**🧩 What is useRef?**

👉 useRef is a React Hook that lets you **store a mutable value** (something that can change) that **does not trigger re-renders** when it changes.

Think of it as a **“box” that can hold a value or a DOM reference**, and React will *not* watch it for re-renders.

**💡 Two Main Use Cases**

**1️⃣ Accessing DOM elements directly**

Sometimes you need to manipulate DOM elements — like focusing an input, scrolling, or reading width/height.

✅ useRef points to the real DOM node.  
No document.querySelector() needed.

**🧠 What is the DOM?**

DOM = **Document Object Model**  
→ It’s how the browser represents HTML elements as objects that JavaScript (and React) can interact with.

Example:  
HTML

<input id="name" />

JS

document.getElementById("name").focus();

That’s vanilla JS DOM manipulation.

**⚙️ React Way — useRef**

In React, instead of document.getElementById(),  
we use **useRef** to get **direct access to a DOM element** rendered by React.

Why?  
Because React controls the DOM — we can’t (and shouldn’t) directly use document.getElementById() everywhere.

**🧩 Basic Syntax**

const inputRef = useRef(null); // Step 1: create ref

<input ref={inputRef} /> // Step 2: attach ref

inputRef.current // Step 3: access DOM node

| **DOM Element** | **Common Use** | **Example Action** |
| --- | --- | --- |
| <input> | Focus, clear, or read value | inputRef.current.focus() |
| <textarea> | Auto-focus or scroll | textRef.current.scrollTop |
| <button> | Click programmatically | btnRef.current.click() |
| <div> | Measure dimensions, scroll | divRef.current.clientHeight |
| <img> | Change image source | imgRef.current.src = "..." |
| <video> | Play/pause control | videoRef.current.play() |
| <form> | Submit or reset | formRef.current.submit() |
| <canvas> | Access drawing context | canvasRef.current.getContext("2d") |
| <audio> | Play/pause control | audioRef.current.pause() |

**2️⃣ Storing values that don’t trigger re-render**

If you used useState, updating it re-renders the component.  
But sometimes you just want to **store a value** (like a timer ID, previous value, or count) without re-rendering.

✅ The count value updates internally  
❌ But React does not re-render each time

If you used useState, it *would* re-render on every click — unnecessary.

**🔁 3️⃣ Comparing useRef vs useState**

| **Feature** | **useRef** | **useState** |
| --- | --- | --- |
| Triggers re-render on change | ❌ No | ✅ Yes |
| Preserves value between renders | ✅ Yes | ✅ Yes |
| Use case | DOM access, store mutable data | Component state |
| **5️⃣ Common Interview Questions**  1️⃣ **What is useRef used for?** → To access DOM elements or persist mutable values without causing re-renders.  2️⃣ **What’s the difference between useState and useRef?** → useState causes re-render; useRef doesn’t.  3️⃣ **Does changing ref.current cause re-render?** → ❌ No. It updates silently.  4️⃣ **Can useRef store previous state values?** → ✅ Yes! Common trick:  **🧩 Scenario: Notification Count Updates**  Let’s say you have a **bell icon** 🔔 and whenever someone does something (like sends a message, or triggers an event), you want to **increase the notification count** and show it live.  **❓Can we use useRef for this?**  ➡️ **No — not by itself.** Because useRef does **not cause re-render** when its value changes.  **⚙️ But where useRef helps together with useState**  You can **combine both** when you need **real-time data** but don’t want re-renders on every update (for example, background tracking, WebSocket events, or API polling).  Perfect 🔥 Let’s take a **real-world example** — a small **chat window** that automatically scrolls to the bottom when a new message arrives (just like WhatsApp or Slack).  We’ll use **useRef** here for DOM manipulation — not useState — because we want to scroll the actual HTML element, not trigger re-renders.  **🔍 What’s happening**   | **Step** | **What Happens** | | --- | --- | | 1️⃣ | You start with a couple of messages | | 2️⃣ | chatEndRef points to the invisible bottom <div> | | 3️⃣ | When messages update → useEffect runs | | 4️⃣ | chatEndRef.current.scrollIntoView() smoothly scrolls the chat window down | | 5️⃣ | The user always sees the latest message without manual scrolling | |  |  |

**🧠 Key Takeaways**

✅ useRef points to a real DOM element — div, input, video, etc.  
✅ It gives **direct control** (focus, scroll, play/pause, etc.)  
✅ Does **not cause re-render** when you change .current  
✅ It’s ideal for **DOM side-effects** like this.

**⚡ Other DOM control examples you can build easily:**

* Autofocus an input on load
* Scroll to top of a list
* Play/pause a video
* Measure width/height of an element
* Detect clicks outside an element

**🧠 So… why & when do we use useRef together with useState?**

Because:

useState is for **UI updates** (causes re-render)  
useRef is for **logic or background tracking** (no re-render)

In real-world apps, you often need **both**:

* Some parts of data should re-render UI (state)
* Some should stay in memory but not re-render (ref)

**🚀 So, is it *more optimized*?**

✅ Yes — in cases where you’d otherwise cause **unnecessary re-renders** by storing everything in useState.  
✅ It reduces React’s workload while keeping logic consistent.  
❌ But — don’t replace all useState with useRef.  
They **work together**, not instead of each other.

**🧩 Think of it like this:**

useState → "React, I need to show this value on screen."  
useRef → "React, just remember this value quietly in the background."

**🧠 What is a WebSocket?**

**WebSocket** is a protocol that allows **real-time, two-way communication** between the client (like your React app) and the server — **over a single, persistent connection**.

**🏗️ Normally (HTTP)**

In normal APIs, your app uses HTTP requests:

* You → send a request ➡️ server
* Server → sends a response ⬅️ back
* Then the connection closes ❌

If you need updates again, you have to send **another request** (called *polling*).

Example:

// You keep asking server every 5 sec

setInterval(() => {

fetch("/api/notifications").then(...)

}, 5000);

❌ Waste of network  
❌ Delays between updates  
❌ Not truly “real-time”

**⚡ With WebSocket**

When you connect once, the connection **stays open**.  
Now both client and server can talk **anytime** — instantly.

* Client sends → messages to server
* Server sends → messages back in real time  
  ✅ No repeated requests  
  ✅ Instant updates

**💬 Real-life example**

**Example 1: Chat App**

In WhatsApp (web):

* You send a message → goes to server via WebSocket
* Server pushes new messages instantly to all connected users

Without WebSocket → you’d have to refresh or poll for new messages every few seconds 😬

**Example 2: Stock Market Dashboard**

* Server sends price updates every second
* Browser instantly reflects new prices  
  💰 No need to refresh or poll every time

**🧱 Where WebSockets Are Used**

| **Use Case** | **Example** |
| --- | --- |
| Chat / Messaging | WhatsApp, Slack |
| Live Dashboard | Stock prices, Crypto tracker |
| Gaming | Multiplayer moves |
| Live Collaboration | Google Docs |
| Notifications | Payment updates, Alerts |

Whenever you see **real-time communication** — where data updates instantly without refreshing — you can bet that **WebSockets (or similar real-time tech)** are working behind the scenes.

**💡 Short Answer:**

🧠 **WebSockets are not *dependent* on useRef,**  
but in **React**, we often **use useRef to store the WebSocket connection** safely.

**🧠 Your question:**

“Is there any other way to use WebSocket other than useRef? And is that optimized or not?”

✅ Yes — there *are other ways* to use WebSockets in React.  
But… not all are optimal.

Let’s compare them one by one 👇

**🧩 Option 1: Using useRef ✅ *(Most common and optimal way)***

**🧩 Option 4: Using a Context Provider (with useRef inside) ✅✅ *(Best for large-scale apps)***

**If multiple components (e.g. ChatList, MessageInput, Notifications) all need socket access,  
you can centralize it using React Context.**