useRef

DOM references, persistent values across renders, useRef vs useState, and using useRef together with useEffect -- with simple examples.

Intermediate 11 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/use-ref

---

useRef

In useEffect, we saw a component reaching outside its render output. This lesson covers another hook that solves a similar need -- "remembering" a value without triggering the screen to update, or accessing a DOM element directly: `useRef`.

DOM References

One of `useRef`'s most common uses is accessing a DOM element directly:

```
import { useRef } from "react";

function DomReferenceExample() {
  const inputRef = useRef(null);

  function focusInput() {
    // inputRef.current is the actual DOM element itself -- we can call the
    // browser methods the input has (like focus()) directly.
    inputRef.current.focus();
  }

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Focuses when the button is clicked" />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}
```

Writing `ref={inputRef}` tells React "put this element's real DOM node into `inputRef.current`." `inputRef.current` is now a real DOM element -- we can call the browser's own methods on it directly, like `focus()`.

Persistent Values Across Renders

`useRef`'s second use: "remembering" a value across renders, without that value changing triggering a RE-RENDER:

```
import { useState, useRef } from "react";

function PersistentValueExample() {
  const [count, setCount] = useState(0);
  const renderCount = useRef(0);

  renderCount.current = renderCount.current + 1;
  // renderCount.current changes, but this does NOT TRIGGER A RE-RENDER --
  // it only shows the up-to-date value on the next render.

  return (
    <div>
      <p>Count: {count}</p>
      <p>This component has rendered {renderCount.current} times so far.</p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

`renderCount.current` increases on every render and its value persists across renders -- but that increase doesn't trigger a re-render on its own. We only see the updated value on the NEXT render (triggered here by the `count` state changing for another reason).

useRef vs. useState

The key difference between `useRef` and `useState`: changing state with `useState` TRIGGERS a re-render, changing a ref does NOT:

```
import { useState, useRef } from "react";

function UseRefVsUseStateExample() {
  const [stateValue, setStateValue] = useState(0);
  const refValue = useRef(0);

  function incrementState() {
    setStateValue(stateValue + 1); // Updates the screen (triggers a re-render).
  }

  function incrementRef() {
    refValue.current = refValue.current + 1;
    console.log("refValue.current:", refValue.current);
    // The value really did change, but the screen is NOT UPDATED -- because
    // changing a ref does not trigger a re-render.
  }

  return (
    <div>
      <p>State: {stateValue}</p>
      <button onClick={incrementState}>Increment State</button>
      <button onClick={incrementRef}>Increment Ref (Check Console)</button>
    </div>
  );
}
```

Clicking "Increment State" updates the screen, because `setStateValue` triggers a re-render. Clicking "Increment Ref" really does change the value (you can see it in the console), but nothing changes on screen -- because a ref change doesn't tell React "re-render." A value that needs to be VISIBLE on screen should be state; a value that just needs to be "remembered," without appearing on screen, can be a ref.

Using useRef and useEffect Together

`useRef` and `useEffect` are often used together -- for example, to answer "what was the value on the previous render":

```
import { useState, useEffect, useRef } from "react";

function PreviousValueWithRefExample() {
  const [count, setCount] = useState(0);
  const previousCountRef = useRef(undefined);

  useEffect(() => {
    // AFTER every render, we save the current value into the ref -- so we
    // can read it as the "previous value" on the next render. We use a ref
    // instead of state here because we want to record this without
    // triggering another render.
    previousCountRef.current = count;
  });

  return (
    <div>
      <p>
        Current: {count}, Previous: {previousCountRef.current}
      </p>
      <button onClick={() => setCount(count + 1)}>+1</button>
    </div>
  );
}
```

The `useEffect`, which runs after every render, saves the current `count` value into `previousCountRef`. On the next render, this ref still holds the value from the PREVIOUS render -- because updating a ref doesn't trigger a new render on its own; the update only happens inside the render that's triggered by the `count` state changing.

Summary and Glossary

`useRef` is used for two main things: accessing a DOM element directly, or remembering a value across renders without triggering the screen to update. Unlike `useState`, changing a ref doesn't trigger a re-render -- so use `useState` for data that needs to be visible on screen, and `useRef` for "background" values that don't.

Glossary

`useRef` — The hook that lets you access a DOM element, or store a value across renders without triggering the screen to update.

`.current` — The field on the object `useRef` returns that holds the currently stored value.

DOM Reference — Direct access to a JSX element's real browser DOM node; set up with the `ref` attribute.