A guide to React refs: useRef and createRef

January 17, 2020 · 9 min read

In this article, we're going to investigate why React, a framework meant to abstract your code away from DOM manipulation, leaves the door open for developers to access it.

As is the case with many other UI libraries, React offers a way to rethink a view as the result of a state of a component.

This is a big pivot away from how we usually build applications.

When we become familiar with some of these new concepts, we discover how easy it is to solve simple problems in the frontend world that used to cause us some trouble.

Part of that benefit comes from creating the views with the abstraction mechanisms React and JSX expose instead of doing it through DOM spec methods.

Still, the React team did something smart that all library authors should do: they provided escape hatches and kept the library open for situations beyond the ones they were specifically designed for, as well as situations the model may not work for.

Creating refs

As I said, refs are escape hatches for React developers, and we should try to avoid using them if possible.

When we obtain a node by using a `ref` and later modify some attribute or the DOM structure of it, it can enter into conflict with React's diff and update approaches.

We're going to cover anti-patterns later in this article. First, let's start with a simple component and grab a node element using refs.

```
import React from 'react'

class ActionButton extends React.Component {

  render() {
    const { label, action } = this.props
    return (
      <button onClick={action}>{label}</button>
    )
  }
}
```

The `<button>` expression here is actually the JSX way of calling the `React.createElement('button')` statement, which is not actually a representation of an HTML Button element — it's a React element.

You can gain access to the actual HTML element by creating a React reference and passing it to the element itself.

```
import React, { createRef } from 'react'

class ActionButton extends React.Component {

  constructor() {
    super()
    this.buttonRef = createRef()
  }

  render() {
    const { label, action } = this.props
    return (
      <button onClick={action} ref={this.buttonRef}>{label}</button>
    )
  }
}
```

This way, at any time in the lifecycle of the component, we can access the actual HTML element at `this.buttonRef.current`.

But what about functions that act as components?

Recently, the React team released Hooks to pair them with the same features class components have.

We can now import `useRef` for refs inside function components as well.

```
import React, { useRef } from 'react'

function ActionButton({ label, action }) {
    const buttonRef = useRef(null)

    return (
      <button onClick={action} ref={buttonRef}>{label}</button>
    )
  }
}
```

We know how to access DOM nodes inside a React component. Let's take a look at some of the situations where this may be useful.

Usage of React refs

One of the many concepts that React expanded in the web sphere is the concept of declarative views.

Before declarative views, most of us were modifying the DOM by calling functions that explicitly changed it.

As mentioned at the introduction of this article, we are now declaring views based on a state, and — though we are still calling functions to alter this `state` — we are not in control of when the DOM will change or even if it should change.

Because of this inversion of control, we'd lose this imperative nature if it weren't for refs.

Here are a few use cases where it may make sense to bring refs into your code.

Focus control

You can achieve focus in an element programmatically by calling `focus()` on the node instance.

Because the DOM exposes this as a function call, the best way to do this in React is to create a ref and manually do it when we think it's suitable.

```
    return (
      <div className="modal--overlay">
        <div className="modal">
          <h1>Insert a new value</h1>
          <form action="?" onSubmit={this.onSubmit}>
            <input
              type="text"
              onChange={this.onChange}

              value={value}
            />
            <button>Save new value</button>
          </form>
        </div>
      </div>
    );
  }
}

export default InputModal;
```

In this modal, we allow the user to modify a value already set in the screen below. It would be a better user experience if the input was on focus when the modal opens.

This could enable a smooth keyboard transition between the two screens.

The first thing we need to do is get a reference for the input:

```
import React, { createRef } from "react";

class InputModal extends React.Component {
  constructor(props) {
    super(props);
    this.inputRef = createRef();

    this.state = { value: props.initialValue };
  }

  onChange = e => {
    this.setState({ value: e.target.value });
  };

  onSubmit = e => {
    e.preventDefault();
    const { value } = this.state;
    const { onSubmit, onClose } = this.props;
    onSubmit(value);
    onClose();
```

Next, when our modal mounts, we imperatively call focus on our input ref:

```
import React, { createRef } from "react";

    class InputModal extends React.Component {
      constructor(props) {
        super(props);
        this.inputRef = createRef();

        this.state = { value: props.initialValue };
      }

      componentDidMount() {
        this.inputRef.current.focus();
      }

      onChange = e => {
        this.setState({ value: e.target.value });
      };

      onSubmit = e => {
        e.preventDefault();
```

See this example in action.

Remember that you need to access the element through the `current` property.

Detect if an element is contained

Similarly, sometimes you want to know if any element dispatching an event should trigger some action on your app. For example, our Modal component could get closed if you click outside of it:

```
import React, { createRef } from "react";

class InputModal extends React.Component {
  constructor(props) {
    super(props);
    this.inputRef = createRef();
    this.modalRef = createRef();

    this.state = { value: props.initialValue };
  }

  componentDidMount() {
    this.inputRef.current.focus();

    document.body.addEventListener("click", this.onClickOutside);
  }

  componentWillUnmount() {
    document.removeEventListener("click", this.onClickOutside);
  }
```

See this example in action

Here, we are checking if the element click is out of the modal limits.

If it is, then we are preventing further actions and calling the `onClose` callback, since the Modal component expects to be controlled by its parent.

Remember to check if the DOM element current reference still exists as state changes in React are asynchronous.

To achieve this, we are adding a global click listener on the body element. It's important to remember to clean the listener when the element gets unmounted.

Integrating with DOM-based libraries

As good as React is, there are a lot of utilities and libraries outside its ecosystem that have been in use on the web for years.

It's good to take advantage of their stability and resolution for some specific problems.

GreenSock library is a popular choice for animation examples. To use it, we need to send a DOM element to any of its methods.

Using refs allows us to combine React with a great animation library.

Let's go back to our modal and add some animation to make its entrance fancier.

```
import React, { createRef } from "react";
import gsap from "gsap";

class InputModal extends React.Component {
  constructor(props) {
    super(props);
    this.inputRef = createRef();
    this.modalRef = createRef();
    this.overlayRef = createRef();

    this.state = { value: props.initialValue };

    const onComplete = () => {
      this.inputRef.current.focus();
    };
    const timeline = gsap.timeline({ paused: true, onComplete });
    this.timeline = timeline;
  }
  componentDidMount() {
    this.timeline
```

See this example in action.

At the constructor level, we are setting up the initial animation values, which will modify the styles of our DOM references. The timeline only plays when the component mounts.

When the element gets unmounted, we'll clean the DOM state and actions by terminating any ongoing animation with the `kill()` method supplied by the `Timeline` instance.

We'll turn our focus to the input after the timeline has completed.

Rule of thumb for refs usage

After knowing how refs work, it's easy to use them where they're not needed.

There's more than one way to achieve the same thing inside a React component, so it's easy to fall into an anti-pattern.

My rule when it comes to ref usage is this: you need to imperatively call a function for a behavior React doesn't allow you to control.

A simpler way to put it would be this: You need to call a function, and that function has no association with a React method or artifact.

Let's explore an anti-pattern that I've seen repeatedly in articles and even in interviews.

```
import React, { createRef } from 'react';

class Form extends React.Component {
  constructor(props) {
    super(props)
    this.inputRef = createRef()

    this.state = { storedValue: '' }
  }

  onSubmit = (e) => {
    e.preventDefault()
    this.setState({ storedValue: this.inputRef.current.value })
  }

  render() {

    return (
      <div className="modal">
        <form action="?" onSubmit={this.onSubmit}>
```

It’s fair to say if you want to send a value on submit, this approach will work.

The issue is that, knowing refs are actually an escape hatch of the view model React offers, we are sniffing into DOM element values or properties that we have access to through the React interface.

Controlling the `input` value we can always check its value.

```
render() {
  const { value } = this.state

  return (
    <input
      type="text"
      onChange={e => this.setState({ value: e.target.value })}
      value={value}
    />
  )
}
```

Let's go back to our rule: "You need to imperatively call a function for a behavior React doesn't allow you to control. "

In our uncontrolled input we are creating a ref but not doing an imperative call. Then that function should exist, which is not satisfied as I can indeed control an input's value.

Forwarding refs

As we've discussed, refs are actually useful for really specific actions. The examples shown are a little simpler than what we usually find in a web application codebase nowadays.

Components are more complex and we barely use plain HTML elements directly. It's really common to include more than one node to encapsulate more logic around the view behavior.

```
import React from 'react'

const LabelledInput = (props) => {
  const { id, label, value, onChange } = props

  return (
    <div class="labelled--input">
      <label for={id}>{label}</label>
      <input id={id} onChange={onChange} value={value} />
    </div>
  )
}

export default LabelledInput
```

The issue now is that passing a ref to this component will return its instance, a React component reference, and not the input element we want to focus on like in our first example.

Luckily, React provides an out-of-the-box solution for this called `forwardRef`, which allows you to define internally what element the `ref` will point at.

```
import React from 'react'

const LabelledInput = (props, ref) => {
  const { id, label, value, onChange } = props

  return (
    <div class="labelled--input">
      <label for={id}>{label}</label>
      <input id={id} onChange={onChange} value={value} ref={ref}/>
    </div>
  )
}

export default React.forwardRef(LabelledInput)
```

See this example in action

To achieve this, we'll pass a second argument to our function and place it in the desired element.

Now, when a parent component passes a ref value, it's going to obtain the input, which is helpful to avoid exposing internals and properties of a component and breaking its encapsulation.

The example of our form that we saw failing at achieving focus will now work as expected.

Using refs? Make sure they aren't breaking anything with LogRocket, a React monitoring solution.

Debugging React applications can be difficult, especially when users experience issues that are difficult to reproduce. If you're interested in monitoring and tracking Redux state, automatically surfacing JavaScript errors, and tracking slow network requests and component load time, try LogRocket.

LogRocket is like a DVR for web apps, recording literally everything that happens on your React app. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when an issue occurred. LogRocket also monitors your app's performance, reporting with metrics like client CPU load, client memory usage, and more.

The LogRocket Redux middleware package adds an extra layer of visibility into your user sessions. LogRocket logs all actions and state from your Redux stores.

Modernize how you debug your React apps — start monitoring for free.

Conclusion

We started with a recap on the basic concepts of React and its usage, why we generally shouldn't break the framework's model, and why we may sometimes need to.

Accessing the DOM through the interface the library exposes helps to maintain the internals of React in place (remember that `setState` contains more logic than just triggering a re-render cycle, like batching updates and in the near future, time slicing).

Breaking this model with anti-patterns can make later performance improvements in the library useless or even create bugs in your applications.

Remember to use refs only when there is an implicit function call React can't handle through its methods.

Also, make sure it doesn't alter the internal state of the components.

For more information, read the official React documentation about refs.

Jeremias Menichelli Follow

Curious mind, writer, and developer.

#react

4 Replies to "A guide to React refs: useRef and createRef"

Robert Werner Says:
January 17, 2020 at 12:45 pm

Reply

Thanks for this article, Jeremias. I use `useRef` and `createRef` extensively with my React code, which is entirely function, not class based, and extensively using Hooks. I've examined some `forwardRef` code and found it insanely complex and confusing so just used `window.###` functions instead.

Rafi Says:
August 18, 2020 at 10:02 pm

Reply

Awesome read, appreciate it man.

Christophe Hamerling Says:
April 9, 2021 at 8:43 am

Reply

Little typo in the last sample, should be

export default React.forwardRef(LabelledInput)

Matt Angelosanto Says:
April 9, 2021 at 12:10 pm

Reply

👍 Thanks for the catch

Leave a Reply

Enter your comment here...