Jake Trent

Re-render a React Component on Window Resize

Jake Trent

Oct 20, 2020 • 6 Min read • 81,526 Views

Oct 20, 2020 • 6 Min read • 81,526 Views

Web Development React

Introduction

141

Introduction

Most of the time, we attempt to create React apps that have flexible UIs, responding to the available visual space. Sometimes, however, this is neither possible or practical. In such instances, it can be useful to re-render a React component explicitly when the window or viewport size changes.

Listen for Resize

React doesn't have a resize event baked into it, but we can listen to the native browser `window` resize event from within our React component:

javascript

```
import React from 'react'
function MyComponent() {
  React.useEffect(() => {
    function handleResize() {
      console.log('resized to: ', window.innerWidth, 'x', window.innerHe

}

    window.addEventListener('resize', handleResize)
  })
  return <div>w00t!</div>
}
```

This code will simply listen for the window resize event and console log something like "resized to: 1024 x 768".

Re-render on Resize

But the above code will not yet re-render anything when the resize event is detected. We still have to tell React itself that something has changed in order to trigger a re-render.

Under normal conditions, React will re-render a component when its props or state changes. To trigger a re-render of `MyComponent` in the example, we'll set internal state on the component when the event is detected:

javascript

```
import React from 'react'
function MyComponent() {
  const [dimensions, setDimensions] = React.useState({
    height: window.innerHeight,
    width: window.innerWidth
  })
  React.useEffect(() => {
    function handleResize() {
      setDimensions({
        height: window.innerHeight,
        width: window.innerWidth
      })

}

    window.addEventListener('resize', handleResize)
  })
  return <div>Rendered at {dimensions.width} x {dimensions.height}</div>
}
```

Now we have set up some internal state, `dimensions`, that has height and width properties. Inside `handleResize`, we no longer simply `console.log`, but instead set new state when the resize is detected, using `setDimensions`. If we only cared about height or width resizes exclusively, we could track only what we needed.

Additionally, to show that a re-render is actually occurring, we've changed the output to print something like "Rendered at 1024 x 768".

Cleanup Listeners

When adding an event listener, such as we are for the resize event, we should make sure to clean up after ourselves. In the example so far, we haven't, and that could cause our app problems later.

React executes components multiple times, whenever it senses the need. And in each re-render, `useEffect` is going to be called again. This will create `n` new event bindings of `handleResize` to the resize event. If this component is re-rendered often, this could create a serious memory leak in our program. We only ever need or want one event listener. If we always clean up established event listeners before creating new ones, we'll ensure a single listener.

React gives us a way to do this with `useEffect`. When passing a function to `useEffect`, if *that* function also returns a function, that returned function will be called to perform any needed cleanup. We can put our `removeEventListener` code there:

javascript

```
import React from 'react'
function MyComponent() {
  const [dimensions, setDimensions] = React.useState({
    height: window.innerHeight,
    width: window.innerWidth
  })
  React.useEffect(() => {
    function handleResize() {
      setDimensions({
        height: window.innerHeight,
        width: window.innerWidth
      })

}

    window.addEventListener('resize', handleResize)

    return _ => {
      window.removeEventListener('resize', handleResize)

}
  })
  return <div>Rendered at {dimensions.width} x {dimensions.height}</div>
}
```

Now we're cleaned up nice and responsibly.

Resizing Less Often

Currently, our example code is set up to call `handleResize` as often as the window resizes. We're setting state and re-rendering for every single pixel change as often as the event loop will let us.

But what if there's a good reason to handling the resizing less often than that? We might want to be less aggressive in our re-rendering for performance reasons, such as in the case of a slow or expensive-to-render component.

In such a case, we can *debounce* the resize handling and thus the re-rendering. This will mean to throttle or wait between calls to our `handleResize` function. There are solid debounce implementations. Let's add a short and simple one to our example:

javascript

```
import React from 'react'

function debounce(fn, ms) {
  let timer
  return _ => {
    clearTimeout(timer)
    timer = setTimeout(_ => {
```

```
        timer = null
        fn.apply(this, arguments)
      }, ms)
    };
  }

  function MyComponent() {
    const [dimensions, setDimensions] = React.useState({
      height: window.innerHeight,
      width: window.innerWidth
    })
    React.useEffect(() => {
      const debouncedHandleResize = debounce(function handleResize() {
        setDimensions({
          height: window.innerHeight,
          width: window.innerWidth
        })
      }, 1000)

      window.addEventListener('resize', debouncedHandleResize)

      return _ => {
        window.removeEventListener('resize', debouncedHandleResize)

  }
    })
    return <div>Rendered at {dimensions.width} x {dimensions.height}</div>
  }
```

Note that we wrap `handleResize` in a `debounce()` call and bind the new function that it returns to the `debouncedHandleResize`

variable. Then we use this variable instead in both the event listener setup and cleanup.

The `debounce()` call has as its second parameter 1000ms, meaning that we are ensuring the `handleResize` code is called a maximum of once per second.

Conclusion

Bringing together the ability to listen to the native resize event, clean up after those event bindings, and control how often the event handler runs, we can now confidently re-render our React component in response to any viewport resize event.

To see this code in action, check out this running example.