How To Use an IntersectionObserver in a React Hook

Justin Travis Waith-Mair
Mar 23, 2019 · 9 min read ★

Photo by chuttersnap on Unsplash

(I would like to give credit, where I believe credit is due. The code examples in this post were written by me but were highly informed by code written by my former co-worker, Jared tuxsudo on twitter.)

One of the most difficult things to do on the web is figuring out if an element is visible or where an element is in relation to its parent element. Historically, this meant running calculations triggered by a scroll event, which can quickly become a performance liability for your app.

Luckily, a better and much more performant way to do this has been introduced: the Intersection Observer. The Intersection Observer API allows for asynchronous checking of the ratio of the intersection of an element with a viewport and will only fire a callback when the predefined threshold(s) are met. This has opened up many user experiences that were difficult to implement in a performant way, such as infinite scrolling, lazy load images, or delaying animations until visible.

Recently, I wanted to explore how one would go about implementing this in a react hook. I ran into many gotchas, but luckily Dan Abramov recently posted a very helpful guide to useEffect over at his blog, Overreacted, which helped me immensely in understanding these gotchas and what I needed to do to fix them. So I thought I would summarize what I learned to hopefully help you avoid the same mistakes I ran into.

How Does The Intersection Observer API Work?

In order to get a complete understanding of the Intersection Observer API, I would recommend that you check out the documentation found at MDN. Simply put, you need to create an Observer that will 'observe' a DOM node and call a callback when one or more of the thresholds are met. A threshold can be any ratio from 0 to 1 where 1 means the element is 100% in the viewport and 0 is 100% out of the viewport. By default, the threshold is set to 0. Here is an example of how to create an observer that I borrowed from MDN:

```
const callback = (entries, observer) => {
  entries.forEach(entry => {
    // Each entry describes an intersection change for one observed
    // target element:
    //   entry.boundingClientRect
    //   entry.intersectionRatio
    //   entry.intersectionRect
    //   entry.isIntersecting
    //   entry.rootBounds
    //   entry.target
    //   entry.time
  });
};

const observer = new IntersectionObserver(callBack);
```

Optionally, You can pass an object as a second parameter to the IntersectionObersver constructor. This object lets you configure the observer. You can configure 3 possible properties: root, rootMargin, and threshold.

The threshold property can either be a single ration, like `0.2` , or an array of thresholds, like `[0.01, 0.02, 0.03,....].` The `rootProperty` is the element to be used as the viewport when calculating the intersection ratio. The root property must be an ancestor to the element being observed and is the browser viewport by default. Finally, you can set the `rootMargin` property, using the CSS margin syntax, to specify an invisible box around the root by which the threshold is calculated.

So the above example could be rewritten like this:

```
const options = {
  root: domNode,
  rootMargin: '0px',
  threshold: [0.98, 0.99, 1]
}

const observer = new IntersectionObserver(callBack, options);
```

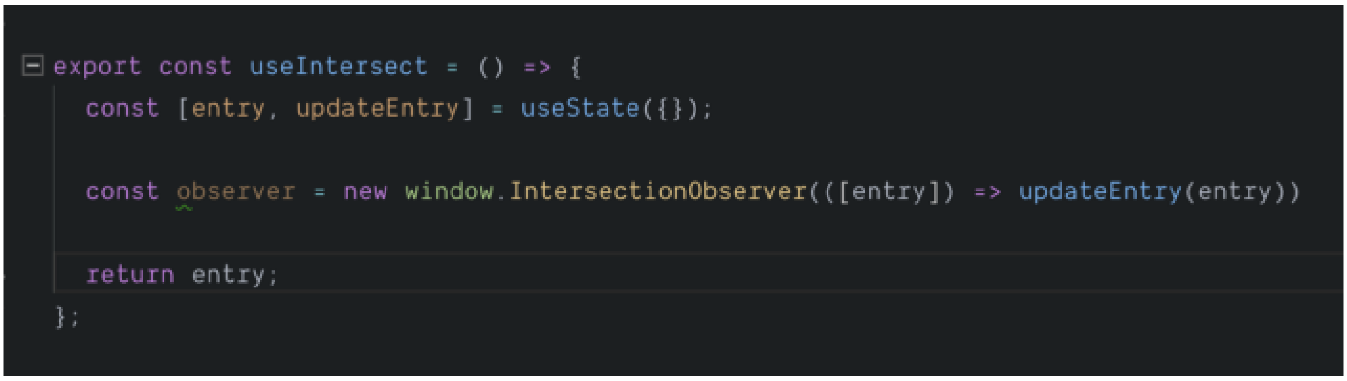
We have the observer, but it’s not yet observing anything. To start it observing, you need to pass a dom node to the `observe` method. It can observe any number of nodes, but you can only pass in one at a time. When you no longer want it to observe a node, you call the `unobserve` method and pass it the node that you would like it to stop watching or you can call the `disconnect` method to stop it from observing any node, like this:

```
observer.observer(nodeOne); //observing only nodeOne
observer.observer(nodeTwo); //observing both nodeOne and nodeTwo

observer.unobserve(nodeOne); //observing only nodeTwo

observer.disconnect(); //not observing any node
```

There are more things, but this covers the most typical use cases of the IntersectionObserver.

How do You Use It in a Hook?

First of all, we need to be able to provide the entry that the IntersectionObserver returns from the callback. To do this we use the `useState` hook. We are going to make the assumption that we are only going to be observing one node at a time, so we are going to destructure the entries array into the first entry into the array and save that to state, like this:



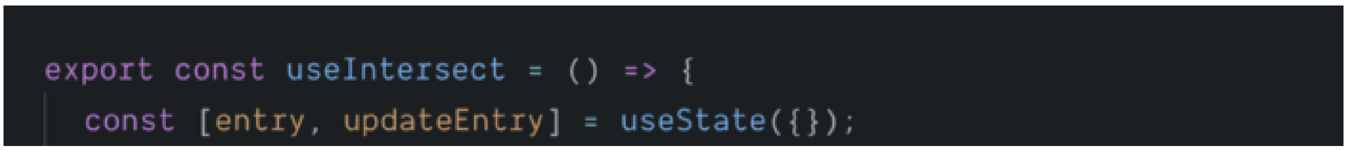
```
export const useIntersect = () => {
  const [entry, updateEntry] = useState({});

  const observer = new window.IntersectionObserver(([entry]) => updateEntry(entry))

  return entry;
};
```

There is already a big gotcha with this. Every time the component rerenders, useIntersect will be called, which means that the observer is going to be instantiated every time with a new IntersectionObserver. This is not the intended behavior.

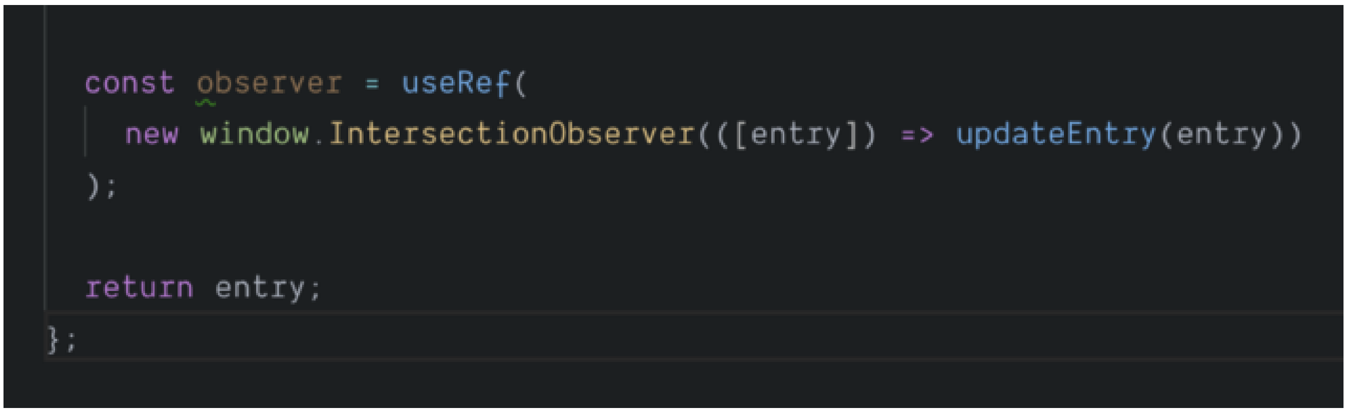
What we want to use is the `useRef` hook. The useRef hook is often used to keep track of a DOM node, so you can do imperative things with it later on (such as give it focus), but useRef can be used to keep any value across rerenders. We access the value of a ref through the `current` property on the ref itself. The ref itself is mutatable and that current value can be reassigned anytime, but we will always get back the same ref object with it's most recent value on every rerender.

One might ask, what is the difference between `useRef` and `useState` since both will return the current value. The biggest difference is how you update the value and what that means to the rest of the component using it. You can only update the state using the second value returned from `useState` where the ref's value can be updated anytime by assigning a new value to the current property. Also, updating the value of a ref will not signal a rerender, where updating the state will.

Let's update our hook to use `useRef`:

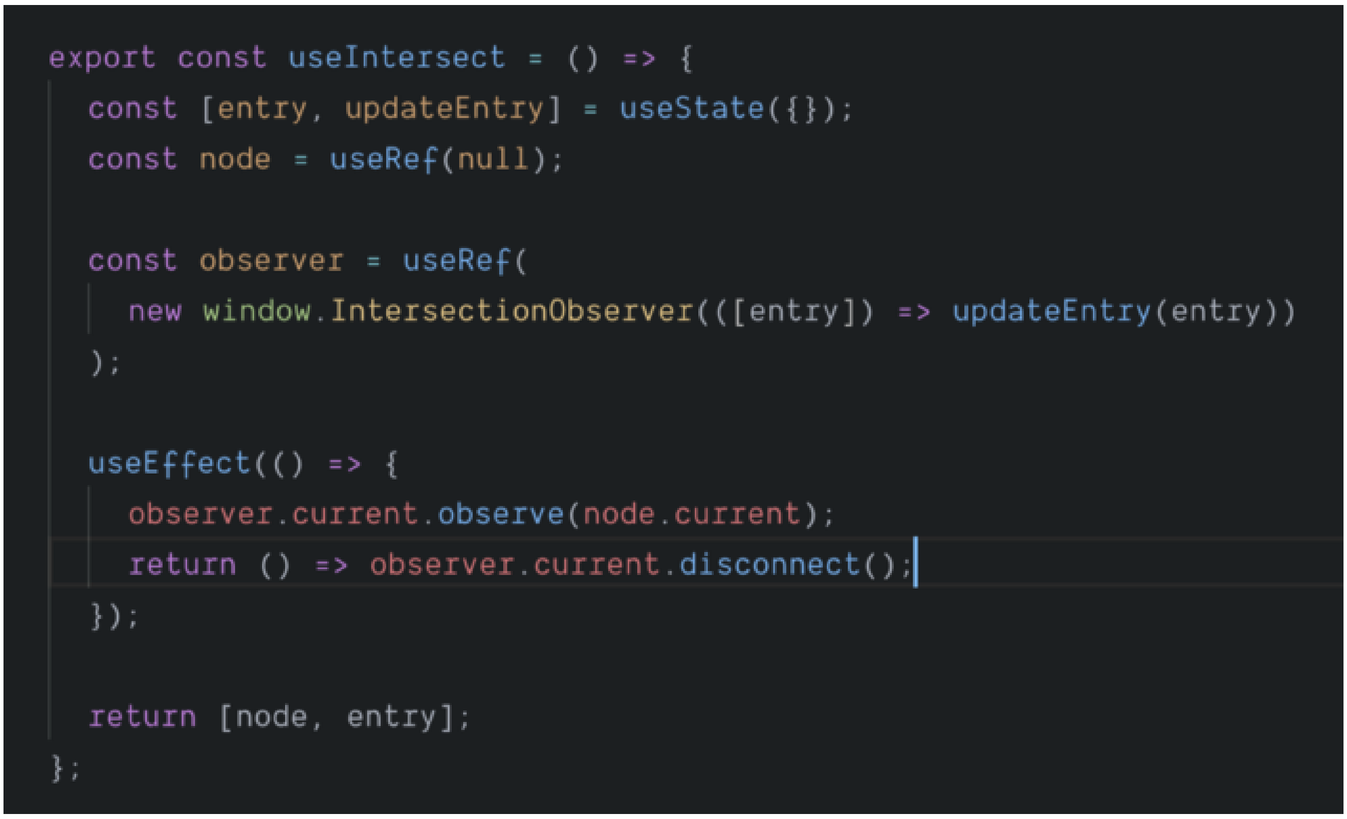


```
export const useIntersect = () => {
  const [entry, updateEntry] = useState({});
```



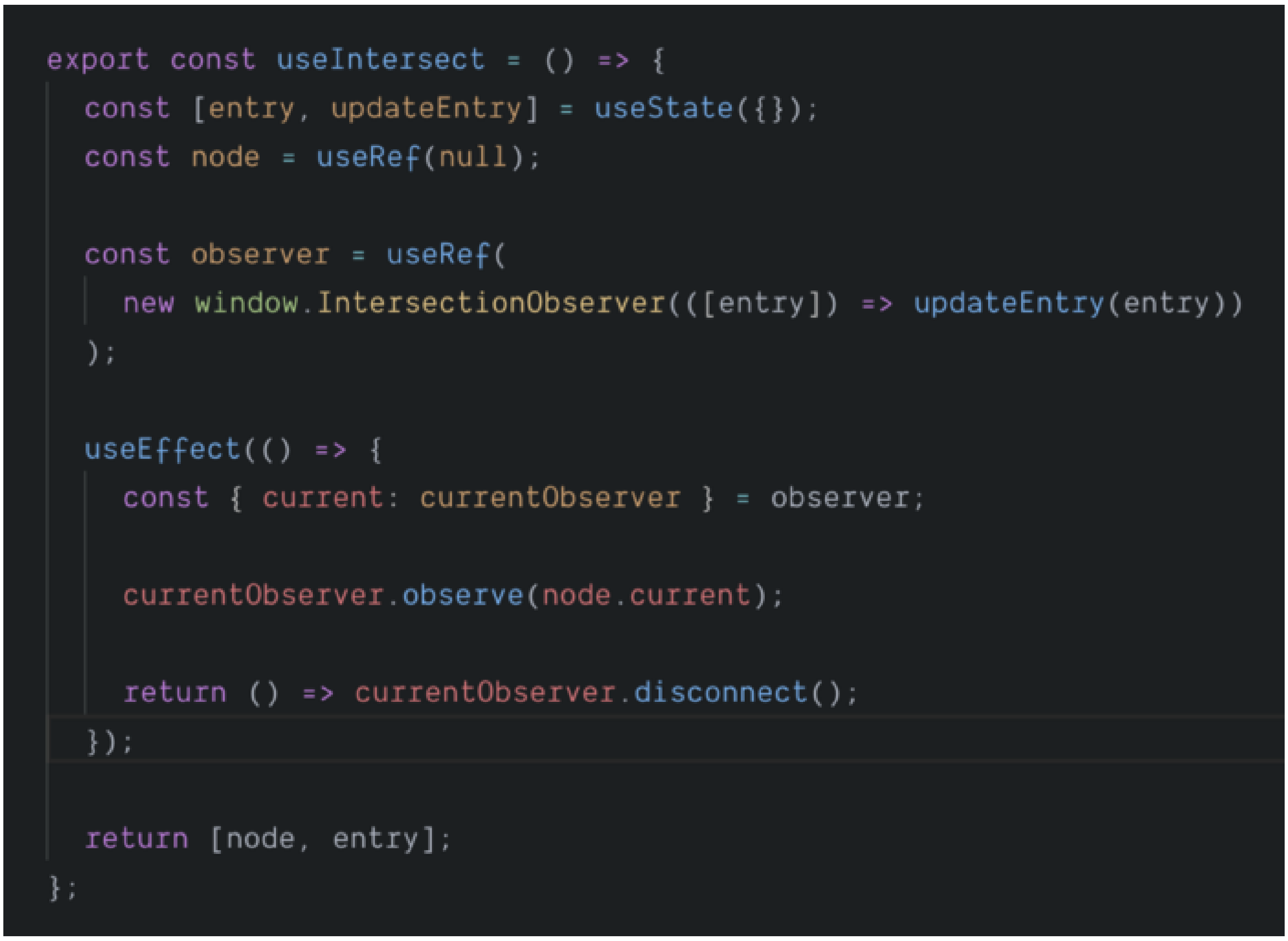
```
  const observer = useRef(
    new window.IntersectionObserver(([entry]) => updateEntry(entry))
  );

  return entry;
};
```

Our hook is coming together, but we are still missing the most important part: the observing. For we need two things: a node reference and we need to start observing it by using the `useEffect` hook. We might try to implement it like this:



```
export const useIntersect = () => {
  const [entry, updateEntry] = useState({});
  const node = useRef(null);

  const observer = useRef(
    new window.IntersectionObserver(([entry]) => updateEntry(entry))
  );

  useEffect(() => {
    observer.current.observe(node.current);
    return () => observer.current.disconnect();
  });

  return [node, entry];
};
```

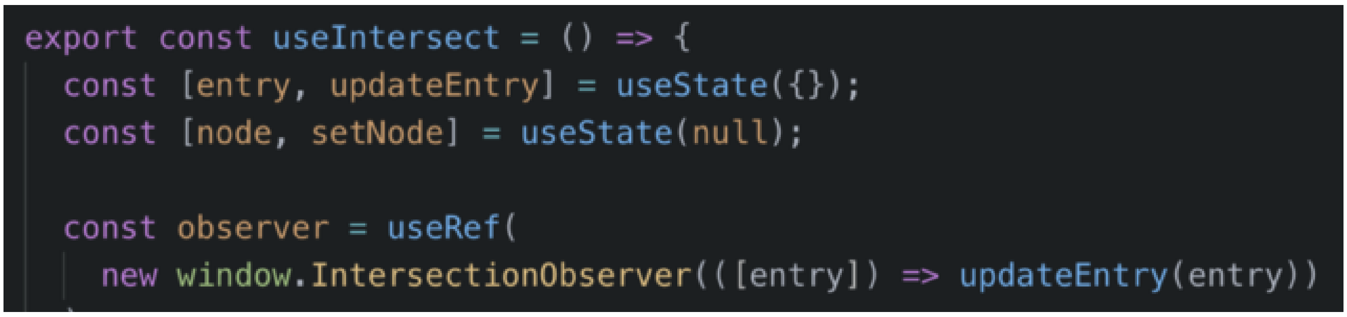
This already has many gotchas. The first gotcha wasn't obvious to me until I read more about how the `useEffect` hook worked. The function you return from the `useEffect` hook is run when the component is unmounting, that way you can clean up things like disconnecting the observer, which is what we are doing. The gotcha is that, since `current` can be mutated, it's not safe to just access it from the `current` property. If for some reason, in the future, I decided to reassign the `current` property to something else, like `null` or another observer, then the cleanup function may not actually clean up as I expect. The safe thing to do is assign the `current` property to a variable in the `useEffect` hook and then use the variable instead of the `current` property directly. Like this:



```
export const useIntersect = () => {
  const [entry, updateEntry] = useState({});
  const node = useRef(null);

  const observer = useRef(
    new window.IntersectionObserver(([entry]) => updateEntry(entry))
  );

  useEffect(() => {
    const { current: currentObserver } = observer;

    currentObserver.observe(node.current);

    return () => currentObserver.disconnect();
  });

  return [node, entry];
};
```

The other gotcha is that this effect is going to run over and over again because when the observer calls the callback, it will update the state, which will cause a rerender, which will cause the `useEffect` hook to be run again. The first thought would be to pass `node` in the dependency array for useEffect. The problem is that passing the ref has it's own 'lifecycle' and cannot be reliable to tell the `useEffect` hook to skip running this render.

To fix this we need to switch from using the `useRef` hook to the `useState` hook like this:



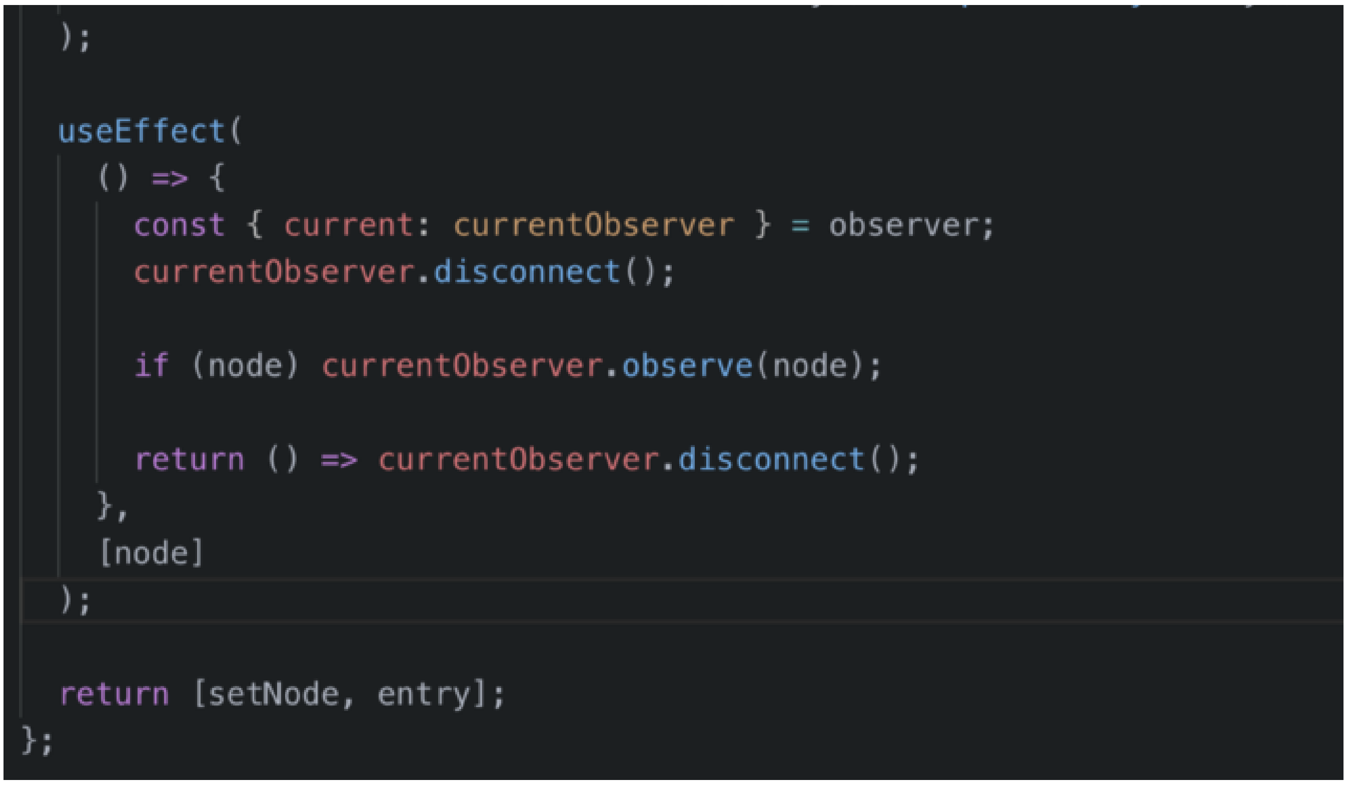
```
export const useIntersect = () => {
  const [entry, updateEntry] = useState({});
  const [node, setNode] = useState(null);

  const observer = useRef(
    new window.IntersectionObserver(([entry]) => updateEntry(entry))
  );

  useEffect(
    () => {
      const { current: currentObserver } = observer;

      if (node) currentObserver.observe(node);

      return () => currentObserver.disconnect();
    },
    [node]
  );

  return [setNode, entry];
};
```

By passing the `setNode` function, we are using the callback ref pattern instead of the new ref pattern (to learn more about the various ways to handle refs, you can read my beginner’s guide to refs in react post). This will pass the node into the callback we provide, which in our case will update the state to be the new node. This does mean that on the first pass, the `node` will be null and so we need to do a check to make sure the node has a value before we attempt to ‘observe’ it.

Things are looking great, but there are two more things that we need to consider. The first is, what happens if the component using the hook changes the node that the observer is ‘observing’? It will trigger a state change, since the `setRef` will be called on the new node. Since the node has changed, the `useEffect` will run again and it will start observing the new node. Sounds good right? What about the old node? Your right, we never stopped observing it. This means that there will be more than one entry in the callback and we may be saving the wrong entry to the state, not to mention that fact that we are observing nodes that we no longer care about.

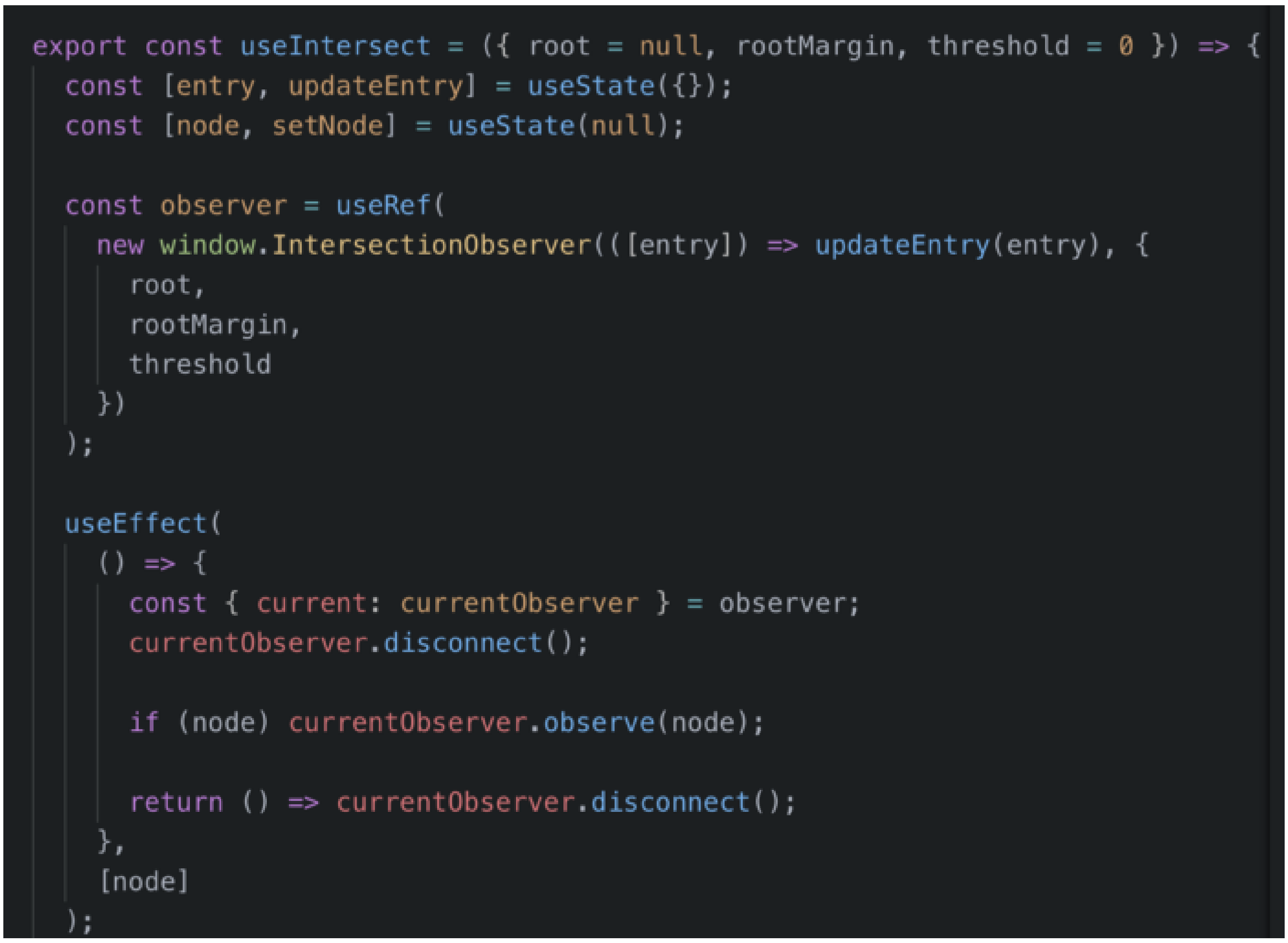
To fix this we need to disconnect the observer every time `useEffect` is called like this:

```
export const useIntersect = () => {
  const [entry, updateEntry] = useState({});
  const [node, setNode] = useState(null);

  const observer = useRef(
    new window.IntersectionObserver(([entry]) => updateEntry(entry))
```



```
  );

  useEffect(
    () => {
      const { current: currentObserver } = observer;
      currentObserver.disconnect();

      if (node) currentObserver.observe(node);

      return () => currentObserver.disconnect();
    },
    [node]
  );

  return [setNode, entry];
};
```

This will ensure that the observer is only looking at the node we care about. The last thing our hook needs is to be able to customize our observer. We can pass in the config object into the hook. We can provide some default values to the options so that we provide some safety to the way our hook runs. Here is the final version of our hook:



```
export const useIntersect = ({ root = null, rootMargin, threshold = 0 }) => {
  const [entry, updateEntry] = useState({});
  const [node, setNode] = useState(null);

  const observer = useRef(
    new window.IntersectionObserver(([entry]) => updateEntry(entry), {
      root,
      rootMargin,
      threshold
    })
  );

  useEffect(
    () => {
      const { current: currentObserver } = observer;
      currentObserver.disconnect();

      if (node) currentObserver.observe(node);

      return () => currentObserver.disconnect();
    },
    [node]
  );
  return [setNode, entry];
};
```

Now we have our fully functional custom hook, let's see it in action. I created a code sandbox using our new `useIntersect` hook. You can look through the code and see how it works, but basically I have several boxes, every other one will either fade or grow, based on its intersection ratio. You can toggle each box into either a fade or grow box and the hook properly accounts for the node changes.

Hooks offer a great way to compose functionality, like an Intersection Observer, into a component. Once you understand how works, you can do some amazing things.

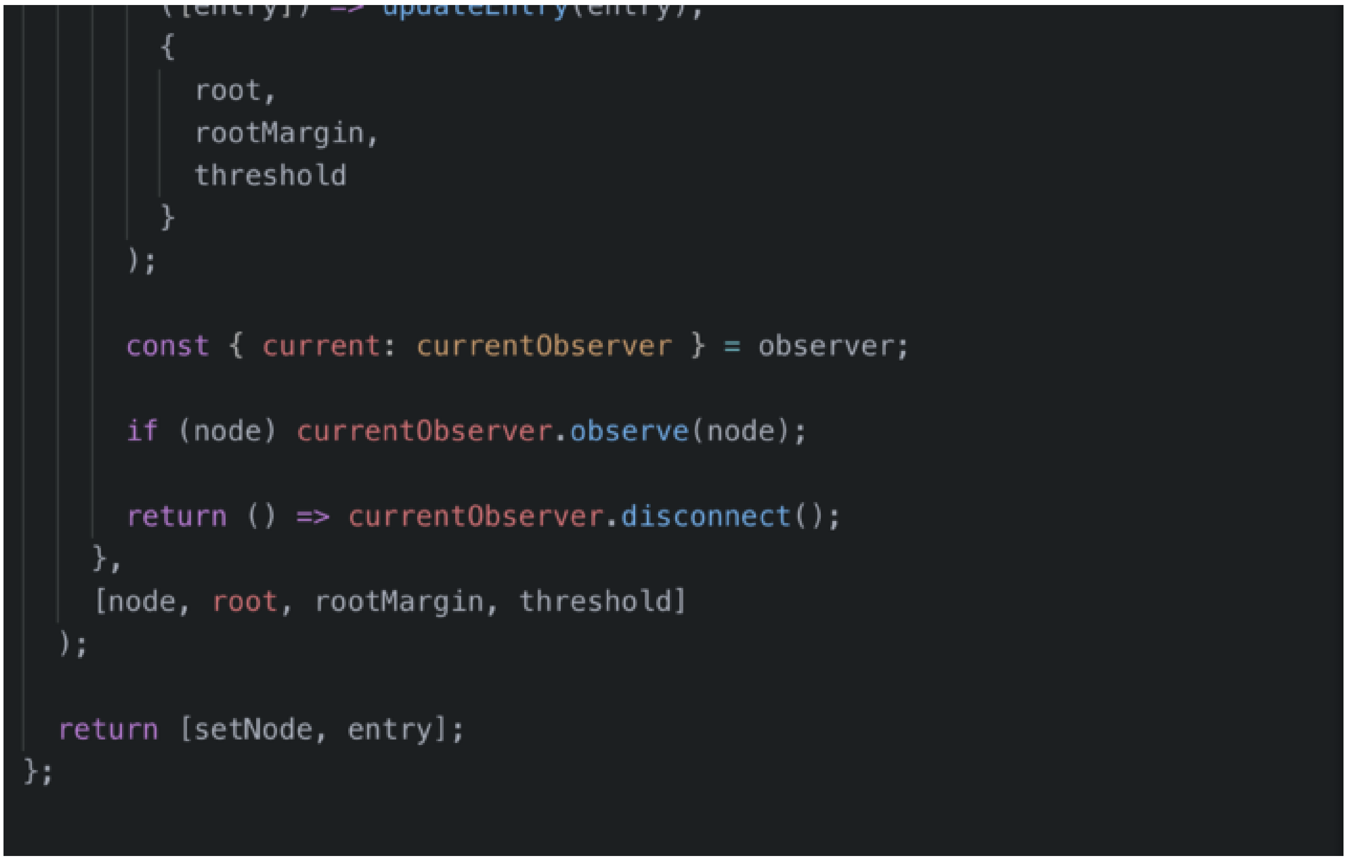
. . .

I have made an update based on the comments by Porfírio Ribeiro. There were two more gotchas that I didn't catch before. The first gotcha is that the hook doesn't update the observer if any of the config values change. The initial values are the only ones respected on the first render and then are effectively ignored every other render.

The other gotcha is related to the first one, which is that useRef will use whatever is passed in as the initial value the first time it's called, but will ignore it every other render. If it was a simple primitive value, that is no big deal, but we are constructing a new IntersectionObserver object every render, even though it is being ignored on all subsequent renders.

Luckily we can solve both problems at once by moving the construction of the IntersectionObserver to the useEffect function, like this:

```
export const useIntersect = ({ root = null, rootMargin, threshold = 0 }) => {
  const [entry, updateEntry] = useState({});
  const [node, setNode] = useState(null);

  const observer = useRef(null);

  useEffect(
    () => {
      if (observer.current) observer.current.disconnect();

      observer.current = new window.IntersectionObserver(
        ([entry]) => updateEntry(entry),
```



```
        ([entry]) => updateEntry(entry),
        {
          root,
          rootMargin,
          threshold
        }
      );

      const { current: currentObserver } = observer;

      if (node) currentObserver.observe(node);

      return () => currentObserver.disconnect();
    },
    [node, root, rootMargin, threshold]
  );

  return [setNode, entry];
};
```

As you can see, now that the observer could be null on the first render, so we are only disconnecting if the current value is not null. We are then constructing the IntersectionObserver and assigning it to current. After that, the hook works exactly the same. This allows us to then add the config options as dependencies.

It is important to note, that since threshold can be an array, it is important that the array passed in is the exact array every time. That is because arrays pass by reference and useEffect will only check if the reference changed in order to determine if it is should re-run the useEffect callback. In the code sandbox, I use `useMemo` with no dependencies to ensure that the array passed in is the same array every time.

JavaScript React React Hook Programming Software Engineering