

24. Explain React Memo

`React.memo` is a higher-order component that memoizes functional components to prevent unnecessary re-renders:

```jsx
const MyComponent = React.memo(function MyComponent(props) {
  // Component logic
});
```

25. What is the Context API?

Context provides a way to pass data through the component tree without passing props manually:

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

function ParentComponent() {
  return (
    <MyContext.Provider value={/* some value */}>
      <ChildComponent />
    </MyContext.Provider>
  );
}

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

26. Explain Server-Side Rendering (SSR) in React

Server-Side Rendering allows rendering React components on the server, improving initial load performance and SEO:

```jsx
// Using Next.js for SSR
function HomePage({ data }) {
  return <div>{data.content}</div>;
}

export async function getServerSideProps() {
```

```
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return { props: { data } };
}
```

27. What are React Portals?

Portals allow rendering children into a different part of the DOM:

```
function Modal({ children }) {
  return ReactDOM.createPortal(
    children,
    document.getElementById('modal-root')
  );
}
```

28. Explain Reconciliation in React

Reconciliation is the process React uses to diff one tree with another to determine what needs to be changed: - React uses a diffing algorithm to minimize DOM manipulation - Compares two trees efficiently - Uses key prop to identify which child elements have changed

29. What is the useReducer Hook?

`useReducer` is an alternative to `useState` for complex state logic:

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

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>+</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>-</button>
    </>
```

```
  );
}
```

30. What is Dependency Injection in React?

Dependency Injection in React involves passing dependencies through props or context:

```
// Context-based Dependency Injection
const ApiContext = React.createContext(null);

function App({ apiClient }) {
  return (
    <ApiContext.Provider value={apiClient}>
      <MainComponent />
    </ApiContext.Provider>
  );
}

function MainComponent() {
  const apiClient = useContext(ApiContext);
  // Use apiClient
}
```

31. Explain Memoization in React

Memoization is an optimization technique to cache expensive function calls:

```
// Memoizing a function
const memoizedValue = useMemo(() => {
  return computeExpensiveValue(a, b);
}, [a, b]);

// Memoizing a callback
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

32. What is Render Props Pattern?

Render props is a technique for sharing code between components:

```
function DataProvider({ render }) {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetchData().then(setData);
  }, []);
```

```
  return render(data);
}

function App() {
  return (
    <DataProvider
      render={(data) => (
        <div>{data ? data.name : 'Loading...'}</div>
      )}
    />
  );
}
```

33. Explain React Suspense

React Suspense allows you to manage loading states declaratively:

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

function MyComponent() {
  return (
    <React.Suspense fallback={<LoadingSpinner />}>
      <LazyComponent />
    </React.Suspense>
  );
}
```

34. What is the Difference Between Controlled and Uncontrolled Components?

Controlled Components: - Form data is controlled by React state - Every state mutation has an associated handler function - More predictable and easier to modify

Uncontrolled Components: - Form data is handled by the DOM itself - Use refs to access form values - Less control, but simpler implementation

35. What are React Hooks Rules?

1. Only call Hooks at the top level of your component
2. Only call Hooks from React function components
3. Custom Hooks should start with 'use'

36. Explain Custom Hooks

Custom Hooks allow you to extract component logic into reusable functions:

```jsx
function useCustomHook(initialValue) {
  const [value, setValue] = useState(initialValue);

  const doSomething = () => {
    // Custom logic
  };

  return [value, doSomething];
}
```

37. What is the Difference Between Shadow DOM and Virtual DOM?

Shadow DOM: - Browser technology for encapsulating element styling - Creates a separate DOM tree - Used in Web Components

Virtual DOM: - React-specific concept - Lightweight copy of actual DOM - Used for efficient rendering and updates

38. Explain React Fiber

React Fiber is a complete rewrite of React's reconciliation algorithm: - Allows rendering work to be split into chunks - Pauses, aborts, or reuses work as new updates come in - Improves performance for complex applications

39. What are Pure Components?

Pure Components only re-render when props or state change:

```jsx
class MyPureComponent extends React.PureComponent {
  render() {
    return <div>{this.props.value}</div>;
  }
}
```

40. Explain React.forwardRef

`forwardRef` allows passing refs through components:

```jsx
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// You can now get a ref directly to the DOM button
const ref = useRef(null);
<FancyButton ref={ref}>Click me!</FancyButton>
```

Performance Optimization

41. How to Optimize React Application Performance?

- Use React.memo for functional components
- Implement shouldComponentUpdate
- Use useMemo and useCallback
- Code splitting
- Lazy loading
- Avoid inline function definitions
- Use production build
- Minimize state

42. What is Code Splitting?

Technique to split your code into smaller chunks:

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <React.Suspense fallback={<div>Loading...</div>}>
      <OtherComponent />
    </React.Suspense>
  );
}
```

43. Explain Lazy Loading

Lazy loading defers loading of non-critical resources:

```
import React, { lazy, Suspense } from 'react';

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

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

44. What is Debouncing in React?

Debouncing limits the rate of function calls:

```
function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);
```

```
  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}
```

45. Explain React Profiler

React Profiler helps measure rendering performance:

```
function App() {
  const onRender = (
    id,
    phase,
    actualDuration,
    baseDuration,
    startTime,
    commitTime
  ) => {
    console.log(`Rendering ${id}`);
  };

  return (
    <React.Profiler id="App" onRender={onRender}>
      <MyComponent />
    </React.Profiler>
  );
}
```

(The document continues with more questions…)

Advanced React Patterns and Techniques

46. What is the Compound Component Pattern?

Allows creating flexible and reusable components:

```
function Tabs({ children }) {
  const [activeTab, setActiveTab] = useState(0);
```

```
  return React.Children.map(children, (child, index) =>
    React.cloneElement(child, {
      isActive: index === activeTab,
      onActivate: () => setActiveTab(index)
    })
  );
}

function Tab({ children, isActive, onActivate }) {
  return (
    <div
      onClick={onActivate}
      style={{ fontWeight: isActive ? 'bold' : 'normal' }}
    >
      {children}
    </div>
  );
}

function App() {
  return (
    <Tabs>
      <Tab>First Tab</Tab>
      <Tab>Second Tab</Tab>
    </Tabs>
  );
}
```

47. Explain the Render Props Pattern Advanced Usage

More complex implementation of render props:

```
class DataFetcher extends React.Component {
  state = {
    data: null,
    loading: true,
    error: null
  };

  componentDidMount() {
    this.fetchData();
  }

  fetchData = async () => {
    try {
      const response = await fetch(this.props.url);
```

```
      const data = await response.json();
      this.setState({ data, loading: false });
    } catch (error) {
      this.setState({ error, loading: false });
    }
  };

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

function App() {
  return (
    <DataFetcher
      url="/api/data"
      render={({ data, loading, error }) => {
        if (loading) return <div>Loading...</div>;
        if (error) return <div>Error: {error.message}</div>;
        return <div>{JSON.stringify(data)}</div>;
      }}
    />
  );
}
```

48. What are React Hooks Limitations?

- Cannot be used in class components
- Must be called at the top level of functional components
- Cannot be conditional
- Performance overhead for complex hooks
- Potential issues with closure capture

49. Explain Advanced useEffect Patterns

```
function useDebugInformation(componentName, props) {
  const count = useRef(0);

  useEffect(() => {
    count.current += 1;
  });

  useEffect(() => {
    console.log(`[${componentName}] Changed props:`, props);
  }, [props]);
```

```
  return { renderCount: count.current };
}
```

50. What is the Proxy Pattern in React?

Creating a wrapper component to add additional functionality:

```
function withLogging(WrappedComponent) {
  return function LoggingComponent(props) {
    useEffect(() => {
      console.log('Component mounted with props:', props);
    }, []);

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

const EnhancedComponent = withLogging(MyComponent);
```

51. Advanced State Management with useReducer

Complex state management example:

```
const initialState = {
  users: [],
  loading: false,
  error: null
};

function userReducer(state, action) {
  switch (action.type) {
    case 'FETCH_USERS_START':
      return { ...state, loading: true };
    case 'FETCH_USERS_SUCCESS':
      return {
        ...state,
        loading: false,
        users: action.payload
      };
    case 'FETCH_USERS_ERROR':
      return {
        ...state,
        loading: false,
        error: action.payload
      };
    default:
      return state;
```

```
  }
}

function UserList() {
  const [state, dispatch] = useReducer(userReducer, initialState);

  useEffect(() => {
    dispatch({ type: 'FETCH_USERS_START' });
    fetchUsers()
      .then(users =>
        dispatch({
          type: 'FETCH_USERS_SUCCESS',
          payload: users
        })
      )
      .catch(error =>
        dispatch({
          type: 'FETCH_USERS_ERROR',
          payload: error
        })
      );
  }, []);

  // Render logic
}
```

Testing in React

52. What are the Common Testing Libraries for React?

- Jest
- React Testing Library
- Enzyme
- Cypress
- Mocha
- Jasmine

53. How to Write Unit Tests in React?

```
import { render, screen, fireEvent } from '@testing-library/react';
import Button from './Button';

test('renders button with correct text', () => {
  render(<Button label="Click me" />);
  expect(screen.getByText('Click me')).toBeInTheDocument();
});
```

```
test('calls onClick prop when clicked', () => {
  const handleClick = jest.fn();
  render(<Button onClick={handleClick} label="Click me" />);

  fireEvent.click(screen.getByText('Click me'));
  expect(handleClick).toHaveBeenCalledTimes(1);
});
```

54. Explain React Testing Best Practices

- Test behavior, not implementation
- Use meaningful test descriptions
- Cover different scenarios (happy path, edge cases)
- Mock external dependencies
- Use snapshot testing sparingly
- Aim for high code coverage
- Write clean, readable tests

55. What is Snapshot Testing?

```
import renderer from 'react-test-renderer';

test('component renders correctly', () => {
  const tree = renderer
    .create(<MyComponent prop1="value" />)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```


Performance and Optimization

56. What is Memoization in Depth?

```
const ExpensiveComponent = React.memo(
  function MyComponent(props) {
    // Expensive rendering logic
  },
  (prevProps, nextProps) => {
    // Custom comparison logic
    return prevProps.value === nextProps.value;
  }
);
```

57. Explain Code Splitting Techniques

- Route-based splitting
- Component-based splitting

- Library splitting

```
// Route-based splitting
const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </Suspense>
  );
}
```

58. What are React Concurrent Mode Concepts?

- Prioritizing updates
- Interruptible rendering
- Smooth user experiences
- Better resource management

59. Explain Performance Profiling

```
function App() {
  return (
    <React.Profiler id="Navigation" onRender={onRenderCallback}>
      <Navigation />
    </React.Profiler>
  );
}

function onRenderCallback(
  id, // the "id" prop of the Profiler tree
  phase, // either "mount" or "update"
  actualDuration, // time spent rendering the committed update
  baseDuration, // estimated time to render the entire subtree without memoization
  startTime, // when React began rendering
  commitTime, // when React committed the update
  interactions // the Set of interactions belonging to this update
) {
  // Log or analyze performance
  console.log(`${id}'s render took ${actualDuration}ms`);
}
```

60. Advanced React Performance Techniques

- Minimize state
- Use immutable data structures
- Avoid unnecessary rerenders
- Optimize context usage
- Lazy load components
- Use web workers for complex computations

Security Considerations

61. How to Prevent XSS in React?

- Use `DOMPurify` to sanitize inputs
- Avoid `dangerouslySetInnerHTML`
- Encode user inputs
- Use built-in React escaping

```
function SafeComponent({ userInput }) {
  // React automatically escapes values
  return <div>{userInput}</div>;
}
```

62. What are Common React Security Vulnerabilities?

- Cross-Site Scripting (XSS)
- Injection attacks
- Insecure direct object references
- Cross-Site Request Forgery (CSRF)
- Unauthorized access

State Management

63. Compare Different State Management Solutions

- Redux
- MobX
- Recoil
- Context API
- Zustand

64. Implement Custom State Management Hook

```
function useCustomState(initialState) {
  const [state, setState] = useState(initialState);

  const updateState = useCallback((newState) => {
    if (typeof newState === 'function') {
```

```
      setState(prevState => newState(prevState));
    } else {
      setState(newState);
    }
  }, []);

  return [state, updateState];
}
```

65. What is the Flux Architecture?

- Unidirectional data flow
- Centralized store
- Actions and dispatchers
- Predictable state management

Advanced React Concepts

66. Explain React Hydration

- Process of making server-rendered markup interactive
- Attaching event listeners
- Matching client and server rendered content

67. What are React Server Components?

- Render on the server
- Reduce client-side JavaScript
- Improve performance
- Seamless integration with existing React code

68. Explain React Concurrent Features

- Prioritized rendering
- Interruptible updates
- Smooth user experiences

69. What is the Difference Between Imperative and Declarative Approaches?

Imperative: - Step-by-step instructions - Describe how to do something

Declarative: - Describe desired outcome - React handles implementation details

70. Advanced Component Composition Techniques

- Higher-Order Components

- Render Props
- Hooks
- Compound Components