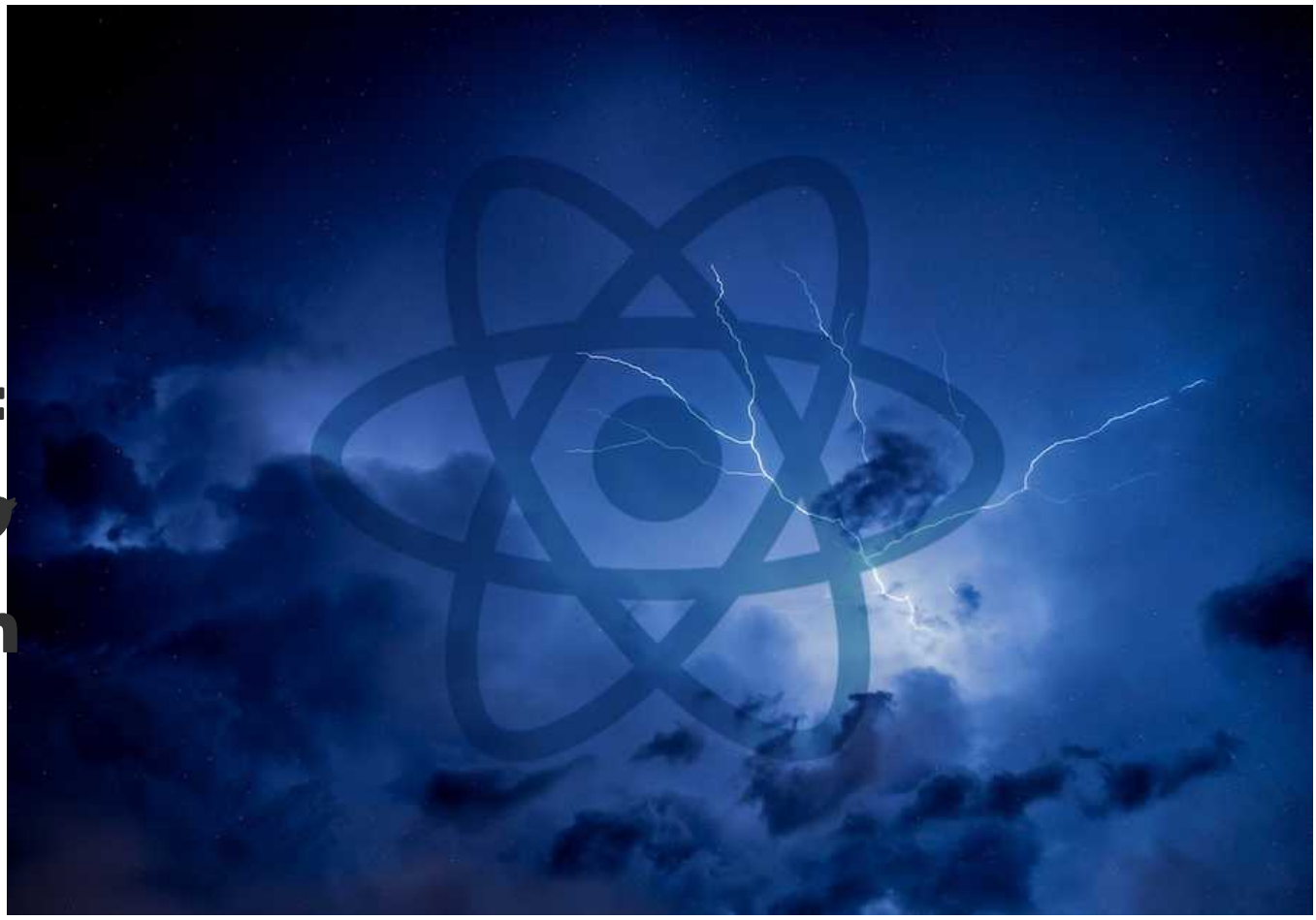
How to use React Ref

APRIL 25, 2020 BY ROBIN WIERUCH - EDIT THIS POST



Using React ref and really understanding it are two different pair of shoes. To be honest, I am not sure if I understand everything correctly to this date, because it's not as often used as state or side-effects in React and because its API did change quite often in React's past. In this React Ref tutorial, I want to give you a step by step introduction to refs in React.

REACT USEREF HOOK: REFS

React refs are strongly associated with the DOM. This has been true in the past, but not anymore since React introduced React Hooks. Ref means just reference, so it can be a reference to anything (DOM node, JavaScript value, ...). So we will take one step back and explore the React ref without the DOM

first, before diving into its usages with HTML elements. Let's take the following React component as example:

```
function Counter() {
  const [count, setCount] = React.useState(0);

  function onClick() {
    const newCount = count + 1;

    setCount(newCount);
  }

  return (
    <div>
      <p>{count}</p>

      <button type="button" onClick={onClick}>
        Increase
      </button>
    </div>
  );
}
```

React offers us the **React useRef Hook** which is the status quo API when using refs in React function components. The useRef Hook returns us a mutable object which stays intact over the lifetime of a React component. Specifically, the returned object has a `current` property which can hold any modifiable value for us:

```
function Counter() {
  const hasClickedButton = React.useRef(false);

  const [count, setCount] = React.useState(0);

  function onClick() {
    const newCount = count + 1;

    setCount(newCount);

    hasClickedButton.current = true;
  }

  console.log('Has clicked button? ' + hasClickedButton.current);

  return (
    <div>
      <p>{count}</p>

      <button type="button" onClick={onClick}>
        Increase
      </button>
    </div>
```

```
    );
  }
```

The ref's current property gets initialized with the argument we provide for the useRef hook (here `false`). Whenever we want, we can reassign the ref's current property to a new value. In the previous example, we are just tracking whether the button has been clicked.

The thing about setting the React ref to a new value is that it doesn't trigger a re-render for the component. While the state updater function (here `setCount`) in the last example updates the state of the component and makes the component re-render, just toggling the boolean for the ref's current property wouldn't trigger a re-render at all:

```
function Counter() {
  const hasClickedButton = React.useRef(false);

  const [count, setCount] = React.useState(0);

  function onClick() {
    // const newCount = count + 1;

    // setCount(newCount);

    hasClickedButton.current = true;
  }

  // Does only run for the first render.
  // Component does not render again, because no state is set anymore.
  // Only the ref's current property is set, which does not trigger a re-rer
  console.log('Has clicked button? ' + hasClickedButton.current);

  return (
    <div>
      <p>{count}</p>

      <button type="button" onClick={onClick}>
        Increase
      </button>
    </div>
  );
}
```

Okay, we can use React's useRef Hook to create a mutable object which will be there for the entire time of the component's existence. But it doesn't trigger a re-render whenever we change it -- because that's what state is for --, so what's the ref's usage here?

REACT REF AS INSTANCE VARIABLE

The ref can be used as **instance variable** for a function component in React whenever we need to track some kind of state without using React's re-render mechanism. For example, we can track whether a component has been rendered for the first time or whether it has been re-rendered:

```
function ComponentWithRefInstanceVariable() {
  const [count, setCount] = React.useState(0);

  function onClick() {
    setCount(count + 1);
  }

  const isFirstRender = React.useRef(true);

  React.useEffect(() => {
    if (isFirstRender.current) {
      isFirstRender.current = false;
    }
  });

  return (
    <div>
      <p>{count}</p>

      <button type="button" onClick={onClick}>
        Increase
      </button>

      {/*
        Only works because setCount triggers a re-render.
        Just changing the ref's current value doesn't trigger a re-render.
      */}
      <p>{isFirstRender.current ? 'First render.' : 'Re-render.'}</p>
    </div>
  );
}
```

In this example, we initialize the ref's current property with true, because we assume rightfully that the component starts with its first render when it gets initialized for the first time. However, then we make use of React's useEffect Hook -- which runs without a dependency array as second argument for the first and every additional render -- to update the ref's current property after the first render of the component. Setting the ref's current property to false doesn't trigger a re-render though.

Now we gain the ability to create a useEffect Hook which only runs its logic for every component update, but not for the initial render. It's certainly a feature which every React developer needs at some point but which isn't provided by React's useEffect Hook:

```
function ComponentWithRefInstanceVariable() {
  const [count, setCount] = React.useState(0);

  function onClick() {
    setCount(count + 1);
  }

  const isFirstRender = React.useRef(true);

  React.useEffect(() => {
    if (isFirstRender.current) {
      isFirstRender.current = false;
    } else {
      console.log(
        `
          I am a useEffect hook's logic
          which runs for a component's
          re-render.
        `
      );
    }
  });

  return (
    <div>
      <p>{count}</p>

      <button type="button" onClick={onClick}>
        Increase
      </button>
    </div>
  );
}
```

Deploying instance variables with refs for React components isn't widely used and not often needed. However, I have seen developers from my React workshops who surely knew that they needed an instance variable with useRef for their particular case after they have learned about this feature during my classes.

Rule of thumb: Whenever you need to track state in your React component which shouldn't trigger a re-render of your component, you can use React's useRef Hooks to create an instance variable for it.

REACT USEREF HOOK: DOM REFS

Let's get to React's ref speciality: the DOM. Most often you will use React's ref whenever you have to interact with your HTML elements. React by nature is declarative, but sometimes you need to read

values from your HTML elements, interact with the API of your HTML elements, or even have to write values to your HTML elements. For these rare cases, you have to use React's refs for interacting with the DOM with an imperative and not declarative approach.

This React component shows the most popular example for the interplay of a React ref and DOM API usage:

```
function App() {
  return (
    <ComponentWithDomApi
      label="Label"
      value="Value"
      isFocus
    />
  );
}

function ComponentWithDomApi({ label, value, isFocus }) {
  const ref = React.useRef(); // (1)

  React.useEffect(() => {
    if (isFocus) {
      ref.current.focus(); // (3)
    }
  }, [isFocus]);

  return (
    <label>
      {/* (2) */}
      {label}: <input type="text" value={value} ref={ref} />
    </label>
  );
}
```

Like before, we are using React's useRef Hook to create a ref object (1). In this case, we don't assign any initial value to it, because that will be done in the next step (2) where we provide the ref object to the HTML element as ref HTML attribute. React automatically assigns the DOM node of this HTML element to the ref object for us. Finally (3) we can use the DOM node, which is now assigned to the ref's current property, to interact with its API.

The previous example has shown us how to interact with the DOM API in React. Next, you will learn how to read values from a DOM node with ref. The following example reads the size from our element to show it in our browser as title:

```
function ComponentWithRefRead() {
  const [text, setText] = React.useState('Some text ...');

  function handleOnChange(event) {
```

```
    setText(event.target.value);
  }

  const ref = React.useRef();

  React.useEffect(() => {
    const { width } = ref.current.getBoundingClientRect();

    document.title = `Width:${width}`;
  }, []);

  return (
    <div>
      <input type="text" value={text} onChange={handleOnChange} />
      <div>
        <span ref={ref}>{text}</span>
      </div>
    </div>
  );
}
```

As before, we are initializing the ref object with React's useRef Hook, use it in React's JSX for assigning the ref's current property to the DOM node, and finally read the element's width for the component's first render via React's useEffect Hook. You should be able to see the width of your element as title in your browser's tab.

Reading the DOM node's size happens only for the initial render though. If you would want to read it for every change of the state, because that's after all what will change the size of our HTML element, you can provide the state as dependency variable to React's useEffect Hook. Whenever the state (here `text`) changes, the new size of the element will be read from the HTML element and written into the document's title property:

```
function ComponentWithRefRead() {
  const [text, setText] = React.useState('Some text ...');

  function handleOnChange(event) {
    setText(event.target.value);
  }

  const ref = React.useRef();

  React.useEffect(() => {
    const { width } = ref.current.getBoundingClientRect();

    document.title = `Width:${width}`;
  }, [text]);

  return (
    <div>
      <input type="text" value={text} onChange={handleOnChange} />
      <div>
```

```
          <span ref={ref}>{text}</span>
        </div>
      </div>
    );
  }
```

Both examples have used React's useEffect Hook to do something with the ref object though. We can avoid this by using callback refs.

REACT CALLBACK REF

A better approach to the previous examples is using a so called **callback ref** instead. With a callback ref, you don't have to use useEffect and useRef hooks anymore, because the callback ref gives you access to the DOM node on every render:

```
function ComponentWithRefRead() {
  const [text, setText] = React.useState('Some text ...');

  function handleOnChange(event) {
    setText(event.target.value);
  }

  const ref = (node) => {
    if (!node) return;

    const { width } = node.getBoundingClientRect();

    document.title = `Width:${width}`;
  };

  return (
    <div>
      <input type="text" value={text} onChange={handleOnChange} />
      <div>
        <span ref={ref}>{text}</span>
      </div>
    </div>
  );
}
```

A callback ref is nothing else than a function which can be used for the HTML element's ref attribute in JSX. This function has access to the DOM node and is triggered whenever it is used on a HTML element's ref attribute. Essentially it's doing the same as our side-effect from before, but this time the callback ref itself notifies us that it has been attached to the HTML element.

Before when you used the useRef + useEffect combination, you were able to run the your side-effect with the help of the useEffect's hook dependency array for certain times. You can achieve the same with the callback ref by enhancing it with React's useCallback Hook to make it only run for the first render of the component:

```
function ComponentWithRefRead() {
  const [text, setText] = React.useState('Some text ...');

  function handleOnChange(event) {
    setText(event.target.value);
  }

  const ref = React.useCallback((node) => {
    if (!node) return;

    const { width } = node.getBoundingClientRect();

    document.title = `Width:${width}`;
  }, []);

  return (
    <div>
      <input type="text" value={text} onChange={handleOnChange} />
      <div>
        <span ref={ref}>{text}</span>
      </div>
    </div>
  );
}
```

You could also be more specific here with the dependency array of the useCallback hook. For example, execute the callback function of the callback ref only if state (here `text`) has changed and, of course, for the first render of the component:

```
function ComponentWithRefRead() {
  const [text, setText] = React.useState('Some text ...');

  function handleOnChange(event) {
    setText(event.target.value);
  }

  const ref = React.useCallback((node) => {
    if (!node) return;

    const { width } = node.getBoundingClientRect();

    document.title = `Width:${width}`;
  }, [text]);

  return (
```

```
      <div>
        <input type="text" value={text} onChange={handleOnChange} />
        <div>
          <span ref={ref}>{text}</span>
        </div>
      </div>
    );
  }
```

However, then we would end up again with the same behavior like we had before without using React's useCallback Hook and just having the plain callback ref in place -- which gets called for every render.

REACT REF FOR READ/WRITE OPERATIONS

So far, we have used the DOM ref only for *read operations* (e.g. reading the size of a DOM node). It's also possible to modify the referenced DOM nodes (*write operations*). The next example shows us how to apply style with React's ref without managing any extra React state for it:

```
function ComponentWithRefReadWrite() {
  const [text, setText] = React.useState('Some text ...');

  function handleOnChange(event) {
    setText(event.target.value);
  }

  const ref = (node) => {
    if (!node) return;

    const { width } = node.getBoundingClientRect();

    if (width >= 150) {
      node.style.color = 'red';
    } else {
      node.style.color = 'blue';
    }
  };

  return (
    <div>
      <input type="text" value={text} onChange={handleOnChange} />
      <div>
        <span ref={ref}>{text}</span>
      </div>
    </div>
  );
}
```

This can be done for any attribute on this referenced DOM node. It's important to note that usually React shouldn't be used this way, because of its declarative nature. Instead you would use React's useState Hook to set a boolean whether you want to color the text red or blue. However, sometimes it can be quite helpful for performance reasons to manipulate the DOM directly while preventing a re-render.

Just for the sake of learning about it, we could also manage state this way in a React component:

```
function ComponentWithImperativeRefState() {
  const ref = React.useRef();

  React.useEffect(() => {
    ref.current.textContent = 0;
  }, []);

  function handleClick() {
    ref.current.textContent = Number(ref.current.textContent) + 1;
  }

  return (
    <div>
      <div>
        <span ref={ref} />
      </div>

      <button type="button" onClick={handleClick}>
        Increase
      </button>
    </div>
  );
}
```

It's not recommended to go down this rabbit hole though... Essentially it should only show you how it's possible to manipulate any elements in React with React's ref attribute with write operations. However, why do we have React then and don't use vanilla JavaScript anymore? Therefore, React's ref is mostly used for read operations.

. . .

This introduction should have shown you how to use React's ref for references to DOM nodes and instance variables by using React's useRef Hooks or callback refs. For the sake of completeness, I want to mention React's `createRef()` top-level API too, which is the equivalent of useRef() for React class components. There are also other refs called **string refs** which are deprecated in React.

Show Comments

KEEP READING ABOUT REACT

WHEN TO USE REACT'S REF ON A DOM NODE IN REACT

This tutorial is outdated. Please read over here everything you need to know about React Ref . In the past there has been a lot of confusion around the ref attribute in React. The attribute makes...

HOW TO USESTATE IN REACT

Since React Hooks have been released, function components can use state and side-effects. There are two hooks that are used for modern state management in React: useState and useReducer. This...

GET THE BOOK >

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✔ Join 50.000+ Developers
- ✔ Learn Web Development with JavaScript
- ✔ Tips and Tricks
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

Your email address SUBSCRIBE

View our Privacy Policy.

PORTFOLIO

Online Courses

ABOUT

About me

Open Source
Tutorials
What I use
How to work with me
How to support me

Contact Me Privacy & Terms