React Internals: Virtual DOM, Diffing Algorithm, and Object Model

Last updated by | Subramanya Dixit | May 5, 2025 at 8:45 PM GMT+5:30

📘 Why Understand React Internals?

Understanding how React works under the hood (like the Virtual DOM and the reconciliation process) helps in writing more efficient and optimized code. It gives you control over performance, rendering behavior, and debugging complex UI issues.

🧠 Virtual DOM

✅ What is the Virtual DOM?

The **Virtual DOM (VDOM)** is an in-memory representation of the actual DOM elements. React uses it to perform efficient updates to the UI.

- It is a **lightweight JavaScript object**.
- React builds a Virtual DOM every time the state or props change.
- Instead of modifying the browser DOM directly, React:
  1. Creates a new Virtual DOM tree.
  2. Compares it with the previous version using the **Diffing Algorithm**.
  3. Applies the minimal number of DOM changes (patches) to the actual DOM.

```
// Simplified example
const vDom = React.createElement('div', { className: 'greeting' }, 'Hello');
```

🔍 React Diffing Algorithm

✅ What is Reconciliation?

The process of updating the actual DOM based on changes in the Virtual DOM is called **reconciliation**. React's Diffing Algorithm is at the core of reconciliation.

⚙️ Key Features of React's Diffing Algorithm

1. **Element Type Comparison**
   - If the element type changes (e.g., `<div>` to `<p>`), the entire subtree is replaced.
2. **Keyed Children Optimization**
   - When rendering lists, React uses `key` to track items between re-renders.
   - Without a key, React defaults to index-based tracking, which is less efficient and may cause issues.
3. **Component Updates**
   - For functional or class components, React will:
     - Re-render the component if props/state change.

- Keep the component as-is if there's no change.

⚡ Performance Tip

Always use **unique keys** for lists:

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

🧱 React Object Model (Working Style)

React elements and components follow a declarative, functional model based on JavaScript objects and functions.

✅ How JSX Compiles

JSX is syntactic sugar for `React.createElement()`:

```
const element = <h1>Hello</h1>;

// Compiles to:
const element = React.createElement('h1', null, 'Hello');
```

✅ React Element

A React element is a plain JavaScript object with:

- `type`: string (for HTML tag) or function/class (for components)
- `props`: attributes and children

```
const el = {
  type: 'div',
  props: {
    className: 'box',
    children: 'Text',
  }
};
```

✅ React Fiber (Advanced)

React Fiber is the **reconciliation engine** used internally to enable:

- Incremental rendering
- Pausing, resuming, and aborting rendering
- Scheduling and prioritizing updates

It allows React to be **asynchronous and concurrent** in rendering, improving performance in large apps.

📊 Visual Overview

```
graph TD
  A[React Element Tree (JSX)] --> B[Virtual DOM]
  B --> C[Previous Virtual DOM]
  C --> D[Diffing Algorithm]
  D --> E[DOM Updates (Reconciliation)]
```

💡 Best Practices

- Avoid frequent or deep DOM updates — React optimizes but isn't magical.
- Use `key` props wisely in lists to reduce re-renders.
- Understand when and why a component re-renders to avoid performance bottlenecks.

❓ Quiz

1. What is the Virtual DOM?
a) A replica of the browser DOM
✅ **b) A lightweight JS object representing the DOM**
c) A React-specific browser API
d) A template rendering engine

---

2. When does React perform reconciliation?
a) When the app starts
b) When the browser DOM changes manually
✅ **c) When state or props change**
d) Every time a new component mounts

---

3. What is the purpose of a `key` in a list?
a) To secure the data
✅ **b) To help React identify changed elements**
c) To sort the list
d) To assign an ID to each element

---

4. What does React.createElement() return?
a) DOM node
✅ **b) JavaScript object representing a UI element**
c) JSX
d) A promise

🛠️ Practice Exercises

1. Write a list-rendering component using proper `key` props.
2. Explore rendering behavior using `React.memo()` or `useMemo`.
3. Manually simulate Virtual DOM comparison by writing two virtual trees and comparing them.
4. Experiment with a profiler to observe unnecessary re-renders.