React Part 3

Q45. What is Conditional Rendering?

Answer: Showing different content based on conditions (like if-else).

Methods:

1. If-Else:

javascript

```
function Greeting({ isLoggedIn }){
if(isLoggedIn){
return
<h1>Welcome back!</h1>;
}
else{
return<h1>Please login</h1>;}
}
```

2. Ternary Operator (most common):

javascript

```
function Greeting({ isLoggedIn }){
return(
<div>
{isLoggedIn ?<h1>Welcome!</h1>:<h1>Please login</h1>}
</div>);}
```

3. Logical AND (&&):

javascript

```
function Notification({ hasMessages }){
return(
<div>
{/* Show only if hasMessages is true */}
{ hasMessages &&<p>You have newmessages!</p>}
</div>);
}
```

4. Switch Case:

javascript

```
function Status({ status }){
switch(status){
case 'loading':return<p>Loading...</p>;

case 'success':return<p>Success!</p>;

case 'error':return<p>Error!</p>;

default:return null;}
}
```

Q46. What is Fragment in React?

Answer: Fragments let you group multiple elements without adding extra DOM nodes.

Problem without Fragment:

javascript

```
// ❌ Must wrap in div (creates extra DOM node)
function Component(){
```

```
return(
<div>
<h1>Title</h1>
<p>Paragraph</p>

</div>

);
}
```

Solution with Fragment:

javascript

```
// ✅ No extra DOM node
function Component(){
return(
<>
<h1>Title</h1>
<p>Paragraph</p>
</>);}

// Or use React.Fragment

function Component(){
return(
<React.Fragment>

<h1>Title</h1>
<p>Paragraph</p>

</React.Fragment>);
}
```

Fragment with Key (for lists):

javascript

```
functionList({ items }){
return(
```

```
<>
{items.map(item=>(
<React.Fragment key={item.id}>

<h3>{item.title}</h3>
<p>{item.description}</p>

</React.Fragment>))}
</>);
}
```

Q47. What is Context API?

Answer: Context API provides a way to pass data through component tree without passing props at every level. Solves "props drilling" problem.

When to use:

- Theme (dark/light mode)
- User authentication
- Language settings
- Any global data

Complete Example:

javascript

```
import{ createContext, useContext, useState }from'react';
// 1. Create Context
const ThemeContext=createContext();

// 2. Provider Component
function App(){
const[theme, setTheme]=useState('light');

return(
<ThemeContext.Provider value={{ theme, setTheme }}>
<Toolbar/>
</ThemeContext.Provider>);}

// 3. Consumer Component (can be deeply nested)
```

```
15
16  function Toolbar(){
17  return(
18  <div>
19  <ThemedButton/>
20  </div>);}
21
22  function ThemedButton(){
23  // 4. Use Context
24  const{ theme, setTheme }=useContext(ThemeContext);
25
26  return(
27
28  <button  style={{background: theme ==='dark'?'#333':'#fff'}}
29  onClick={()=>setTheme(theme ==='dark'?'light':'dark')}>Current theme:{th
30  </button>);}
```

Q48. What is Props Drilling?

Answer: Props drilling is passing props through multiple levels of components to reach a deeply nested component.

Problem Example:

javascript

```
1   // ❌ Props drilling - passing through many levels
2   function App(){
3   const[user, setUser]=useState({name:'John'});
4
5   return <Level1 user={user}/>;
6   }
7
8   function Level1({ user }){
9   return <Level2 user={user}/>;// Just passing through
10
11  }
12
13  function Level2({ user }){
14  return <Level3 user={user}/>;// Just passing through
15
```

```
}
function Level3({ user }){
return <h1>{user.name}</h1>;// Finally used here!
}
```

Solution - Use Context:

javascript

```
// ✅ No props drilling
const UserContext=createContext();

function App(){
const[user, setUser]=useState({name:'John'});

return(
<UserContext.Provider value={user}>
<Level1/>
</UserContext.Provider>);
}

function Level1(){return<Level2/>;// No props needed

}

function Level2(){return<Level3/>;// No props needed

}

function Level3(){
const user =useContext(UserContext);// Get directly!

return <h1>{user.name}</h1>;
}
```

Q49. What is Higher-Order Component (HOC)?

Answer: HOC is a function that takes a component and returns a new component with additional props or functionality.

Simple Example:

javascript

```
// HOC that adds loading functionality

function withLoading(Component){
return function WithLoadingComponent({ isLoading,...props })
{
if(isLoading){
return<div>Loading...</div>;
}
return<Component{...props}/>;
};}

// Original Component

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

// Enhanced Component with loading

const UserListWithLoading=withLoading(UserList);

// Usage
function App(){
const[users, setUsers]=useState([]);
const[loading, setLoading]=useState(true);

return(
<UserListWithLoading isLoading={loading} users={users}/>
);
}
```

Another Example - Authentication HOC:

javascript

```
// HOC for authentication
```

```
2
3   function withAuth(Component){
4   return function AuthComponent(props)
5   {
6   const isAuthenticated =checkAuth();
7
8   // Check if user logged in
9   if(!isAuthenticated){
10  return <Redirect to="/login"/>;
11
12  }
13  return<Component{...props}/>;
14  };}
15
16  // Protected component
17  function Dashboard(){
18  return <h1>Dashboard(Protected)</h1>;
19  }
20  // Wrap with authentication
21  const ProtectedDashboard=withAuth(Dashboard);
```

Q50. What is Render Props Pattern?

Answer: Render Props is a technique where a component receives a function as a prop that returns React elements.

Example:

javascript

```
1   // Component with render prop
2
3   function Mouse({ render }){
4   const[position, setPosition]=useState({x:0,y:0});
5
6   const handleMouseMove=(e)=>{
7   setPosition({x: e.clientX,y: e.clientY});
8   };
9   return(
10  <div onMouseMove={handleMouseMove}>
11  {/* Call the render function with data */}
12  {render(position)}
```

```
</div>);}

// Usage
function App(){

return(

<Mouse render={(position)=>(

<h1>Mouse position:{position.x},{position.y}</h1>)}/>
);}
```

Another Example - Data Fetching:

javascript

```
function DataFetcher({ url, render }){
const[data, setData]=useState(null);
const[loading, setLoading]=useState(true);

useEffect(()=>{
fetch(url).then(res=> res.json()).then(data=>{
setData(data);
setLoading(false);
});
},[url]);
return render({ data, loading });// Pass data to render prop
}
// Usage
function App(){
return(
<DataFetcher url="/api/users" render={({ data, loading })=>(
loading ?<p>Loading...</p>:
<ul>{data.map(...)}</ul>)}/>
);}
```

Q51. What is Error Boundary?

Answer: Error Boundaries catch JavaScript errors in child components, log errors, and display fallback UI.

Note: Must be a class component (no hooks version yet).

Example:

javascript

```
class ErrorBoundary extends React.Component{

constructor(props){super(props);this.state={hasError:false};}

// Catch errors

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

// Log error details
componentDidCatch(error, errorInfo){
console.log('Error:', error);console.log('Error Info:', errorInfo);
}render(){
if(this.state.hasError)
{// Fallback UI
return<h1>Something went wrong!</h1>;
}return this.props.children;// Normal render
}}

// Usage
function App(){
return(<ErrorBoundary><BuggyComponent/></ErrorBoundary>);
}
function BuggyComponent(){
// This will be caught by ErrorBoundary
thrownnewnError('I crashed!');}
```

What Error Boundaries DON'T catch:

- Event handlers (use try-catch)
- Async code (setTimeout, promises)
- Server-side rendering
- Errors in Error Boundary itself

Q52. What is Lazy Loading?

Answer: Lazy loading loads components only when needed, reducing initial bundle size.

Example:

javascript

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

// Lazy load component

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

function App(){

return(
<div>
{/* Suspense shows fallback while loading */}

<Suspense fallback={
<div>Loading...</div>
}><HeavyComponent/>
</Suspense>
</div>);}
```

Multiple Lazy Components:

javascript

```
// Lazy load multiple components

const Home=lazy(()=>import('./Home'));
const About=lazy(()=>import('./About'));
const Contact=lazy(()=>import('./Contact'));

function App(){
const[page, setPage]=useState('home');

```

```
return(
<div>
<nav>
<button onClick={()=>setPage('home')}>Home</button>
<button onClick={()=>setPage('about')}>About</button>
<button onClick={()=>setPage('contact')}>Contact</button>

</nav>

<Suspense fallback={
<div>Loading page...</div>}>
{page ==='home'&&<Home/>}
{page ==='about'&&<About/>}
{page ==='contact'&&<Contact/>}
</Suspense>

</div>);}
```

Q53. What is Code Splitting?

Answer: Code splitting divides your code into smaller bundles that load on demand.

Benefits:

- Faster initial load
- Better performance
- Load code when needed

Example with React Router:

javascript

```
import{ lazy,Suspense}from'react';
import{BrowserRouter,Routes,Route}from'react-router-dom';

// Split code by routes

const Home=lazy(()=>import('./pages/Home'));
const Products=lazy(()=>import('./pages/Products'));
```

```
const Cart=lazy(()=>import('./pages/Cart'));

function App(){
return(
<BrowserRouter>
 <Suspense fallback={<div>Loading...</div>}>
<Routes>
 <Route path="/" element={<Home/>}/>
 <Route path="/products" element={<Products/>}/>
 <Route path="/cart" element={<Cart/>}/>
</Routes>
</Suspense>
</BrowserRouter>
);}
```

Q54. What is React.memo()?

Answer: React.memo() prevents unnecessary re-renders by memoizing a component. Re-renders only if props change.

Without memo:

javascript

```
// ❌ Re-renders every time parent updates

function ExpensiveComponent({ name })
{
console.log('Rendered');

return<h1>Hello{name}</h1>;
}
```

With memo:

javascript

```
import{ memo }from'react';
```

```
2
3   //  Re-renders only if name changes
4   const ExpensiveComponent=memo(functionExpensiveComponent({ name }){
5
6   console.log('Rendered');
7   return<h1>Hello{name}</h1>;
8   });
9
10  // Usage
11  function Parent(){
12  const[count, setCount]=useState(0);
13  const[name, setName]=useState('John');
14
15  return(
16  <div>
17  <button onClick={()=>setCount(count +1)}>Count:{count}</button>
18
19  {/* Won't re-render when count changes */}
20  <ExpensiveComponent name={name}/>
21
22  </div>);}
```

Custom comparison:

javascript

```
1   // Custom comparison function
2
3   const MyComponent=memo(function MyComponent({ user }){
4
5   return <div>{user.name}</div>;},(prevProps, nextProps)=>{
6
7   // Return true if props are equal (skip re-render)
8
9   return prevProps.user.id=== nextProps.user.id;});
```

🎯 Section 9: Performance Optimization

Q55. How to optimize React performance?

Answer: Multiple techniques:

1. Use React.memo():

javascript

```
const ExpensiveComponent=memo(functionComponent({ data })
{
// Only re-renders if data changes
return <div>{data}</div>;
});
```

2. Use useMemo() for expensive calculations:

javascript

```
function Component({ numbers })
{
// Recalculates only if numbers change
const sum =useMemo(()=>{

return numbers.reduce((a, b)=> a + b,0);},[numbers]);

return <div>Sum:{sum}</div>;
}
```

3. Use useCallback() for functions:

javascript

```
function Parent(){
const[count, setCount]=useState(0);
// Function recreated only if count changes
const handleClick =useCallback(()=>{

console.log(count);

```

```
},[count]);
return <Child onClick={handleClick}/>;
}
```

4. Lazy load components:

javascript

```
const HeavyComponent=lazy(()=>import('./Heavy'));
```

5. Use proper keys in lists:

javascript

```
// Good - unique id
{
items.map(item=><li key={item.id}>{item.name}</li>)
}

// ❌ Bad - index
{
items.map((item, i)=><li key={i}>{item.name}</li>)
}
```

6. Avoid inline functions:

javascript

```
// ❌ Bad - creates new function on every render

<button onClick={()=>handleClick()}>Click</button>

//  Good - reuses same function
<button onClick={handleClick}>Click</button>
```

7. Use production build:

bash

```
npm run build
```

Q56. What is useCallback() and when to use it?

Answer: useCallback returns a memoized function. Use it to prevent child re-renders.

Problem without useCallback:

javascript

```
function Parent(){
const[count, setCount]=useState(0);
// ❌ New function created on every render
const handleClick=()=>{
console.log('Clicked');
};
return(
<div>
<button onClick={()=>setCount(count +1)}>Count:{count}</button>

<ExpensiveChild onClick={handleClick}/>

</div>);
}
```

Solution with useCallback:

javascript

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

// ✅ Function created once, reused on every render

const handleClick =useCallback(()=>{console.log('Clicked');},[]);

// Empty deps = never recreated

return(
<div>
<button onClick={()=>setCount(count +1)}>Count:{count}</button>

<ExpensiveChild onClick={handleClick}/>
</div>);}

const ExpensiveChild=memo(function ExpensiveChild({ onClick })
{
console.log('Child rendered');
return <button onClick={onClick}>ChildButton</button>;
});
```

With dependencies:

javascript

```
function SearchBox(){
const[query, setQuery]=useState('');

// Recreated only when query changes
const handleSearch =useCallback(()=>{

console.log('Searching for:', query);},[query]);

return <SearchButton onSearch={handleSearch}/>;
}
```

Q57. What is useMemo() and when to use it?

Answer: useMemo returns a memoized value. Use it for expensive calculations.

Problem without useMemo:

javascript

```
function Component({ numbers }){
// ❌ Recalculates on every render (expensive!)

const sum = numbers.reduce((a, b)=> a + b,0);
return <div>Sum:{sum}</div>;
}
```

Solution with useMemo:

javascript

```
function Component({ numbers }){
// ✅ Recalculates only when numbers change

const sum =useMemo(()=>{console.log('Calculating sum...');
return numbers.reduce((a, b)=> a + b,0);},[numbers]);

return<div>Sum:{sum}</div>;}
```

Filtering example:

javascript

```
function UserList({ users, searchTerm }){
// Filter only when users or searchTerm changes

const filteredUsers =useMemo(()=>{
console.log('Filtering users...');
return users.filter(user=> user.name.toLowerCase().includes(searchTerm.t
},[users, searchTerm]);
return(
```

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

When to use:

- ✅ Expensive calculations
- ✅ Filtering/sorting large arrays
- ✅ Complex computations
- ❌ Simple operations (overhead not worth it)

Q58. Difference between useMemo and useCallback?

Answer:

```
useMemo(() => value, deps)
```

useMemo example:

javascript

```
// Returns calculated VALUE
const sum =useMemo(()=> a + b,[a, b]);
```

useCallback example:

javascript

```
// Returns memoized FUNCTION
const handleClick =useCallback(()=>{console.log('clicked');},[]);
```

They're related:

javascript

```
// These are equivalent:
useCallback(fn, deps)useMemo(()=> fn, deps)
```

Q59. What is StrictMode?

Answer: StrictMode is a tool to highlight potential problems in your app. Only works in development.

Features:

- Identifies unsafe lifecycles
- Warns about legacy API usage
- Detects unexpected side effects
- No visible UI
- Doesn't affect production

Usage:

javascript

```
import{StrictMode}from'react';
functionApp(){
return(
<StrictMode>
 <MyApp/>
</StrictMode>);
}
```

What it does:

javascript

```
// In development, components render twice
// to detect side effects
function MyComponent(){

console.log('Rendered');// Logs twice in StrictMode

return<div>Hello</div>;
}
```

Partial usage:

javascript

```
function App(){
return(
<div>
<Header/> {/* Not in StrictMode */}

<StrictMode>
 <Content/>   {/* Only this in StrictMode */}
</StrictMode>

<Footer/>{/* Not in StrictMode */}
</div>);}
```

Q60. What is PropTypes?

Answer: PropTypes validate the types of props passed to components. Helps catch bugs.

Installation:

bash

```
npm install prop-types
```

Q61. What is React DevTools?

Answer: React DevTools is a browser extension for debugging React applications.

Features:

- View component tree
- Inspect props and state
- Track component updates
- Profile performance
- Debug hooks

Installation:

- Chrome: Chrome Web Store
- Firefox: Firefox Add-ons

How to use:

javascript

```
// 1. Install extension// 2. Open browser DevTools (F12)// 3. Click "Com
functionMyComponent(){const[count, setCount]=useState(0);
// View in DevTools:// - Component name: MyComponent// - State: { count:
return<button onClick={()=>setCount(count +1)}>{count}</button>;}
```

Q62. What is Portals?

Answer: Portals render children into a DOM node outside the parent component hierarchy.

Use cases:

- Modals
- Tooltips
- Dropdowns
- Notifications

Example:

javascript

```
import{ createPortal }from'react-dom';
function Modal({ children, isOpen }){

if(!isOpen)return null;
// Render into document.body (outside root div)
return createPortal(
<div className="modal-backdrop"><div className="modal">{children}</div>
</div>,document.body
// Target DOM node
);}

// Usage
function App(){
const[showModal, setShowModal]=useState(false);
return(
<div>
<h1>MyApp</h1>
<button onClick={()=>setShowModal(true)}>OpenModal</button>

<Modal isOpen={showModal}>
<h2>ModalContent</h2>
<button onClick={()=>setShowModal(false)}>Close</button>
</Modal>
</div>);}
```