Documntaion

* Why do I need to use `.current` when working with `useRef` instead of directly modifying the object (e.g., `formRefs[refName] = el`)?

**Answer**

You need to use `.current` when working with `useRef` because `useRef` creates a mutable object with a specific property, `current`, that React guarantees to persist across renders. Here’s why:

**1.** **`useRef` Returns an Object:**

The `useRef` hook returns an object in the form of `{ current: ... }`. This is the designated location for storing your mutable data. For example:

```javascript

const formRefs = useRef({});

console.log(formRefs); // { current: {} }

```

**2. Direct Assignment (`formRefs[refName] = el`) Won’t Persist:**

If you try to modify `formRefs` directly, such as:

```javascript

formRefs[refName] = el;

```

it won’t persist across renders because React expects your data to live within `current`. The outer `formRefs` object itself is immutable from React’s perspective.

**3. `current` Is Stable Across Renders:**

React ensures the `current` property persists across renders without resetting or being replaced. It provides a stable and predictable place to store mutable references or values that React won’t interfere with during its reconciliation process.

**Key Example:**

**Incorrect Usage:**

```javascript

const formRefs = useRef({});

formRefs[refName] = el; // This modifies the outer object, not React's managed `current`

console.log(formRefs[refName]); // Appears to work but won't persist

```

**Correct Usage:**

```javascript

const formRefs = useRef({});

formRefs.current[refName] = el; // Properly stores it in the stable `current` object

console.log(formRefs.current[refName]); // Stable and persists across renders

```

**Why This Matters:**

React uses `current` as a "safe zone" to store mutable data. By relying on `current`, your data persists reliably between renders without triggering re-renders, ensuring consistent behavior.

* **How does this line of code solve my issue with `useRef`?**

```javascript

ref={(el) => (formRefs.current[refName] = el)}

```

**Answer:**

This line solves your issue because it dynamically assigns the DOM element (`el`) to the correct key (`refName`) inside `formRefs.current`. This ensures that each field's reference is correctly stored, accessible, and stable.

**Explanation of How It Works:**

**1. Dynamic Assignment:**

The `ref` callback `(el) => (formRefs.current[refName] = el)` is called whenever the component renders or unmounts. React passes the DOM element (`el`) to this callback. By using `formRefs.current[refName] = el`, the specific field's reference is stored in `current`.

- For example:

- If the `refName` is `emailEl`, the `email` input’s DOM element will be assigned to `formRefs.current.emailEl`.

**2. React Lifecycle Compliance:**

React ensures that this `ref` callback is invoked during:

- Mounting: `el` is set to the DOM element.

- Unmounting: `el` is set to `null`, clearing the reference.

**3. Avoids Null Access Errors:**

Previously, your `formRefs.current.emailEl` was `null` because React hadn’t assigned the DOM element yet. Using the callback guarantees the assignment happens after the DOM is available, allowing you to safely access `formRefs.current.emailEl.value`.

**Example Usage:**

For an input field like:

```javascript

<InputField

ref={(el) => (formRefs.current.emailEl = el)}

type="email"

/>

```

**-** **On Render:** `formRefs.current.emailEl` is set to the actual DOM element.

**- On Access**: In your `handleSubmit`, you can safely retrieve the value:

```javascript

console.log(formRefs.current.emailEl.value);

```

**Key Benefits:**

- Correctly stores the DOM references dynamically.

- Ensures refs are set only after the DOM is available.

- Avoids issues with `null` references and React’s render lifecycle.

By using `formRefs.current[refName] = el`, you create a clean, dynamic, and React-compliant way to handle multiple refs.

**Let Vs UseState Vs const**



###### When You Should Not Use let:

* If the Value Needs to Persist Across Renders: Use useState instead.
* If the Value Affects the UI: Use useState so React knows to re-render the component.
* If the Value Is Not Constant: Use const for values that don’t change.

###### Conclusion

* Use let for **temporary, local values** within a render cycle.
* Use useState for **dynamic, persistent values** that require reactivity and trigger UI updates.
* Use const for **static values** that won’t change.