useRef

Refs are something we already had in the previous versions of React, and the most common use case for them is to be able to access a DOM element and perform imperative actions on it, *jQuery style*.

This hook allows us to get values from a specific HTML element like its width, or even trigger actions in the HTML element. By using this, we can get access to the full DOM API as an escape hatch if we need to change the DOM without React intervening.

Let's import it and attach it somewhere so we can see how it works:

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

export default function App() {
  const button = useRef(null);

  useEffect(() => {
    console.log(button.current);
  }, []);

  return <button ref={button}>A button</button>;
}
```

You may be wondering why I `console.log` the `current` out of the button. That is because of the way React stores this. It always stores the last value of the `ref` in the current key of the `ref` object.

Your console should now have the HTML element as you would by using:

```
$("button");
```

And like you could in jQuery, we can click the button like so:

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

export default function App() {
  const button = useRef(null);

  useEffect(() => {
    button.current.click();
  }, []);

  return (
    <button ref={button} onClick={() => alert("I have been
clicked")}>
      A button
    </button>
  );
}
```

Like in the good ol' days, you will see an alert as soon the page is loaded.

In my last example, the button was clicked programmatically using the `ref` we placed on it.

We can take this even further by having two buttons and using `refs`. We can click on one button, which triggers a click on the other button.

Let's see this in code:

```
import React, { useRef, useState } from "react";
import "./styles.css";

export default function App() {
  const [count, setCount] = useState(0);
  const button = useRef(null);

  const fakeClick = () => {
    button.current.click();
  };

  return (
    <>
      <button onClick={fakeClick}>
        This button triggers a click somewhere else
      </button>
      Count: <span>{count}</span>
      <button ref={button} onClick={() => setCount(count +
1)}>
        +
      </button>
    </>
  );
}
```

If you click on the first button, you can see that it acts as if the increment button was clicked, and it will increase the counter.

You can also do some handy things with useRef like redirect user focus to where you want it to be, or get the dimensions of an element.

The idea of React is that it will handle DOM updates and manipulation for you, so please consider that when using `useRef` and use it with caution.

Link to CodeSandbox

As a final for hooks, let's also look at an interesting part about `useRef`: the fact that it's a mutable value that can hold anything in it's `current` property.

This idea may seem a bit confusing so let's test some code:

```
export default function App() {
  const count = useRef(0);
  const [countState, setCountState] = useState(0);

  useEffect(() => {
    count.current = 1;
    console.log("Ref " + count.current);
  }, []);

  useEffect(() => {
    setCountState(1);
    console.log("State " + countState);
  }, []);

  return <h1>Hello</h1>;
}
```

You can see we have two effects, one sets the value of `count.current` by accessing and mutating its value, and the next one uses `useState` to change that same value.

Both of these do a console.log after the value has been changed.

If we look at the console, we can see the following:

```
Ref 1
State 0
```

Interesting ah? :ponders-in-british:

The reason for that is that when you set setState in a react component, you trigger that component to run again with the new values it has, in this case, with the new state of `1`. So the `console.log` we have there doesn't actually get the new value since the effect doesn't run when that value changes. In the case of a `ref`, it's a mutable value, so it means that when you run, the value will be automagically updated with no re-render necessary, and that's why we can see `Ref 1` in the console.

You may be wondering what happens when we pass the `stateCount` as a dependency like so:

```
export default function App() {
  const count = useRef(0);
  const [countState, setCountState] = useState(0);

  useEffect(() => {
    count.current = 1;
    console.log("Ref " + count.current);
  }, []);

  useEffect(() => {
    setCountState(1);
    console.log("State " + countState);
  }, [countState]);
```

```
  return <>Hello</>;
}
```

In this case, we can see that what happens is that the component renders the same in the first run, but then since we have the dependency of the `countState`, the component re-renders, and only now can we see the new value.

I hope this makes sense and helps you understand a bit of the magic of `useRef`.

Link to CodeSandbox