100+ React.js Interview Questions With Answers

Comprehensive React.js Interview Preparation

Master React.js interviews with this comprehensive collection of 100+ questions covering all essential topics. Each question includes detailed explanations, code examples, and best practices to help you succeed in your next interview.

Question 1 BEGINNER

What is useRef and when would you use it?

```
function TextInput() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <>
      <input ref={inputRef} />
      <button onClick={focusInput}>Focus</button>
    </>
  );
}
```

Answer: useRef is a React hook that lets you create a mutable reference that persists across re-renders - without causing the component to re-render when it changes.

Use cases:

- Access DOM elements directly
- Store mutable values that don't trigger re-renders
- Keep track of previous values
- Store interval or timeout IDs

Unlike state, updating a ref doesn't cause re-renders.

Question 2 INTERMEDIATE

Explain code splitting and lazy loading in React

```
// Lazy loading a component
const LazyComponent = React.lazy(() => import('./HeavyComponent'));

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

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

function App() {
  return (
    <Router>
      <Suspense fallback={<Loading />}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
        </Routes>
      </Suspense>
    </Router>
  );
}
```

Explanation: Code splitting splits your bundle into smaller chunks that are loaded on demand. So all the components used in the application are not loaded on initial load of the application. Instead only the component which is currently displayed on the screen is loaded.

Benefits:

- Reduces initial bundle size
- Faster initial page load of the application
- Better performance on slower connections

Suspense provides a loading fallback while the component loads.

Question 3 INTERMEDIATE

What will happen in this code?

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

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1);
    }, 1000);

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

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

Result:

Count increments only to 1 and then stops

Explanation: As there is no dependency added in the useEffect, the effect runs only once, capturing the initial count value (0). The setInterval always uses count = 0, so it keeps setting state to 1 using setCount function.

Fix: Use setCount(prev => prev + 1) or add count to useEffect dependency.

CodeSandbox Demo

Question 4 INTERMEDIATE

What causes unnecessary re-renders and how do you prevent them?

Common causes:

- Creating new object/array references in render
- Inline function definitions passed as props
- Parent component re-rendering
- Context value changes

Prevention techniques:

- **React.memo:** Memoize components
- **useMemo:** Memoize expensive calculations and object references
- **useCallback:** Memoize function references
- **Move state down:** Keep state as close as possible to where it's used
- **Split contexts:** Separate frequently and rarely changing data

Question 5 INTERMEDIATE

What is Reconciliation in React?

Answer: Reconciliation is the algorithm React uses to diff one tree with another to determine which parts of the UI need to be changed.

Process:

1. When state/props change, React creates a new Virtual DOM tree
2. Compares new tree with previous tree (diffing)
3. Calculates minimum operations needed
4. Updates only changed parts in real DOM

Optimization rules:

- Different element types produce different trees
- Adding keys in array map method help identify elements across re-renders
- Siblings are processed in order

Question 6 INTERMEDIATE

What is React Fiber?

Answer: React Fiber is the reconciliation engine introduced in React 16. It's a complete rewrite of React's core algorithm that enables:

- **Incremental rendering:** Split work into chunks
- **Pause, abort, or reuse work:** More control over rendering
- **Priority-based rendering:** Prioritize urgent updates
- **Concurrent features:** Multiple state updates at once

Fiber makes React more efficient and enables features like Suspense and Concurrent Mode.

Question 7 INTERMEDIATE

What is forwardRef?

```
const FancyInput = React.forwardRef((props, ref) => (
  <input ref={ref} className="fancy" {...props} />
));

// Parent can now get a ref to the input
function Parent() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <>
      <FancyInput ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
    </>
  );
}
```

Explanation: forwardRef allows a component to pass a ref to a child component. This is useful when creating reusable component libraries where parent components need direct access to DOM elements. Without forwardRef, passing refs to functional components don't work.

Question 8 BEGINNER

What are Fragments and why use them?

```
// Long syntax
function List() {
  return (
    <React.Fragment>
      <li>Item 1</li>
      <li>Item 2</li>
    </React.Fragment>
  );
}

// Short syntax
function List() {
  return (
    <>
      <li>Item 1</li>
      <li>Item 2</li>
    </>
  );
}

// With key (only long syntax supports key)
function List({ items }) {
  return items.map(item => (
    <React.Fragment key={item.id}>
      <dt>{item.term}</dt>
      <dd>{item.definition}</dd>
    </React.Fragment>
  ));
}
```

Answer: Fragments lets you group multiple children without adding extra nodes to the DOM.

Benefits:

- Cleaner DOM structure
- Better performance (fewer nodes)
- Avoid CSS layout issues (specially when using flexbox)
- Provides semantically valid HTML structure (e.g., table rows)

Question 9 INTERMEDIATE

What is the Container/Presentational pattern?

```
// Container Component (logic)
function UserListContainer() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetchUsers().then(data => {
      setUsers(data);
      setLoading(false);
    });
  }, []);

  return <UserList users={users} loading={loading} />;
}

// Presentational Component (UI)
function UserList({ users, loading }) {
  if (loading) return <Loading />;

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Explanation: Separates concerns.

- **Container:** Handles logic, state, API calls
- **Presentational:** Receives data via props, focuses on displaying UI

Benefits: Reusability, testability, separation of concerns

Modern approach: Custom Hooks often replace container components.

Question 10 BEGINNER

What are some React best practices?

Answer:

- **Keep components small:** Single responsibility principle
- **Use functional components:** With Hooks for newer React applications
- **Lift state up:** Share state at lowest common ancestor
- **Avoid prop drilling:** Use Context or state management
- **Use keys properly:** Stable, unique identifiers
- **Memoize wisely:** Don't over-optimize
- **Keep state close:** Don't lift state unnecessarily
- **Use TypeScript:** Type safety and better developer experience
- **Write tests:** Test behavior, not implementation
- **Code splitting:** Lazy load routes and heavy components

Question 11 INTERMEDIATE

What is a race condition and how do you prevent it?

```
// Problem: Race condition
function SearchResults({ query }) {
  const [results, setResults] = useState([]);

  useEffect(() => {
    fetch(`/api/search?q=${query}`)
      .then(res => res.json())
      .then(data => setResults(data));
      // If query changes quickly, old responses
      // might arrive after new ones
  }, [query]);
}

// Solution 1: Cleanup flag
useEffect(() => {
  let active = true;

  fetch(`/api/search?q=${query}`)
    .then(res => res.json())
    .then(data => {
      if (active) setResults(data);
    });

  return () => { active = false; };
}, [query]);

// Solution 2: AbortController
useEffect(() => {
  const controller = new AbortController();

  fetch(`/api/search?q=${query}`, {
    signal: controller.signal
  })
    .then(res => res.json())
    .then(data => setResults(data))
    .catch(err => {
      if (err.name !== 'AbortError') {
        console.error(err);
      }
    });

  return () => controller.abort();
}, [query]);
```

Explanation: Race conditions occur when async operations complete out of order.

Solutions:

- **Cleanup flag:** Ignore outdated responses
- **AbortController:** Cancel pending requests

- **Debouncing:** Delay requests until user stops typing

Question 12 INTERMEDIATE

What is useReducer hook and when should you use it?

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return { count: 0 };
    default:
      throw new Error('Unknown action');
  }
}

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>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
    </>
  );
}
```

Answer: useReducer is an alternative to useState for complex state logic.

Use when:

- State logic is complex with multiple sub-values
- Next state depends on previous state
- You want to centralize state update logic
- You need to pass dispatch down to child components instead of callbacks

Similar to Redux but local to component.

Question 13 INTERMEDIATE

How do you type hooks with TypeScript?

```
// useState with type inference
const [count, setCount] = useState(0); // inferred as number

// useState with explicit type
const [user, setUser] = useState<User | null>(null);

// useRef
const inputRef = useRef<HTMLInputElement>(null);

// useReducer
type State = { count: number };
type Action =
  | { type: 'increment' }
  | { type: 'decrement' }
  | { type: 'set'; payload: number };

function reducer(state: State, action: Action): State {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'set':
      return { count: action.payload };
  }
}

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

Explanation: TypeScript hooks typing:

- useState often infers type from initial value
- Use generic for complex types or null initial values
- useRef needs element type for DOM refs
- useReducer action types use discriminated unions

Learn More

Question 14 INTERMEDIATE

What's wrong with this code?

```
function Component() {
  const [items, setItems] = useState([1, 2, 3]);

  const addItem = () => {
    items.push(4); // ❌ Wrong!
    setItems(items);
  };

  return <button onClick={addItem}>Add</button>;
}
```

Problem:

Mutating state directly

Explanation: Never mutate state directly! React uses reference equality to detect changes. Since items still has the same reference, React won't re-render.

Correct approach:

```
const addItem = () => {
  setItems([...items, 4]); // Create new array
};
```

Question 15 INTERMEDIATE

Why does this component render infinitely?

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

  useEffect(() => {
    setCount(count + 1);
  }); // ❌ No dependency array!

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

Problem:

Infinite loop

Explanation: Without a dependency array, useEffect runs on every component re-render. Since we're setting state inside useEffect, it triggers another re-render every time useEffect is executed, causing an infinite loop.

Solution: Add proper dependencies or empty array:

```
useEffect(() => {
  setCount(count + 1);
}, []); // Runs only once
```

About Me

Hi, I'm Yogesh,

I'm a Freelancer, Mentor, Full Stack Developer and an Industry/Corporate Trainer.

Specializing in: HTML, JavaScript, CSS, React, Next.js, Node.js, and related web technologies.

I love sharing my knowledge through articles and tutorials.

I usually write at: freeCodeCamp, dev.to, Medium, Hashnode

Check out my all the courses/ebooks/webinars.

Don't forget to follow me on LinkedIn to learn new things everyday.