**React.js Interview Preparation Guide**

I'll create a comprehensive React.js interview preparation guide that covers everything from basics to advanced concepts. This material will help you prepare thoroughly for your React interviews.

**React.js Interview Preparation Guide: Basic to Advanced**

**Table of Contents**

1. [React Fundamentals](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#react-fundamentals)
2. [Component Patterns](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#component-patterns)
3. [State Management](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#state-management)
4. [React Hooks](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#react-hooks)
5. [Performance Optimization](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#performance-optimization)
6. [Testing in React](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#testing-in-react)
7. [React Router](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#react-router)
8. [Server-Side Rendering](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#server-side-rendering)
9. [React Ecosystem](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#react-ecosystem)
10. [Advanced Patterns](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#advanced-patterns)
11. [Interview Questions & Answers](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#interview-questions--answers)
12. [Coding Challenges](https://claude.ai/chat/46c00003-66b5-4183-a563-316af689e26b#coding-challenges)

**React Fundamentals**

**What is React?**

React is a JavaScript library for building user interfaces, particularly single-page applications. It's developed and maintained by Facebook (now Meta) and a community of individual developers and companies.

**Key Features**

* **Component-Based Architecture**: Build encapsulated components that manage their own state, then compose them to make complex UIs
* **Declarative UI**: Design simple views for each state in your application, and React will efficiently update and render the right components when your data changes
* **Virtual DOM**: A lightweight representation of the real DOM that enables React to update the UI efficiently
* **JSX**: A syntax extension to JavaScript that allows you to write HTML-like code in your JavaScript files

**JSX Basics**

JSX is a syntax extension for JavaScript that looks similar to HTML but gets transformed into JavaScript functions.

// JSX syntax

const element = <h1>Hello, world!</h1>;

// Compiles to

const element = React.createElement('h1', null, 'Hello, world!');

Key characteristics:

* Elements must be closed (self-closing or with closing tag)
* You can embed JavaScript expressions using curly braces {}
* Attributes use camelCase naming convention (className instead of class)
* Multiple elements must be wrapped in a parent element or fragment

**Creating React Components**

**Function Components**

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

**Class Components**

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

**Props**

Props (short for properties) are read-only data passed from parent to child components.

// Parent component

function App() {

return <Welcome name="Sara" />;

}

// Child component

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

Props are:

* Immutable (components must never modify their props)
* Can be any JavaScript value (strings, numbers, objects, functions)
* Can be destructured for cleaner code

**Props destructuring**

function Welcome({ name, age }) {

return (

<div>

<h1>Hello, {name}</h1>

<p>You are {age} years old.</p>

</div>

);

}

**State**

State is private data controlled by a component that can change over time.

**Class component state**

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>

);

}

}

**Function component state (using hooks)**

function Counter() {

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

return (

<div>

<p>Count: {count}</p>

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

</div>

);

}

**Lifecycle of Components**

**Class Component Lifecycle Methods**

1. **Mounting**:
   * constructor()
   * static getDerivedStateFromProps()
   * render()
   * componentDidMount()
2. **Updating**:
   * static getDerivedStateFromProps()
   * shouldComponentUpdate()
   * render()
   * getSnapshotBeforeUpdate()
   * componentDidUpdate()
3. **Unmounting**:
   * componentWillUnmount()
4. **Error Handling**:
   * static getDerivedStateFromError()
   * componentDidCatch()

**Function Component Lifecycle using Hooks**

1. **Mounting**: useEffect(() => {}, [])
2. **Updating**: useEffect(() => {})
3. **Unmounting**: useEffect(() => { return () => {} }, [])

**Component Patterns**

**Container vs. Presentational Components**

**Container Components**:

* Focus on how things work
* Provide data and behavior to other components
* Call APIs and handle state
* Often stateful

**Presentational Components**:

* Focus on how things look
* Receive data and callbacks via props
* Rarely have own state (except for UI state)
* Highly reusable

**Composition vs. Inheritance**

React recommends using composition over inheritance for component reuse:

function Dialog(props) {

return (

<div className="dialog">

<h1>{props.title}</h1>

<div>{props.children}</div>

</div>

);

}

function WelcomeDialog() {

return (

<Dialog title="Welcome">

<p>Thank you for visiting our website!</p>

</Dialog>

);

}

**Higher-Order Components (HOC)**

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

function withSubscription(WrappedComponent, selectData) {

return class extends React.Component {

constructor(props) {

super(props);

this.state = {

data: selectData(DataSource, props)

};

}

// Lifecycle methods and data updates

render() {

return <WrappedComponent data={this.state.data} {...this.props} />;

}

};

}

const CommentListWithSubscription = withSubscription(

CommentList,

(DataSource, props) => DataSource.getComments()

);

**Render Props**

A technique for sharing code between components using a prop whose value is a function.

class Mouse extends React.Component {

state = { x: 0, y: 0 };

handleMouseMove = (event) => {

this.setState({

x: event.clientX,

y: event.clientY

});

};

render() {

return (

<div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>

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

</div>

);

}

}

function App() {

return (

<Mouse render={mouse => (

<p>The mouse position is {mouse.x}, {mouse.y}</p>

)}/>

);

}

**Compound Components**

Components that work together to form a complete UI.

function Tabs({ children, defaultIndex = 0 }) {

const [activeIndex, setActiveIndex] = useState(defaultIndex);

return React.Children.map(children, (child, index) => {

return React.cloneElement(child, {

active: index === activeIndex,

setActiveIndex: () => setActiveIndex(index)

});

});

}

function Tab({ active, setActiveIndex, children }) {

return (

<button

style={{ fontWeight: active ? 'bold' : 'normal' }}

onClick={setActiveIndex}

>

{children}

</button>

);

}

function TabPanel({ active, children }) {

return active ? <div>{children}</div> : null;

}

// Usage

function App() {

return (

<div>

<Tabs>

<Tab>Tab 1</Tab>

<TabPanel>Content for Tab 1</TabPanel>

<Tab>Tab 2</Tab>

<TabPanel>Content for Tab 2</TabPanel>

</Tabs>

</div>

);

}

**State Management**

**Local Component State**

Managing state within a component using useState or class component state.

function Counter() {

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

return (

<div>

<p>Count: {count}</p>

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

</div>

);

}

**Lifting State Up**

Moving shared state to the closest common ancestor of components that need it.

function ParentComponent() {

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

return (

<div>

<DisplayCount count={count} />

<ButtonControls onIncrement={() => setCount(count + 1)} />

</div>

);

}

function DisplayCount({ count }) {

return <p>Count: {count}</p>;

}

function ButtonControls({ onIncrement }) {

return <button onClick={onIncrement}>Increment</button>;

}

**Context API**

Providing a way to share values like themes, user data, or locale between components without explicitly passing props.

// Create a context

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

function App() {

return (

<ThemeContext.Provider value="dark">

<Toolbar />

</ThemeContext.Provider>

);

}

function Toolbar() {

return <ThemedButton />;

}

function ThemedButton() {

const theme = useContext(ThemeContext);

return <button className={theme}>Themed Button</button>;

}

**Redux**

A predictable state container for JavaScript apps:

// Action

const increment = () => {

return {

type: 'INCREMENT'

};

};

// Reducer

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

switch (action.type) {

case 'INCREMENT':

return state + 1;

default:

return state;

}

};

// Store

const store = createStore(counterReducer);

// React Component with Redux

function Counter({ count, increment }) {

return (

<div>

<p>Count: {count}</p>

<button onClick={increment}>Increment</button>

</div>

);

}

const mapStateToProps = (state) => {

return {

count: state

};

};

const mapDispatchToProps = (dispatch) => {

return {

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

};

};

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

**React Query**

A library for fetching, caching, and updating server state in React:

function Users() {

const { data, isLoading, error } = useQuery('users', fetchUsers);

if (isLoading) return <p>Loading...</p>;

if (error) return <p>Error: {error.message}</p>;

return (

<ul>

{data.map(user => (

<li key={user.id}>{user.name}</li>

))}

</ul>

);

}

**Zustand**

A small, fast state management solution:

import create from 'zustand';

const useStore = create(set => ({

count: 0,

increment: () => set(state => ({ count: state.count + 1 })),

}));

function Counter() {

const { count, increment } = useStore();

return (

<div>

<p>Count: {count}</p>

<button onClick={increment}>Increment</button>

</div>

);

}

**Recoil**

A state management library for React:

import { atom, useRecoilState } from 'recoil';

const countState = atom({

key: 'countState',

default: 0,

});

function Counter() {

const [count, setCount] = useRecoilState(countState);

return (

<div>

<p>Count: {count}</p>

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

</div>

);

}

**React Hooks**

**Basic Hooks**

**useState**

function Counter() {

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

// Functional updates for state that depends on previous state

const increment = () => {

setCount(prevCount => prevCount + 1);

};

return (

<div>

<p>Count: {count}</p>

<button onClick={increment}>Increment</button>

</div>

);

}

**useEffect**

function UserStatus({ userId }) {

const [isOnline, setIsOnline] = useState(null);

useEffect(() => {

// Setup effect

const handleStatusChange = (status) => {

setIsOnline(status.isOnline);

};

UserAPI.subscribeToUserStatus(userId, handleStatusChange);

// Cleanup function

return () => {

UserAPI.unsubscribeFromUserStatus(userId, handleStatusChange);

};

}, [userId]); // Only re-run if userId changes

if (isOnline === null) return <p>Loading...</p>;

return <p>User is {isOnline ? 'online' : 'offline'}</p>;

}

**useContext**

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

function ThemedButton() {

const theme = useContext(ThemeContext);

return <button className={theme}>Themed Button</button>;

}

**Additional Hooks**

**useReducer**

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();

}

}

function Counter() {

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

return (

<div>

<p>Count: {state.count}</p>

<button onClick={() => dispatch({ type: 'increment' })}>+</button>

<button onClick={() => dispatch({ type: 'decrement' })}>-</button>

</div>

);

}

**useCallback**

function ParentComponent() {

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

// This function is memoized and only changes if dependencies change

const handleClick = useCallback(() => {

setCount(c => c + 1);

}, []); // Empty dependency array means this function never changes

return <ChildComponent onClick={handleClick} />;

}

// Using React.memo to prevent unnecessary renders

const ChildComponent = React.memo(function ChildComponent({ onClick }) {

console.log("Child rendered");

return <button onClick={onClick}>Increment</button>;

});

**useMemo**

function ExpensiveComponent({ a, b }) {

// Result is memoized and only recalculated when a or b changes

const result = useMemo(() => {

console.log("Computing expensive result");

return computeExpensiveValue(a, b);

}, [a, b]);

return <p>Result: {result}</p>;

}

**useRef**

function TextInputWithFocusButton() {

const inputRef = useRef(null);

const focusInput = () => {

// Direct access to DOM node

inputRef.current.focus();

};

return (

<div>

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

<button onClick={focusInput}>Focus Input</button>

</div>

);

}

**useLayoutEffect**

Similar to useEffect, but fires synchronously after all DOM mutations.

function Tooltip() {

const [tooltipHeight, setTooltipHeight] = useState(0);

const tooltipRef = useRef();

// Runs synchronously after DOM mutations but before browser paint

useLayoutEffect(() => {

const height = tooltipRef.current.getBoundingClientRect().height;

setTooltipHeight(height);

}, []);

return (

<div style={{ marginTop: -tooltipHeight }}>

<div ref={tooltipRef}>Tooltip content</div>

</div>

);

}

**useImperativeHandle**

function FancyInput(props, ref) {

const inputRef = useRef();

useImperativeHandle(ref, () => ({

focus: () => {

inputRef.current.focus();

},

value: () => {

return inputRef.current.value;

}

}));

return <input ref={inputRef} type="text" />;

}

// Create forwarded ref component

FancyInput = React.forwardRef(FancyInput);

// Parent component

function Parent() {

const fancyInputRef = useRef();

const focusInput = () => {

fancyInputRef.current.focus();

console.log(fancyInputRef.current.value());

};

return (

<div>

<FancyInput ref={fancyInputRef} />

<button onClick={focusInput}>Focus Input</button>

</div>

);

}

**Custom Hooks**

Creating reusable hooks for shared logic:

// Custom hook for form handling

function useForm(initialValues) {

const [values, setValues] = useState(initialValues);

const handleChange = (e) => {

const { name, value } = e.target;

setValues({

...values,

[name]: value

});

};

const resetForm = () => {

setValues(initialValues);

};

return { values, handleChange, resetForm };

}

// Using the custom hook

function SignupForm() {

const { values, handleChange, resetForm } = useForm({

username: '',

email: '',

password: ''

});

const handleSubmit = (e) => {

e.preventDefault();

// Submit form

console.log(values);

resetForm();

};

return (

<form onSubmit={handleSubmit}>

<input

name="username"

value={values.username}

onChange={handleChange}

placeholder="Username"

/>

<input

name="email"

value={values.email}

onChange={handleChange}

placeholder="Email"

/>

<input

type="password"

name="password"

value={values.password}

onChange={handleChange}

placeholder="Password"

/>

<button type="submit">Sign Up</button>

</form>

);

}

**Performance Optimization**

**React.memo**

Memoizing functional components to prevent unnecessary renders:

const MyComponent = React.memo(function MyComponent(props) {

/\* render using props \*/

});

**useMemo for Expensive Calculations**

function TodoList({ todos, filter }) {

// Memoize filtered todos

const filteredTodos = useMemo(() => {

console.log("Filtering todos");

return todos.filter(todo => todo.type === filter);

}, [todos, filter]);

return (

<ul>

{filteredTodos.map(todo => (

<li key={todo.id}>{todo.text}</li>

))}

</ul>

);

}

**useCallback for Stable Event Handlers**

function ParentComponent() {

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

// Stable function reference that only changes when dependencies change

const increment = useCallback(() => {

setCount(c => c + 1);

}, []);

return <ChildComponent onIncrement={increment} />;

}

**Code Splitting with React.lazy and Suspense**

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

function MyComponent() {

return (

<div>

<Suspense fallback={<div>Loading...</div>}>

<OtherComponent />

</Suspense>

</div>

);

}

**Virtualization**

Using libraries like react-window or react-virtualized for large lists:

import { FixedSizeList } from 'react-window';

function ListComponent({ items }) {

const Row = ({ index, style }) => (

<div style={style}>

Item {items[index]}

</div>

);

return (

<FixedSizeList

height={400}

width={300}

itemSize={50}

itemCount={items.length}

>

{Row}

</FixedSizeList>

);

}

**Using the useTransition Hook**

function SearchResults() {

const [query, setQuery] = useState('');

const [results, setResults] = useState([]);

const [isPending, startTransition] = useTransition();

const handleChange = (e) => {

const value = e.target.value;

setQuery(value);

// Mark this state update as low priority

startTransition(() => {

// Expensive operation

setResults(computeSearchResults(value));

});

};

return (

<div>

<input value={query} onChange={handleChange} />

{isPending ? (

<p>Loading...</p>

) : (

<ul>

{results.map(result => (

<li key={result.id}>{result.text}</li>

))}

</ul>

)}

</div>

);

}

**useDeferredValue**

function SearchResults({ query }) {

// Defer updating the list until after other UI updates

const deferredQuery = useDeferredValue(query);

// This will use the deferred value

const results = useSearch(deferredQuery);

return (

<ul>

{results.map(result => (

<li key={result.id}>{result.text}</li>

))}

</ul>

);

}

**Profiling with React DevTools**

Using the React DevTools Profiler to identify performance bottlenecks:

* Record renders
* Identify components that render too often
* See which props cause re-renders
* Measure render times

**Testing in React**

**Jest and React Testing Library**

Basic component test:

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

import Counter from './Counter';

test('increments count when button is clicked', () => {

// Render component

render(<Counter />);

// Get elements

const counterValue = screen.getByText(/count: 0/i);

const incrementButton = screen.getByRole('button', { name: /increment/i });

// Interact with elements

fireEvent.click(incrementButton);

// Assert

expect(screen.getByText(/count: 1/i)).toBeInTheDocument();

});

**Testing Hooks**

import { renderHook, act } from '@testing-library/react-hooks';

import useCounter from './useCounter';

test('should increment counter', () => {

const { result } = renderHook(() => useCounter());

act(() => {

result.current.increment();

});

expect(result.current.count).toBe(1);

});

**Mocking API Calls**

import { render, screen, waitFor } from '@testing-library/react';

import UserList from './UserList';

import \* as api from './api'; // Import the API module

// Mock the API module

jest.mock('./api');

test('renders users after API call', async () => {

// Mock implementation

api.fetchUsers.mockResolvedValue([

{ id: 1, name: 'John' },

{ id: 2, name: 'Jane' }

]);

render(<UserList />);

// Check loading state

expect(screen.getByText(/loading/i)).toBeInTheDocument();

// Wait for users to be displayed

await waitFor(() => {

expect(screen.getByText('John')).toBeInTheDocument();

expect(screen.getByText('Jane')).toBeInTheDocument();

});

});

**Snapshot Testing**

import { render } from '@testing-library/react';

import Button from './Button';

test('button matches snapshot', () => {

const { container } = render(<Button label="Click me" />);

expect(container).toMatchSnapshot();

});

**Testing Context**

import { render, screen } from '@testing-library/react';

import { ThemeContext } from './ThemeContext';

import ThemedButton from './ThemedButton';

test('uses the correct theme', () => {

render(

<ThemeContext.Provider value="dark">

<ThemedButton />

</ThemeContext.Provider>

);

const button = screen.getByRole('button');

expect(button).toHaveClass('dark');

});

**React Router**

**Basic Setup**

import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

function App() {

return (

<BrowserRouter>

<nav>

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

<Link to="/about">About</Link>

<Link to="/dashboard">Dashboard</Link>

</nav>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

<Route path="/dashboard" element={<Dashboard />} />

</Routes>

</BrowserRouter>

);

}

**Route Parameters**

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/users" element={<UserList />} />

<Route path="/users/:id" element={<UserDetails />} />

</Routes>

</BrowserRouter>

);

}

function UserDetails() {

const { id } = useParams();

return <h1>User details for user {id}</h1>;

}

**Nested Routes**

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Layout />}>

<Route index element={<Home />} />

<Route path="about" element={<About />} />

<Route path="dashboard" element={<Dashboard />} />

</Route>

</Routes>

</BrowserRouter>

);

}

function Layout() {

return (

<div>

<nav>

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

<Link to="/about">About</Link>

<Link to="/dashboard">Dashboard</Link>

</nav>

<Outlet /> {/\* Nested routes render here \*/}

</div>

);

}

**Protected Routes**

function ProtectedRoute({ children }) {

const { user } = useAuth();

if (!user) {

// Redirect to login if not authenticated

return <Navigate to="/login" replace />;

}

return children;

}

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

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

<Route

path="/dashboard"

element={

<ProtectedRoute>

<Dashboard />

</ProtectedRoute>

}

/>

</Routes>

</BrowserRouter>

);

}

**Programmatic Navigation**

function LoginForm() {

const navigate = useNavigate();

const [formData, setFormData] = useState({ username: '', password: '' });

const handleSubmit = async (e) => {

e.preventDefault();

try {

await login(formData);

navigate('/dashboard'); // Navigate on successful login

} catch (error) {

console.error('Login failed', error);

}

};

// Form JSX

}

**Location State**

// Sending state

<Link

to="/product"

state={{ referrer: 'home', productId: 123 }}

>

View Product

</Link>

// Or programmatically

navigate('/product', { state: { referrer: 'home', productId: 123 } });

// Receiving state

function Product() {

const location = useLocation();

const { referrer, productId } = location.state || {};

return (

<div>

<p>Product ID: {productId}</p>

<p>Referred from: {referrer}</p>

</div>

);

}

**Server-Side Rendering**

**Next.js Basics**

// pages/index.js

export default function Home() {

return (

<div>

<h1>Welcome to Next.js</h1>

</div>

);

}

// pages/about.js

export default function About() {

return (

<div>

<h1>About Us</h1>

</div>

);

}

**Data Fetching in Next.js**

**getServerSideProps**

// Server-side rendering with data fetching

export async function getServerSideProps() {

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

const data = await res.json();

return {

props: { data }, // Will be passed to the page component as props

};

}

export default function Page({ data }) {

return (

<div>

<h1>Data from API</h1>

<pre>{JSON.stringify(data, null, 2)}</pre>

</div>

);

}

**getStaticProps**

// Static Generation with data

export async function getStaticProps() {

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

const data = await res.json();

return {

props: { data },

revalidate: 60, // Regenerate page on server at most once every 60 seconds

};

}

export default function Page({ data }) {

return (

<div>

<h1>Static Data</h1>

<pre>{JSON.stringify(data, null, 2)}</pre>

</div>

);

}

**getStaticPaths**

// Define which paths to pre-render

export async function getStaticPaths() {

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

const posts = await res.json();

const paths = posts.map((post) => ({

params: { id: post.id.toString() },

}));

return {

paths,

fallback: 'blocking', // Generate missing pages on-demand

};

}

export async function getStaticProps({ params }) {

const res = await fetch(`https://api.example.com/posts/${params.id}`);

const post =

const post = await res.json();

return { props: { post }, revalidate: 60, }; }

export default function Post({ post }) { return ( <div> <h1>{post.title}</h1> <p>{post.content}</p> </div> ); }

### API Routes in Next.js

```jsx

// pages/api/users.js

export default function handler(req, res) {

if (req.method === 'GET') {

// Handle GET request

res.status(200).json({ users: ['John', 'Jane'] });

} else if (req.method === 'POST') {

// Handle POST request

const { name } = req.body;

res.status(201).json({ message: `User ${name} created` });

} else {

res.setHeader('Allow', ['GET', 'POST']);

res.status(405).end(`Method ${req.method} Not Allowed`);

}

}

**Client-Side Navigation**

import Link from 'next/link';

import { useRouter } from 'next/router';

function Navigation() {

const router = useRouter();

const handleClick = () => {

router.push('/dashboard');

};

return (

<nav>

<Link href="/">Home</Link>

<Link href="/about">About</Link>

<button onClick={handleClick}>Dashboard</button>

</nav>

);

}

**React Ecosystem**

**Styling Solutions**

**CSS Modules**

// Button.module.css

.button {

background-color: blue;

color: white;

padding: 10px 20px;

border-radius: 4px;

}

.primary {

background-color: green;

}

// Button.jsx

import styles from './Button.module.css';

function Button({ primary, children }) {

const className = `${styles.button} ${primary ? styles.primary : ''}`;

return <button className={className}>{children}</button>;

}

**Styled Components**

import styled from 'styled-components';

const Button = styled.button`

background-color: ${props => props.primary ? 'green' : 'blue'};

color: white;

padding: 10px 20px;

border-radius: 4px;

&:hover {

opacity: 0.8;

}

`;

function App() {

return (

<div>

<Button>Normal Button</Button>

<Button primary>Primary Button</Button>

</div>

);

}

**Emotion**

/\*\* @jsx jsx \*/

import { css, jsx } from '@emotion/react';

function Button({ primary, children }) {

return (

<button

css={css`

background-color: ${primary ? 'green' : 'blue'};

color: white;

padding: 10px 20px;

border-radius: 4px;

&:hover {

opacity: 0.8;

}

`}

>

{children}

</button>

);

}

**Tailwind CSS**

function Button({ primary, children }) {

const className = primary

? 'bg-green-500 hover:bg-green-600'

: 'bg-blue-500 hover:bg-blue-600';

return (

<button className={`${className} text-white px-4 py-2 rounded`}>

{children}

</button>

);

}

**Form Libraries**

**Formik**

import { Formik, Form, Field, ErrorMessage } from 'formik';

import \* as Yup from 'yup';

// Validation schema

const SignupSchema = Yup.object().shape({

name: Yup.string()

.min(2, 'Too Short!')

.max(50, 'Too Long!')

.required('Required'),

email: Yup.string()

.email('Invalid email')

.required('Required'),

password: Yup.string()

.min(8, 'Password must be at least 8 characters')

.required('Required'),

});

function SignupForm() {

return (

<Formik

initialValues={{ name: '', email: '', password: '' }}

validationSchema={SignupSchema}

onSubmit={(values, { setSubmitting }) => {

setTimeout(() => {

alert(JSON.stringify(values, null, 2));

setSubmitting(false);

}, 400);

}}

>

{({ isSubmitting }) => (

<Form>

<div>

<label htmlFor="name">Name</label>

<Field name="name" />

<ErrorMessage name="name" component="div" className="error" />

</div>

<div>

<label htmlFor="email">Email</label>

<Field name="email" type="email" />

<ErrorMessage name="email" component="div" className="error" />

</div>

<div>

<label htmlFor="password">Password</label>

<Field name="password" type="password" />

<ErrorMessage name="password" component="div" className="error" />

</div>

<button type="submit" disabled={isSubmitting}>

{isSubmitting ? 'Submitting...' : 'Submit'}

</button>

</Form>

)}

</Formik>

);

}

**React Hook Form**

import { useForm } from 'react-hook-form';

import { yupResolver } from '@hookform/resolvers/yup';

import \* as yup from 'yup';

const schema = yup.object().shape({

name: yup.string().required('Name is required'),

email: yup.string().email('Invalid email').required('Email is required'),

password: yup.string()

.min(8, 'Password must be at least 8 characters')

.required('Password is required'),

});

function SignupForm() {

const { register, handleSubmit, formState: { errors, isSubmitting } } = useForm({

resolver: yupResolver(schema)

});

const onSubmit = data => {

return new Promise(resolve => {

setTimeout(() => {

alert(JSON.stringify(data, null, 2));

resolve();

}, 1000);

});

};

return (

<form onSubmit={handleSubmit(onSubmit)}>

<div>

<label htmlFor="name">Name</label>

<input id="name" {...register('name')} />

{errors.name && <p className="error">{errors.name.message}</p>}

</div>

<div>

<label htmlFor="email">Email</label>

<input id="email" type="email" {...register('email')} />

{errors.email && <p className="error">{errors.email.message}</p>}

</div>

<div>

<label htmlFor="password">Password</label>

<input id="password" type="password" {...register('password')} />

{errors.password && <p className="error">{errors.password.message}</p>}

</div>

<button type="submit" disabled={isSubmitting}>

{isSubmitting ? 'Submitting...' : 'Submit'}

</button>

</form>

);

}

**UI Component Libraries**

* **Material-UI (MUI)**: React components based on Material Design
* **Chakra UI**: Simple, modular and accessible component library
* **Ant Design**: Enterprise-class UI design language and React implementation
* **Tailwind UI**: Pre-designed components built with Tailwind CSS
* **React Bootstrap**: Bootstrap components built with React
* **Semantic UI React**: The official React integration for Semantic UI

**Animation Libraries**

**React Spring**

import { useSpring, animated } from 'react-spring';

function AnimatedComponent() {

const [flipped, setFlipped] = useState(false);

const { opacity, transform } = useSpring({

opacity: flipped ? 1 : 0,

transform: `perspective(600px) rotateX(${flipped ? 180 : 0}deg)`,

config: { mass: 5, tension: 500, friction: 80 }

});

return (

<div onClick={() => setFlipped(!flipped)}>

<animated.div

style={{

opacity: opacity.to(o => 1 - o),

transform

}}

>

Click to flip

</animated.div>

<animated.div

style={{

opacity,

transform: transform.to(t => `${t} rotateX(180deg)`)

}}

>

Hello from the other side

</animated.div>

</div>

);

}

**Framer Motion**

import { motion } from 'framer-motion';

function AnimatedBox() {

return (

<motion.div

initial={{ opacity: 0, scale: 0.8 }}

animate={{ opacity: 1, scale: 1 }}

transition={{ duration: 0.5 }}

whileHover={{ scale: 1.1 }}

whileTap={{ scale: 0.9 }}

style={{

width: 100,

height: 100,

background: 'blue',

borderRadius: 10

}}

/>

);

}

**Advanced Patterns**

**Render Props with Typescript**

interface RenderProps<T> {

data: T;

loading: boolean;

error: Error | null;

refetch: () => void;

}

interface DataFetcherProps<T> {

url: string;

render: (props: RenderProps<T>) => React.ReactNode;

}

function DataFetcher<T>({ url, render }: DataFetcherProps<T>) {

const [data, setData] = useState<T | null>(null);

const [loading, setLoading] = useState<boolean>(true);

const [error, setError] = useState<Error | null>(null);

const fetchData = async () => {

setLoading(true);

try {

const response = await fetch(url);

const json = await response.json();

setData(json);

setError(null);

} catch (err) {

setError(err instanceof Error ? err : new Error(String(err)));

} finally {

setLoading(false);

}

};

useEffect(() => {

fetchData();

}, [url]);

return (

<>

{render({

data: data as T,

loading,

error,

refetch: fetchData

})}

</>

);

}

// Usage

function UserList() {

return (

<DataFetcher<User[]>

url="/api/users"

render={({ data, loading, error, refetch }) => {

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

if (error) return <p>Error: {error.message}</p>;

return (

<div>

<button onClick={refetch}>Refresh</button>

<ul>

{data.map(user => (

<li key={user.id}>{user.name}</li>

))}

</ul>

</div>

);

}}

/>

);

}

**Custom Hooks with TypeScript**

interface User {

id: number;

name: string;

email: string;

}

interface UseUsersReturn {

users: User[];

loading: boolean;

error: Error | null;

fetchUsers: () => Promise<void>;

}

function useUsers(): UseUsersReturn {

const [users, setUsers] = useState<User[]>([]);

const [loading, setLoading] = useState<boolean>(true);

const [error, setError] = useState<Error | null>(null);

const fetchUsers = async () => {

setLoading(true);

try {

const response = await fetch('/api/users');

if (!response.ok) {

throw new Error(`HTTP error! Status: ${response.status}`);

}

const data = await response.json();

setUsers(data);

setError(null);

} catch (err) {

setError(err instanceof Error ? err : new Error(String(err)));

} finally {

setLoading(false);

}

};

useEffect(() => {

fetchUsers();

}, []);

return { users, loading, error, fetchUsers };

}

// Usage

function UserList() {

const { users, loading, error, fetchUsers } = useUsers();

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

if (error) return <p>Error: {error.message}</p>;

return (

<div>

<button onClick={fetchUsers}>Refresh</button>

<ul>

{users.map(user => (

<li key={user.id}>{user.name} ({user.email})</li>

))}

</ul>

</div>

);

}

**Context with TypeScript**

interface AuthContextType {

user: User | null;

login: (email: string, password: string) => Promise<void>;

logout: () => void;

isAuthenticated: boolean;

loading: boolean;

error: Error | null;

}

const AuthContext = React.createContext<AuthContextType | undefined>(undefined);

function useAuth(): AuthContextType {

const context = useContext(AuthContext);

if (context === undefined) {

throw new Error('useAuth must be used within an AuthProvider');

}

return context;

}

interface AuthProviderProps {

children: React.ReactNode;

}

function AuthProvider({ children }: AuthProviderProps) {

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

const [loading, setLoading] = useState<boolean>(false);

const [error, setError] = useState<Error | null>(null);

const login = async (email: string, password: string) => {

setLoading(true);

setError(null);

try {

const response = await fetch('/api/login', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ email, password })

});

if (!response.ok) {

throw new Error('Login failed');

}

const userData = await response.json();

setUser(userData);

localStorage.setItem('user', JSON.stringify(userData));

} catch (err) {

setError(err instanceof Error ? err : new Error(String(err)));

setUser(null);

} finally {

setLoading(false);

}

};

const logout = () => {

setUser(null);

localStorage.removeItem('user');

};

useEffect(() => {

const storedUser = localStorage.getItem('user');

if (storedUser) {

try {

setUser(JSON.parse(storedUser));

} catch (e) {

localStorage.removeItem('user');

}

}

}, []);

const value = {

user,

login,

logout,

isAuthenticated: !!user,

loading,

error

};

return (

<AuthContext.Provider value={value}>

{children}

</AuthContext.Provider>

);

}

// Usage

function App() {

return (

<AuthProvider>

<Router>

<Routes>

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

<Route path="/dashboard" element={<ProtectedRoute><Dashboard /></ProtectedRoute>} />

</Routes>

</Router>

</AuthProvider>

);

}

function Login() {

const { login, loading, error } = useAuth();

const [email, setEmail] = useState('');

const [password, setPassword] = useState('');

const handleSubmit = async (e: React.FormEvent) => {

e.preventDefault();

await login(email, password);

};

return (

<form onSubmit={handleSubmit}>

{error && <p className="error">{error.message}</p>}

<input

type="email"

value={email}

onChange={e => setEmail(e.target.value)}

placeholder="Email"

required

/>

<input

type="password"

value={password}

onChange={e => setPassword(e.target.value)}

placeholder="Password"

required

/>

<button type="submit" disabled={loading}>

{loading ? 'Logging in...' : 'Login'}

</button>

</form>

);

}

**Compound Components Pattern**

import React, { createContext, useContext, useState } from 'react';

// Types

interface TabsContextType {

activeTab: string;

setActiveTab: (id: string) => void;

}

interface TabsProps {

children: React.ReactNode;

defaultTab?: string;

}

interface TabProps {

children: React.ReactNode;

id: string;

}

interface TabPanelProps {

children: React.ReactNode;

id: string;

}

// Create context

const TabsContext = createContext<TabsContextType | undefined>(undefined);

// Custom hook to use the context

const useTabs = () => {

const context = useContext(TabsContext);

if (!context) {

throw new Error('Tabs components must be used within a Tabs provider');

}

return context;

};

// Compound component

const Tabs = ({ children, defaultTab }: TabsProps) => {

const [activeTab, setActiveTab] = useState(defaultTab || '');

return (

<TabsContext.Provider value={{ activeTab, setActiveTab }}>

<div className="tabs-container">{children}</div>

</TabsContext.Provider>

);

};

// Sub-components

const TabList = ({ children }: { children: React.ReactNode }) => {

return <div className="tab-list">{children}</div>;

};

const Tab = ({ children, id }: TabProps) => {

const { activeTab, setActiveTab } = useTabs();

return (

<button

className={`tab ${activeTab === id ? 'active' : ''}`}

onClick={() => setActiveTab(id)}

>

{children}

</button>

);

};

const TabPanel = ({ children, id }: TabPanelProps) => {

const { activeTab } = useTabs();

if (activeTab !== id) return null;

return <div className="tab-panel">{children}</div>;

};

// Attach sub-components to main component

Tabs.TabList = TabList;

Tabs.Tab = Tab;

Tabs.TabPanel = TabPanel;

// Usage

function App() {

return (

<Tabs defaultTab="tab1">

<Tabs.TabList>

<Tabs.Tab id="tab1">Tab 1</Tabs.Tab>

<Tabs.Tab id="tab2">Tab 2</Tabs.Tab>

<Tabs.Tab id="tab3">Tab 3</Tabs.Tab>

</Tabs.TabList>

<Tabs.TabPanel id="tab1">

<h2>Content for Tab 1</h2>

<p>This is the content for the first tab.</p>

</Tabs.TabPanel>

<Tabs.TabPanel id="tab2">

<h2>Content for Tab 2</h2>

<p>This is the content for the second tab.</p>

</Tabs.TabPanel>

<Tabs.TabPanel id="tab3">

<h2>Content for Tab 3</h2>

<p>This is the content for the third tab.</p>

</Tabs.TabPanel>

</Tabs>

);

}

**Interview Questions & Answers**

**React Basics**

**Q: What is React and why would you use it?**

A: React is a JavaScript library for building user interfaces, particularly single-page applications. I would use React for several reasons:

* Component-based architecture allows for reusable code and better organization
* Virtual DOM makes UI updates efficient
* Declarative approach makes code more predictable and easier to debug
* Strong ecosystem with many libraries and tools
* Backed by Facebook and a large community, ensuring longevity and support

**Q: What is JSX?**

A: JSX is a syntax extension for JavaScript that looks similar to HTML. It allows us to write HTML-like code in our JavaScript files, which makes building React elements more intuitive. JSX gets transformed into JavaScript function calls during the build process. For example, <div>Hello</div> becomes React.createElement('div', null, 'Hello').

**Q: What is the difference between a class component and a functional component?**

A:

* **Class Components**: Extend from React.Component and have access to lifecycle methods, state via this.state, and refs. They use the render() method to return JSX.
* **Functional Components**: JavaScript functions that accept props and return JSX. With hooks (introduced in React 16.8), functional components can now use state, effects, context, and other React features previously only available in class components.

Today, functional components with hooks are preferred due to their simplicity, reduced boilerplate, and easier testing.

**Q: What are props in React?**

A: Props (short for properties) are inputs to React components. They are read-only data passed from parent to child components. Props allow components to be dynamic and reusable by configuring them with different data. The one-way flow of props helps make applications more predictable and easier to understand.

**Q: What is state in React?**

A: State is a JavaScript object that stores component data that can change over time. When state changes, the component re-renders. Unlike props, which are passed from parent to child, state is managed internally by the component. In class components, state is accessed via this.state and updated with this.setState(). In functional components, state is managed with the useState hook.

**React Hooks**

**Q: What are React Hooks and why were they introduced?**

A: React Hooks are functions that let you use React features (like state, lifecycle methods, context) in functional components. They were introduced in React 16.8 to:

* Allow reusing stateful logic between components without changing component hierarchy
* Split complex components into smaller functions based on related pieces
* Use React features without classes, which can be confusing (with this binding, boilerplate code)
* Provide more direct APIs for React concepts like props, state, context, refs, and lifecycle

**Q: Explain the useState hook.**

A: useState is a hook that lets functional components use state. It returns a pair: the current state value and a function to update it.

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

In this example:

* count is the state variable
* setCount is the function to update the state
* 0 is the initial state value

When setCount is called with a new value, React re-renders the component with the updated state.

**Q: Explain the useEffect hook.**

A: useEffect handles side effects in functional components, such as data fetching, subscriptions, or manually changing the DOM. It runs after render and can replace lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.

useEffect(() => {

// Side effect code

document.title = `Count: ${count}`;

// Optional cleanup function

return () => {

// Cleanup code

};

}, [count]); // Dependency array

The dependency array controls when the effect runs:

* Empty array ([]): Runs once after initial render (like componentDidMount)
* With values ([count]): Runs when any dependency changes
* No array: Runs after every render

The optional cleanup function runs before the component unmounts or before the effect runs again.

**Q: What is the difference between useMemo and useCallback?**

A: Both hooks are used for optimization by memoizing values, but they serve different purposes:

* useMemo memoizes a computed value. It recalculates the value only when one of its dependencies changes.
* const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
* useCallback memoizes a function definition. It returns a memoized version of the callback that only changes if one of the dependencies changes.
* const memoizedCallback = useCallback(() => {
* doSomething(a, b);
* }, [a, b]);

The key difference is that useMemo returns a memoized value (result of a function), while useCallback returns a memoized function itself. useCallback(fn, deps) is equivalent to useMemo(() => fn, deps).

**Advanced React**

**Q: What are controlled and uncontrolled components?**

A:

* **Controlled Components**: Form elements whose values are controlled by React state. Any input value changes are handled by state update functions, making React the "single source of truth."
* function ControlledInput() {
* const [value, setValue] = useState('');
* return <input value={value} onChange={e => setValue(e.target.value)} />;
* }
* **Uncontrolled Components**: Form elements that maintain their own internal state. Values are accessed using refs instead of event handlers.
* function UncontrolledInput() {
* const inputRef = useRef();
* const handleSubmit = () => {
* console.log(inputRef.current.value);
* };
* return (
* <>
* <input ref={inputRef} defaultValue="Initial value" />
* <button onClick={handleSubmit}>Submit</button>
* </>
* );
* }

**Q: What is the Context API and when would you use it?**

A: The Context API provides a way to share values between components without explicitly passing props through every level of the component tree. It's useful for global data that many components need, such as:

* User authentication
* Theme settings
* Language preferences
* Global state that doesn't warrant using Redux

// Create context

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

// Provider in parent component

function App() {

return (

<ThemeContext.Provider value="dark">

<ThemedButton />

</ThemeContext.Provider>

);

}

// Consumer in child component

function ThemedButton() {

const theme = useContext(ThemeContext);

return <button className={theme}>Themed Button</button>;

}

**Q: What is code splitting in React and why is it important?**

A: Code splitting is a technique that splits your code into smaller chunks which can be loaded on demand. This improves application performance by reducing the initial load time.

React provides React.lazy and Suspense for component-level code splitting:

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

function MyComponent() {

return (

<div>

<Suspense fallback={<div>Loading...</div>}>

<OtherComponent />

</Suspense>

</div>

);

}

Code splitting is important because:

* It reduces the initial bundle size
* It improves the initial load time of the application
* Users only download the code they need for the features they use
* It enables more efficient caching

**Q: Explain the concept of Virtual DOM and how it works.**

A: The Virtual DOM is a lightweight JavaScript representation of the actual DOM. When state changes in a React component:

1. React creates a new Virtual DOM tree
2. It compares the new Virtual DOM with the previous one using a diffing algorithm
3. It calculates the minimum number of operations needed to update the real DOM
4. It applies only those changes to the real DOM in a batched operation

This process is more efficient than directly manipulating the DOM for every state change because:

* DOM operations are slow compared to JavaScript operations
* Batch updates reduce browser reflows and repaints
* React can optimize updates by prioritizing changes

**Q: What are Error Boundaries in React?**

A: Error Boundaries are React components that catch JavaScript errors in their child component tree, log those errors, and display a fallback UI. They prevent an error in one component from breaking the entire application.

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError(error) {

// Update state so next render shows fallback UI

return { hasError: true };

}

componentDidCatch(error, info) {

// Log the error

console.error(error, info);

}

render() {

if (this.state.hasError) {

// Fallback UI

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

// Usage

<ErrorBoundary>

<MyComponent />

</ErrorBoundary>

Error boundaries only catch errors in:

* Render methods
* Lifecycle methods
* Constructors

They don't catch errors in:

* Event handlers
* Asynchronous code (e.g., setTimeout or requestAnimationFrame callbacks)
* Server-side rendering
* Errors thrown in the error boundary itself

**Performance Optimization**

**Q: How can you optimize performance in a React application?**

A: There are several ways to optimize React performance:

1. **Use React.memo for component memoization**
2. const MyComponent = React.memo(function MyComponent(props) {
3. // Component logic
4. });
5. **Implement shouldComponentUpdate or PureComponent**
6. class MyComponent extends React.PureComponent {
7. // Only re-renders if props or state change (shallow comparison)
8. }
9. **Use useMemo for expensive calculations**
10. const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
11. **Use useCallback for event handlers**
12. const memoizedCallback = useCallback(() => {
13. doSomething(a, b);
14. }, [a, b]);
15. **Code splitting with React.lazy and Suspense**
16. const OtherComponent = React.lazy(() => import('./OtherComponent'));
17. **Virtualization for long lists**
18. import { FixedSizeList } from 'react-window';
19. // Render only visible items in a long list
20. **Avoid unnecessary re-renders**
    * Keep component state as local as possible
    * Avoid using indexes as keys
    * Use stable references for objects and functions
21. **Use Production Builds**
    * Remove development code and warnings
    * Enable minification and compression
22. **Implement bundle analysis and optimization**
    * Remove unused dependencies
    * Use tree-shaking
23. **Implement proper React keys**
    * Use unique and stable IDs for list items

**Q: Why is it important to use keys in React lists, and what makes a good key?**

A: Keys help React identify which items have changed, been added, or been removed. This helps React update the DOM efficiently without recreating all list elements.

A good key should be:

* Unique among siblings
* Stable (not change during re-renders)
* Deterministic (same input = same key)

Good keys:

* Database IDs (e.g., user.id)
* Unique identifiers (e.g., item.slug)
* UUID or other guaranteed unique values

Bad keys:

* Array indexes (change when list order changes)
* Random numbers (change on each render)
* Non-unique values (like common names)

Example:

function To