React Lists, Keys, and Hooks

Lists and Keys

Question 1: How do you render a list of items in React? Why is it important to use keys when rendering lists?

Rendering Lists: You render lists in React using the map() method to transform an array of data into an array of JSX elements:

```
const fruits = ['apple', 'banana', 'orange'];
const FruitList = () => {
return (
<ul>
{fruits.map((fruit, index) => (
<li key={index}>{fruit}</li>
))}
</ul>
);
};
```

Why Keys are Important:

Performance Optimization: Keys help React identify which items have changed, been added, or removed

Efficient Virtual DOM Diffing: React uses keys to match elements between renders, avoiding unnecessary re-creation of DOM nodes

Preserve Component State: Keys help maintain component state and focus when list items are reordered

Prevent Rendering Issues: Without keys, React may incorrectly update elements, leading to bugs

Question 2: What are keys in React, and what happens if you do not provide a unique key?

What are Keys: Keys are special string attributes that uniquely identify elements in a list. They should be stable, predictable, and unique among siblings.

Best Practices for Keys:

```
// Good - using unique ID

{users.map(user => <User key={user.id} user={user} />)}

// Avoid - using array index (problematic for dynamic lists)

{users.map((user, index) => <User key={index} user={user} />)}
```

What Happens Without Unique Keys:

Performance Issues: React can't efficiently update the DOM

State Bugs: Component state may be incorrectly preserved or lost

Rendering Problems: List items may display wrong data after reordering

Console Warnings: React will show warnings in development mode

Incorrect Form Behavior: Input values may stick to wrong items

React Hooks

Question 1: What are React hooks? How do useState() and useEffect() hooks work in functional components?

What are React Hooks: Hooks are functions that let you "hook into" React features from functional components. They allow you to use state and lifecycle methods without writing class components.

useState() Hook:

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

const Counter = () => {

const [count, setCount] = useState(0); // Initial state is 0
return (
<div>
<p>Count: {count}</p>
<button onClick={() => setCount(count +
1)}>Increment</button>
<button onClick={() => setCount(prev => prev -
1)}>Decrement</button>
</div>
);
};
```

Returns an array with current state value and setter function

Setter can accept a new value or a function that receives previous state

State updates are asynchronous and may be batched

useEffect() Hook:

```
import React, { useState, useEffect } from 'react';
const UserProfile = ({ userId }) => {
const [user, setUser] = useState(null);
// Effect runs after every render
useEffect(() => {
fetchUser(userId).then(setUser);
}, [userId]); // Dependency array - effect runs when userId changes
// Cleanup effect
useEffect(() => {
const timer = setInterval(() => console.log('Timer'), 1000);
return () => clearInterval(timer); // Cleanup function
}, []);
return <div>{user?.name}</div>;
};
```

Question 2: What problems did hooks solve in React development? Why are hooks considered an important addition to React?

Problems Hooks Solved:

1. Complex Class Components: Eliminated need for complex class syntax and this binding

2. Code Reusability: Made it easier to share stateful logic between components

3. Lifecycle Method Confusion: Simplified component lifecycle management

4. Wrapper Hell: Reduced need for Higher-Order Components and render props

5. Bundle Size: Functional components with hooks are more tree-shakable

Why Hooks are Important:

Simpler Mental Model: Easier to understand and reason about

Better Code Organization: Logic can be grouped by concern rather than lifecycle

Improved Testing: Easier to test individual pieces of logic

Better Performance: More optimization opportunities

Future-Proof: Foundation for React's concurrent features

Question 3: What is useReducer? How do we use it in React app?

useReducer is a hook for managing complex state logic, similar to Redux but built into React.

When to Use useReducer:

Complex state with multiple sub-values

State transitions depend on previous state

State logic is complex and involves multiple actions

Example:

```
import React, { useReducer } from 'react';
// Reducer function
const counterReducer = (state, action) => {
switch (action.type) {
case 'INCREMENT':
return { count: state.count + 1 };
case 'DECREMENT':
return { count: state.count - 1 };
case 'RESET':
return { count: 0 };
default:
throw new Error(`Unknown action: ${action.type}`);
}
};
const Counter = () => {
```

```
const [state, dispatch] = useReducer(counterReducer, { count:
0 });
return (
<div>
<p>Count: {state.count}</p>
<button onClick={() => dispatch({ type: 'INCREMENT'
})}>+</button>
<button onClick={() => dispatch({ type: 'DECREMENT' })}>-
</button>
<button onClick={() => dispatch({ type: 'RESET'
})}>Reset</button>
</div>
);
};
```

Complex Example with Form:

```
const formReducer = (state, action) => {
switch (action.type) {
case 'SET_FIELD':
return { ...state, [action.field]: action.value };
case 'SET_ERROR':
```

return { ...state, errors: { ...state.errors, [action.field]: action.error } };

case 'RESET':

return { name: '', email: '', errors: {} };

default:

return state;

}

};

const ContactForm = () => {

const [state, dispatch] = useReducer(formReducer, {

name: '',

email: '',

errors: {}

});

// Usage with dispatch actions

};

Question 4: What is the purpose of useCallback & useMemo Hooks?

useCallback Purpose:

Memoizes function references to prevent unnecessary re-renders

Returns a memoized version of the callback that only changes if dependencies change

useMemo Purpose:

Memoizes expensive computations

Only recalculates when dependencies change

Optimizes performance by avoiding redundant calculations

Performance Benefits:

Prevents child component re-renders when props haven't changed

Optimizes expensive operations

Reduces memory allocation for function references

Question 5: What's the	useCallback	useMemo

Difference between useCallback & useMemo Hooks? Aspect		
Purpose	Memoizes functions	Memoizes computed values
Returns	Memoized function reference	Memoized computed value
Use Case	Prevent function recreation	Avoid expensive calculations
Syntax	useCallback(fn, deps)	useMemo(() => computation, deps)