**React Debugging Instructions**

**Prerequisites: Install React Developer Tools**

1. Install the **React Developer Tools** browser extension:
   * Chrome: [React Developer Tools](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi)
   * Firefox: Search "React Developer Tools" in Firefox Add-ons
2. Open your browser's Developer Tools (F12 or right-click → Inspect)
3. You should see "⚛️ Components" and "⚛️ Profiler" tabs

**General Debugging Workflow**

**Step 1: Open Developer Tools**

* **Windows/Linux**: F12 or Ctrl+Shift+I
* **Mac**: Cmd+Option+I
* **Right-click method**: Right-click on page → "Inspect" or "Inspect Element”

**Step 2: Check Console First**

* Click the **Console** tab
* Look for red error messages
* Look for yellow warnings
* Check for any console.log output

**Step 3: Use React Developer Tools**

* Click the ⚛️ **Components** tab
* Navigate the component tree on the left
* Inspect props and state on the right
* Use the search box to find specific components

**Here are the key tools referenced in the exercises:**

**Browser Developer Tools**

* **Console** - For error messages, console.log output, and warnings
* **Elements/Inspector** - For examining the DOM structure
* **Network tab** - For debugging API calls and resource loading
* **Sources tab** - For setting breakpoints in React components
* **Performance tab** - For profiling and identifying performance bottlenecks

**React-Specific Debugging Tools**

* **React Developer Tools** (browser extension) - The most important React debugging tool:
  + Components tab: Inspect component hierarchy, props, and state
  + Profiler tab: Identify performance issues and unnecessary re-renders
  + Hooks inspection
  + Context value tracking

**Debugging Techniques Shown in the Exercises**

* **Strategic console.log placement** - Exercise 7 shows this with console.log('ExpensiveChild rendered')
* **Error boundary implementation** - For catching and handling component errors
* **Console methods**:
  + console.log() - Basic debugging output
  + console.table() - For displaying objects/arrays nicely
  + console.trace() - For call stack information

**Tools for Specific Issues**

* **Memory leak detection** - Browser's Memory tab in DevTools
* **Performance monitoring** - React DevTools Profiler + browser Performance tab
* **State inspection** - React DevTools for live state monitoring
* **Props flow debugging** - React DevTools component tree inspection

**Advanced Debugging Tools (not implemented)**

* **Redux DevTools** - For Redux state debugging (if using Redux)
* **React Query DevTools** - For server state debugging
* **Storybook** - For component isolation and testing
* **React Testing Library** - For writing debuggable tests

**These exercises specifically use:**

1. **Browser console** - Check for errors and debug output
2. **React DevTools** - Inspect component state, props, and re-render patterns
3. **Breakpoint debugging** - Setting breakpoints in component code
4. **Console debugging** - Strategic placement of console.log statements

**Hints:**

* **Always use functional updates** for state that depends on previous state
* **Memoize callbacks and expensive calculations** to prevent unnecessary re-renders
* **Clean up side effects** in useEffect return functions
* **Use stable unique keys** for list items, never array indices
* **Wrap context consumers** in providers
* **Handle async operations properly** with cleanup

**Now let’s debug the buggy components:**

**Exercise 1: Component Not Rendering**

**Follow These Follow These Debugging Steps:**

1. **Console Tab**: Look for syntax errors or warnings
2. **Elements Tab**: Check if the component's div exists in DOM but is empty
3. **Components Tab**: Find the BuggyComponent and see if it's rendered
4. **Sources Tab**:
   * Find the component file
   * Set a breakpoint on the line with JSX
   * Notice execution never reaches the JSX line

**You'll Notice:**

* Empty div in Elements tab
* No console errors (tricky!)
* Component appears in React DevTools but renders nothing

**Debugging Technique:** Add console.log('Component rendering') at the start of the component to verify it's being called.

// ========================================

// EXERCISE 1: Component Not Rendering

// ========================================

// BUGGY VERSION - Missing return statement

const BuggyComponent1 = () => {

const message = "Hello World!";

// BUG: Missing return statement

<div>{message}</div>;

};

// FIXED VERSION

const FixedComponent1 = () => {

const message = "Hello World!";

return <div>{message}</div>; // Added return

};

// Alternative fixes:

const FixedComponent1Alt = () => {

const message = "Hello World!";

return ( // Explicit return with parentheses

<div>{message}</div>

);

};

**Exercise 2: Props Not Passing Through**

**Follow These Debugging Steps:**

1. **Components Tab**:
   * Find ChildComponent in the component tree
   * Look at the props panel on the right
   * Notice user prop exists, but name and age don't
2. **Console Tab**: Look for "Cannot read property of undefined" warnings

**You'll Notice:**

* Props panel shows: user: {name: "John", age: 25}
* But component expects name and age directly
* Text shows "undefined" values

**Debugging Technique:** Right-click the child component → Inspect → check props structure vs. what component expects.

// ========================================

// EXERCISE 2: Props Not Passing Through

// ========================================

// BUGGY VERSION - Wrong prop structure

const BuggyParent2 = () => {

const userData = { name: "John", age: 25 };

// BUG: Passing object as 'user' prop, but child expects 'name' and 'age'

return <ChildComponent user={userData} />;

};

const ChildComponent = ({ name, age }) => (

<div>

<p>Name: {name}</p> {/\* Will be undefined \*/}

<p>Age: {age}</p> {/\* Will be undefined \*/}

</div>

);

// FIXED VERSION - Proper prop passing

const FixedParent2 = () => {

const userData = { name: "John", age: 25 };

// Fix 1: Spread the object

return <ChildComponent {...userData} />;

// Fix 2: Pass individual props

// return <ChildComponent name={userData.name} age={userData.age} />;

};

// Alternative: Change child to accept user object

const AlternativeChild = ({ user }) => (

<div>

<p>Name: {user?.name}</p>

<p>Age: {user?.age}</p>

</div>

);

**Exercise 3: Event Handler Issues**

**Follow These Debugging Steps:**

1. **Console Tab**: Look for immediate function call errors
2. **Elements Tab**:
   * Find the button element
   * Look at the onclick attribute in HTML
   * Notice it might show the result of the function call, not the function
3. **Components Tab**: Check if state updates when button is clicked

**You'll Notice:**

* Console error: "Cannot read property 'call' of undefined" or similar
* Button's onClick shows result instead of function reference

**Debugging Technique:** Add console.log('Button clicked') inside a proper arrow function to test event handling.

// ========================================

// EXERCISE 3: Event Handler Issues

// ========================================

// BUGGY VERSION - Calling function immediately

const BuggyButton3 = () => {

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

return (

<div>

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

{/\* BUG: Calling setCount immediately, not passing function reference \*/}

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

Increment

</button>

</div>

);

};

// FIXED VERSION - Proper event handlers

const FixedButton3 = () => {

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

return (

<div>

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

{/\* Fix 1: Arrow function \*/}

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

Increment

</button>

{/\* Fix 2: Functional update (preferred) \*/}

<button onClick={() => setCount(prev => prev + 1)}>

Better Increment

</button>

</div>

);

};

// Fix 3: Separate handler function

const FixedButton3Alt = () => {

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

const handleIncrement = () => {

setCount(prev => prev + 1);

};

return (

<div>

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

<button onClick={handleIncrement}>

Increment

</button>

</div>

);

};

**Exercise 4: State Not Updating**

**Follow These Debugging Steps:**

1. **Components Tab**:
   * Find the component with items state
   * Click the button and watch the state panel
   * Notice the state reference doesn't change (same array object)

**Console Tab**: Add debugging:   
  
const addItem = () => {

console.log('Before:', items);

items.push('orange');

console.log('After:', items);

console.log('Same reference?', items === items);

setItems(items);

};

**You'll Notice:**

* State panel in React DevTools shows same array reference
* Console shows array contents change but React doesn't re-render
* Array reference stays the same

**Debugging Technique:** React DevTools will highlight when state actually changes. If it doesn't highlight, React didn't detect the change.

// ========================================

// EXERCISE 4: State Not Updating

// ========================================

// BUGGY VERSION - Mutating state directly

const BuggyList4 = () => {

const [items, setItems] = useState(['apple', 'banana']);

const addItem = () => {

// BUG: Mutating the original array

items.push('orange');

setItems(items); // React won't detect this change

};

const removeItem = (index) => {

// BUG: Mutating with splice

items.splice(index, 1);

setItems(items);

};

return (

<div>

<ul>

{items.map((item, index) => (

<li key={index}>

{item}

<button onClick={() => removeItem(index)}>Remove</button>

</li>

))}

</ul>

<button onClick={addItem}>Add Orange</button>

</div>

);

};

// FIXED VERSION - Immutable updates

const FixedList4 = () => {

const [items, setItems] = useState(['apple', 'banana']);

const addItem = () => {

// Fix: Create new array with spread operator

setItems([...items, 'orange']);

// Alternative: setItems(prev => [...prev, 'orange']);

};

const removeItem = (index) => {

// Fix: Filter creates new array

setItems(items.filter((\_, i) => i !== index));

// Alternative: setItems(prev => prev.filter((\_, i) => i !== index));

};

const updateItem = (index, newValue) => {

// Fix: Map creates new array

setItems(items.map((item, i) => i === index ? newValue : item));

};

return (

<div>

<ul>

{items.map((item, index) => (

<li key={index}>

{item}

<button onClick={() => removeItem(index)}>Remove</button>

</li>

))}

</ul>

<button onClick={addItem}>Add Orange</button>

</div>

);

};

**Exercise 5: Infinite Re-renders**

**Follow These Debugging Steps:**

1. **Console Tab**:
   * You'll see hundreds/thousands of console messages
   * Error: "Too many re-renders. React limits the number of renders..."
2. **Components Tab**:
   * Component will be constantly updating
   * State keeps changing rapidly
3. **Profiler Tab**:
   * Click "Start profiling"
   * See constant re-renders happening

**You'll Notice:**

* Console flooded with messages
* Browser may become unresponsive
* React error about render limits

**Debugging Technique:** Add console.log('Render') at component start and console.log('Effect') in useEffect to see the loop.

// ========================================

// EXERCISE 5: Infinite Re-renders

// ========================================

// BUGGY VERSION - Missing dependency array

const BuggyInfiniteRender5 = () => {

const [data, setData] = useState(null);

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

// BUG: useEffect runs after every render, causing infinite loop

useEffect(() => {

if (!data) {

setData({ loaded: true, timestamp: Date.now() });

}

}); // Missing dependency array!

// BUG: Object in dependency array causes infinite re-renders

useEffect(() => {

console.log('Data changed:', data);

}, [data]); // data is a new object every time

return <div>Data loaded: {data ? 'Yes' : 'No'}</div>;

};

// FIXED VERSION - Proper dependency arrays

const FixedInfiniteRender5 = () => {

const [data, setData] = useState(null);

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

// Fix 1: Empty dependency array for one-time effect

useEffect(() => {

if (!data) {

setData({ loaded: true, timestamp: Date.now() });

}

}, []); // Runs only once

// Fix 2: Depend on specific properties, not the whole object

useEffect(() => {

console.log('Data loaded status changed:', data?.loaded);

}, [data?.loaded]); // Only re-run if loaded status changes

// Fix 3: Use useCallback for stable function references

const handleClick = useCallback(() => {

setCount(prev => prev + 1);

}, []);

return (

<div>

<div>Data loaded: {data ? 'Yes' : 'No'}</div>

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

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

</div>

);

};