A Thoughtful Way To Use React's `useRef()` Hook

QUICK SUMMARY ↬ *In a React component,* `useState` *and* `useReducer` *can cause your component to re-render each time there is a call to the update functions. In this article, you will find out how to use the* `useRef()` *hook to keep track of variables without causing re-renders, and how to enforce the re-rendering of React Components.*

In React components, there are times when frequent changes have to be tracked without enforcing the re-rendering of the component. It can also be that there is a need to re-render the component efficiently. While `useState` and `useReducer` hooks are the React API to manage local state in a React component, they can also come at a cost of being called too often making the component to re-render for each call made to the update functions.

In this article, I'll explain why `useState` is not efficient for tracking some states, illustrate how `useState` creates too much re-render of a component, how values that are stored in a variable are not persisted in a component, and last but not least, how `useRef` can be used keep track of variables without causing re-render of the component. And give a solution on how to enforce re-render without affecting the performance of a component.

After the evolution of functional components, functional components got the ability to have a local state that causes re-rendering of the component once there is an update to any of their local state.

```
function Card (props) {
  const [toggled, setToggled] = useState(false);

  const handleToggleBody  = () => {
    setToggled(!toggled)
  }

  return (<section className="card">
    <h3 className="card__title" onMouseMove={handleToggleBody}>
       {props.title}
    </h3>

    {toggled && <article className="card__body">
      {props.body}
    </article>}
  </section>)
}

// Consumed as:
<Card name="something" body="very very interesting" />
```

In the component above, a card is rendered using a `section` element having a child `h3` with a `card__title` class which holds the title of the card, the body of the card is rendered in an article tag with the body of `card__body`. We rely on the

`title` and `body` from the props to set the content of the title and body of the card, while the body is only toggled when the header is hovered over.

Scenario – mouseover event (Large preview)

More after jump! Continue reading below ↓

RE-RENDERING A COMPONENT WITH `useState`

Initial rendering of a component is done when a component has its pristine, undiluted state values, just like the Card component, its initial render is when the mouseover event is yet to be triggered. Re-rendering of a component is done in a component when one of its local states or props have been updated, this causes the component to call its render method to display the latest elements based on the state update.

In the `Card` component, the `mousemove` event handler calls the `handleToggleBody` function to update the state negating the previous value of the toggled state.

We can see this scenario in the `handleToggleBody` function calling the `setToggled` state update function. This causes the function to be called every time the event is triggered.

STORING STATE VALUES IN A VARIABLE

A workaround for the repeated re-rendering is using a *local variable* within the component to hold the toggled state which can also be updated to prevent the frequent re-rendering — which is carried out only when there is an update to local states or props of a component.

```
function Card (props) {
  let toggled = false;

  const handleToggleBody  = () => {
    toggled = !toggled;
    console.log(toggled);
  }

  return (<section className="card">
    <section className="cardTitle" onMouseMove={handleToggleBody}>
       {title}
    </section>

    {toggled && <article className="cardBody">
      {body}
    </article>}
  </section>)
}

<Card name="something" body="very very interesting" />
```

This comes with an unexpected behavior where the value is updated but the component is not re-rendered because no internal state or props has changed to trigger a re-render of the component.

Using variable in place of state (Large preview)

LOCAL VARIABLES ARE NOT PERSISTED ACROSS RERENDER

Let's consider the steps from initial rendering to a re-rendering of a React component.

- Initially, the component initializes all variables to the default values, also stores all the state and refs to a unique store as defined by the React algorithm.
- When a new update is available for the component through an update to its props or state, React pulls the old value for states and refs from its store and re-initializes the state to the old value also applying an update to the states and refs that have an update.
- It then runs the function for the component to render the component with the updated states and refs. This re-rendering will also re-initialize variables to hold their initial values as defined in the component since they are not tracked.
- The component is then re-rendered.

Below is an example that can illustrate this:

```
function Card (props) {

  let toggled = false;
```

```
  const handleToggleBody = () => {
    toggled = true;
    console.log(toggled);
  };

  useEffect(() => {
    console.log("Component rendered, the value of toggled is:", toggled);
  }, [props.title]);

  return (
    <section className="card">
      <h3 className="card__title" onMouseMove={handleToggleBody}>
        {props.title}
      </h3>

      {toggled && <article className="card__body">{props.body}</article>}
    </section>
  );
}

// Renders the application
function App () {

  const [cardDetails, setCardDetails] = useState({
```

```
    title: “Something”,
    body: “uniquely done”,
  });

  useEffect(() => {
    setTimeout(() => {
      setCardDetails({
        title: “We”,
        body: “have updated something nice”,
      });
    }, 5000); // Force an update after 5s
  }, []);

  return (
    <div>
      <Card title={cardDetails.title} body={cardDetails.body} />
    </div>
  );
}
```

In the above code, the `Card` component is being rendered as a child in the `App` component. The `App` component is relying on an internal state object named `cardDetails` to store the details of the card. Also, the component makes an update to the `cardDetails` state after 5seconds of initial rendering to force a re-rendering of the `Card` component list.

The `Card` has a slight behavior; instead of switching the toggled state, it is set to `true` when a mouse cursor is placed on the title of the card. Also, a `useEffect` hook is used to track the value of the `toggled` variable after re-rendering.

Using variable in place of state (second test) (Large preview)

The result after running this code and placing a mouse on the title updates the variable internally but does not cause a re-render, meanwhile, a re-render is triggered by the parent component which re-initializes the variable to the initial state of `false` as defined in the component. Interesting!

About `useRef()` Hook

Accessing DOM elements is core JavaScript in the browser, using vanilla JavaScript a `div` element with class `"title"` can be accessed using:

```
<div class="title">
  This is a title of a div
</div>
<script>
  const titleDiv = document.querySelector("div.title")
</script>
```

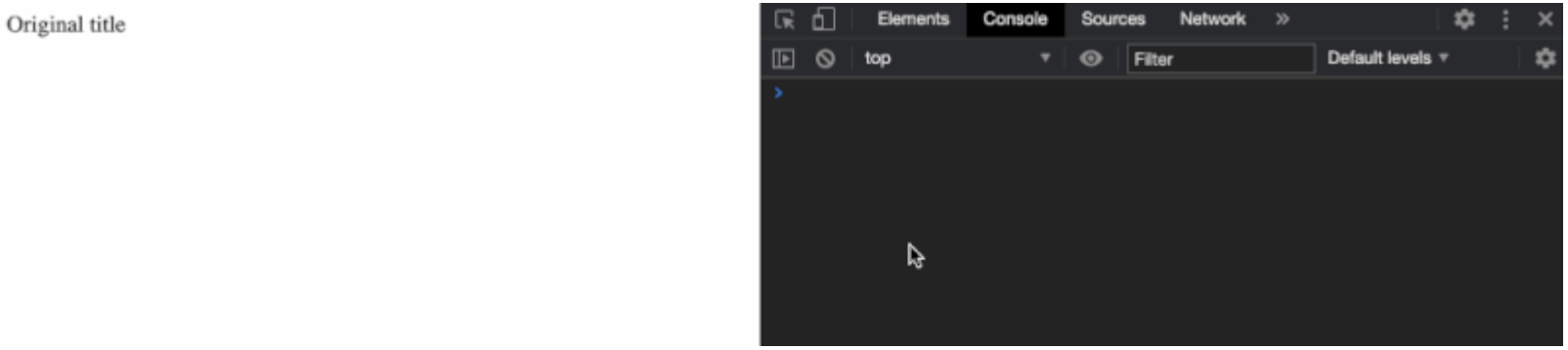
The reference to the element can be used to do interesting things such as changing the text content `titleDiv.textContent = "this is a newer title"` or changing the class name `titleDiv.classList = "This is the class"` and much more operations.

Overtime, DOM manipulation libraries like jQuery made this process seamless with a single function call using the `$` sign. Getting the same element using jQuery is possible through `const el = ("div.title")`, also the text content can be updated through the jQuery's API: `el.text("New text for the title div")`.

REFS IN REACT THROUGH THE `useRef` HOOK

ReactJS being a modern frontend library took it further by providing a Ref API to access its element, and even a step further through the `useRef` hook for a functional component.

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


export default function (props) {
  // Initialized a hook to hold the reference to the title div.
  const titleRef = useRef();


  useEffect(function () {
    setTimeout(() => {
      titleRef.current.textContent = "Updated Text"
    }, 2000); // Update the content of the element after 2seconds
  }, []);


  return <div className="container">
    {/** The reference to the element happens here **/ }
    <div className="title" ref={titleRef}>Original title</div>
  </div>
}
```



Using Ref to store state (Large preview)

As seen above, after the 2 seconds of the component initial rendering, the text content for the `div` element with the className of title changes to "Updated text".

HOW VALUES ARE STORED IN `useRef`

A Ref variable in React is a mutable object, but the value is persisted by React across re-renders. A ref object has a single property named `current` making refs have a structure similar to `{ current: ReactElementReference }`.

The decision by the React Team to make refs persistent and mutable should be seen as a wise one. For example, during the re-rendering of a component, the DOM element may get updated during the process, then it is necessary for the ref to the DOM element to be updated too, and if not updated, the reference should be maintained. This helps to avoid inconsistencies in the final rendering.

EXPLICITLY UPDATING THE VALUE OF A `useRef` VARIABLE

The update to a `useRef` variable, the new value can be assigned to the `.current` of a ref variable. This should be done with caution when a ref variable is referencing a DOM element which can cause some unexpected behavior, aside from this, updating a ref variable is *safe*.

```
function User() {
  const name = useRef("Aleem");

  useEffect(() => {
    setTimeout(() => {
      name.current = "Isiaka";
      console.log(name);
    }, 5000);
  });

  return <div>{name.current}</div>;
}
```

Storing Values In `useRef`

A unique way to implement a `useRef` hook is to use it to store values instead of DOM references. These values can either be a state that does not need to change too often or a state that should change as frequently as possible but should not trigger full re-rendering of the component.

Bringing back the card example, instead of storing values as a state, or a variable, a ref is used instead.

```
function Card (props) {


  let toggled = useRef(false);


  const handleToggleBody  = () => {
    toggled.current = !toggled.current;
  }


  return (
    <section className="card">
      <h3 className="card__title" onMouseMove={handleToggleBody}>
        {props.title}
      </h3>


      {toggled && <article className="card__body">{props.body}</article>}
    </section>
  );
  </section>)
}
```

This code gives the desired result internally but not visually. The value of the toggled state is persisted but no re-rendering is done when the update is done, this is because refs are expected to hold the same values throughout the

lifecycle of a component, React does not expect them to change.

SHALLOW AND DEEP RERENDER

In React, there are two rendering mechanisms, *shallow* and *deep* rendering. Shallow rendering affects just the component and not the children, while deep rendering affects the component itself and all of its children.

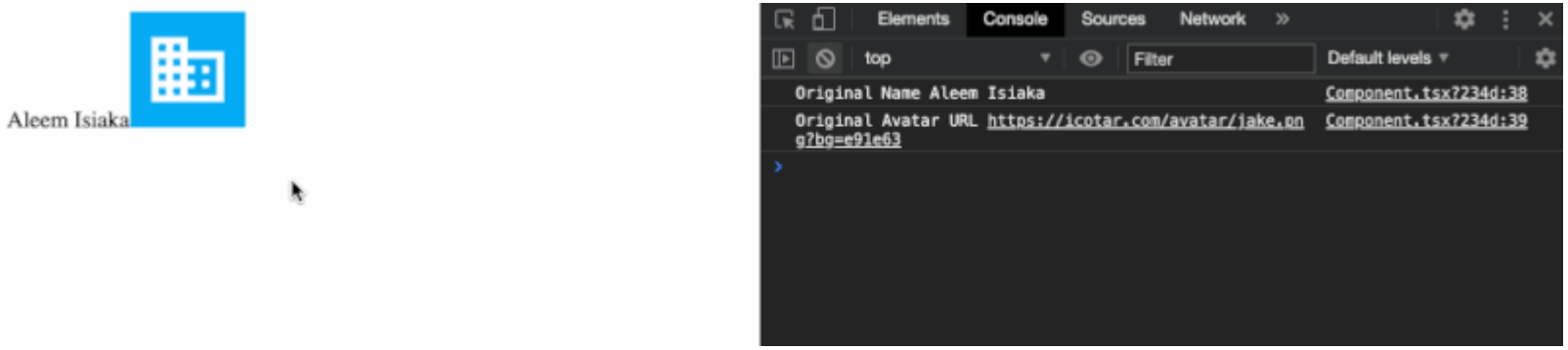
When an update is made to a ref, the shallow rendering mechanism is used to re-render the component.

```
function UserAvatar (props) {
  return <img src={props.src} />
}

function Username (props) {
  return <span>{props.name}</span>
}

function User () {
  const user = useRef({
    name: "Aleem Isiaka",
    avatarURL: "https://icotar.com/avatar/jake.png?bg=e91e63",
  })

  console.log("Original Name", user.current.name);
  console.log("Original Avatar URL", user.current.avatarURL);
```

```
  useEffect(() => {
    setTimeout(() => {
      user.current = {
        name: "Isiaka Aleem",
        avatarURL: "https://icotar.com/avatar/craig.png?s=50", // a new image
      };
    },5000)
  })

  // Both children won't be re-rendered due to shallow rendering mechanism
  // implemented for useRef
  return (<div>
    <Username name={user.name} />
      <UserAvatar src={user.avatarURL} />
  </div>);
}
```

In the above example, the user's details are stored in a ref which is updated after 5 seconds, the User component has two children, Username to display the user's name and `UserAvatar` to display the user's avatar image.

After the update has been made, the value of the `useRef` is updated but the children are not updating their UI since they are not re-rendered. This is shallow re-rendering, and it is what is implemented for useRef hook.

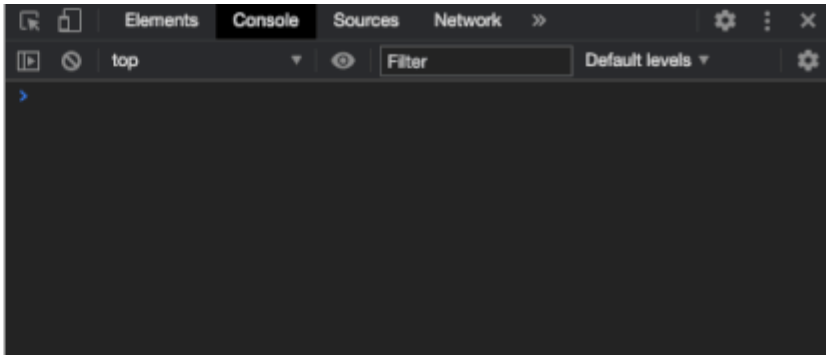


Shallow-rerender (Large preview)

Deep re-rendering is used when an update is carried out on a state using the `useState` hook or an update to the component's props.

```
function UserAvatar (props) {
  return <img src={props.src} />
}

function Username (props) {
  return <span>{props.name}</span>
}

function User () {
  const [user, setUser] = useState({
    name: "Aleem Isiaka",
    avatarURL: "https://icotar.com/avatar/jake.png?bg=e91e63",
  });
```

```
  useEffect(() => {
    setTimeout(() => {
      setUser({
        name: "Isiaka Aleem",
        avatarURL: "https://icotar.com/avatar/craig.png?s=50", // a new image
      });
    },5000);
  })

  // Both children are re-rendered due to deep rendering mechanism
  // implemented for useState hook
  return (<div>
    <Username name={user.name} />
      <UserAvatar src={user.avatarURL} />
  </div>);
}
```



Deep rerender (Large preview)

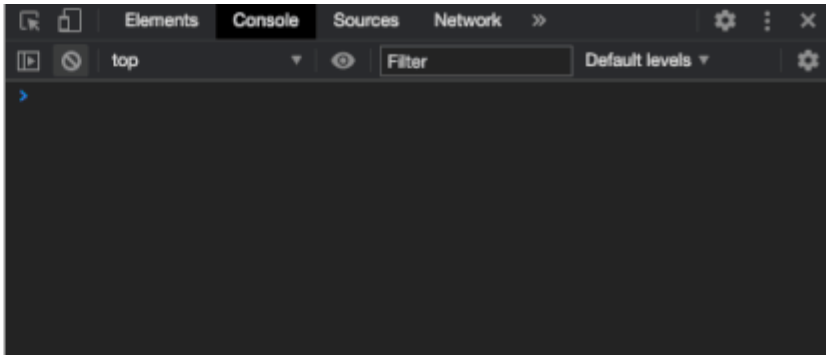
Contrary to the result experienced when `useRef` is used, the children, in this case, get the latest value and are re-rendered making their UIs have the desired effects.

FORCING A DEEP RE-RENDER FOR `useRef` UPDATE

To achieve a deep re-render when an update is made to refs, the deep re-rendering mechanism of the `useState` hook can be *partially* implemented.

```
function UserAvatar (props) {
  return <img src={props.src} />
}

function Username (props) {
  return <span>{props.name}</span>
}

function User () {
  const user = useRef({
    name: "Aleem Isiaka",
    avatarURL: "https://icotar.com/avatar/jake.png?bg=e91e63",
  })

  const [, setForceUpdate] = useState(Date.now());
```

```
  useEffect(() => {
    setTimeout(() => {
      user.current = {
        name: "Isiaka Aleem",
        avatarURL: "https://icotar.com/avatar/craig.png?s=50", // a new image
      };

      setForceUpdate();
    },5000)
  })
  return (<div>
    <Username name={user.name} />
      <UserAvatar src={user.avatarURL} />
  </div>);
}
```



Deep + Shallow rerender (Large preview)

In the above improvement to the `User` component, a state is introduced but its value is ignored since it is not required, while the update function to enforce a rerender of the component is named `setForceUpdate` to maintain the naming convention for `useState` hook. The component behaves as expected and re-renders the children after the ref has been updated.

This can raise questions such as:

> *"Is this not an anti-pattern?"*
>
> *or*
>
> *"Is this not doing the same thing as the initial problem but differently?"*

Sure, this is an anti-pattern, because we are taking advantage of the flexibility of `useRef` hook to store local states, and still calling `useState` hook to ensure the children get the latest value of the `useRef` variable current value both of which can be achieved with `useState`.

Yes, this is doing *almost* the same thing as the initial case but differently. The `setForceUpdate` function does a deep re-rendering but does not update any state that is acting on the component's element, which keeps it consistent across re-render.

Conclusion

Frequently updating state in a React component using `useState` hook can cause undesired effects. We have also seen while variables can be a go-to option; they are not persisted across the re-render of a component like a state is persisted.

Refs in React are used to store a reference to a React element and their values are persisted across re-render. Refs are mutable objects, hence they can be updated explicitly and can hold values other than a reference to a React element.

Storing values in refs solve the problem of frequent re-rendering but brought a new challenge of the component not being updated after a ref's value has changed which can be solved by introducing a `setForceUpdate` state update function.

Overall, the takeaways here are:

- We can store values in refs and have them updated, which is more efficient than `useState` which can be expensive when the values are to be updated multiple times within a second.
- We can force React to re-render a component, even when there is no need for the update by using a non-reference `useState` update function.
- We can combine 1 and 2 to have a high-performance ever-changing component.