The event listening toolkit: five ways to get out of an event handling mess

Baldur Bjarnason

—*I can't figure out how to remove this goddamn event listener?!*

One of the basic tenets of life is that you're *supposed* to clean up after yourself. Life, in general, gets much easier if you get into the habit of cleaning up as you go and don't let things pile up (*side-eyes own pile of laundry*). The same applies to coding, which is why pretty much every course, lesson, and book on web development will tell you that you should always remove your event listeners when you're done with them.

And in web development, there are a lot of event listeners, as event handling is pretty much the job:

1. Build markup.
2. Style it.
3. Render it in a browser, somehow.
4. Respond to events.
5. Go back to step 1, 2, or 3. Repeat until the end user refreshes the tab out of frustration.

That's web development.

But, because, event listening *is* the job and because there are so many events to listen to, your listeners quickly get out of hand. It's a common trial that binds us all together as web developers. React, Svelte, vanilla JS: we may work with events at different levels of abstraction, but our common bond is that we regularly fuck it up.

- Event listeners are, still to this day, a frequent source of memory leaks. How *do* you remove that goddamn event listener?
- Handling the same event from multiple elements can get complicated very quickly. "*How can I add an event listener to every single input field and make that listener target the correct context?*"
- Handling many events from a single element leads to complex spaghetti code. (So, so many switch statements and if-elses.)
- State management. Bloody state management.

It's a lot to deal with.

The way I deal with it is I have an event handling toolkit. These are patterns and tactics I’ve used over the years in dealing with the various event handling messes.

1

Don’t worry, the browser will take care of it all

We've all torn our hair out (figuratively, I hope) trying to remove event listeners in contexts where that turns out to be surprisingly difficult. Usually there is a simple reason behind it: it's often bloody hard to keep a reference to the event handling *function* around so you can remove it later on.

Turns out that it isn't always necessary.

The first tool in the event handling toolbox is to just not worry about it. Browsers are surprisingly good at collecting memory garbage as long as you follow certain principles and rules-of-thumb then coding.

If you:

- Need to listen to events on an element until it gets removed from the DOM. A click handler on a button element, for example.
- Know that the element's lifecycle is managed somewhere (that is, there isn't enything else that's keeping around a reference to it). Like a button in a component-rendered form.
- Attach the listener to the element directly.
- Don't reference that element in the function except via the `event.currentTarget` property, which always points to the element the listener was attached to.

Then you don't have to worry about it! Generally speaking.

That means that if you do this:

```
element.addEventListener(
  "click",
  (event) => {
    console.log("I'm attached to ", event.currentTarget);
  },
  false
);
```

Then you don't have to worry about removing the listener. The browser's garbage collection will take care of it.

—But how do I pass in, for example, a bookingId variable to the event listener function if I can’t refer to it? I’m working on a booking system not an interactive textbook. This is less useful than you think.

That’s why there are five tools in this toolkit. Different circumstances require different solutions and sometimes you need to carry around references to useful variables in your listeners.

2

‘Self-cleaning’ event handlers

One tactic is to use ‘self-cleaning’ event handlers, ones that remove themselves when you no longer need them. The simplest of these is the one and done: you need to listen to an event on an element and remove it after it’s done its job.

You only need to listen to this event once then remove it. A classic use case is waiting for a stylesheet to load or a CSS transition to end. Thankfully, you don’t need to do anything weird or hinky to do this. As long as you don’t have to support an ancient browser, you can use:

```
element.addEventListener("load", (event) =>
        console.log(event), { once: true });
```

… and the browser will take care of removing the listener after it has been fired once.

And if you need to support older browsers that do support `addEventListener` but not the options object, read on.

If you need to support older browsers or more complex conditions than “remove this after it fires once” then you need a slightly different tactic:

This is a click handler that removes itself after the third time the button has been clicked.

```
// First lets grab us the first button we find
// `var` because we're targeting crappy old browsers
var button = document.querySelector("button");
var count = 0;
button.addEventListener(
  "click",
  function clicker(event) {
    // I don't use += because it's incredibly easy to
    // make mistakes with it, either by not noticing
    //it while reading code or by mistyping it as `=+`
    count = count + 1;
    if (count === 3) {
      // event.currentTarget refers to the element
      // the listener was added to.
      // And because we gave the function a handy
      // name, we can use that to refer to it even
      // while in the function's scope
      event.currentTarget.removeEventListener("click", click-
        er);
    }
  },
  false
); // The useCapture argument didn't always default to
        `false` in every browser.
```

Note that you can't use an arrow function here because they are all anonymous and you need the function to have a name so it can refer to itself.

This tactic means you don't have to keep a reference around to the event handling function, which can get complicated very quickly.

It does use a closure to capture an external variable (`count`) and keeps a reference to it for the lifetime of the fuction.Those are a frequent source of memory leaks, but it won't cause a problem because we're cleaning up after ourselves. This is exactly the sort of thing you'd be trying to avoid when you're relying on the browser's garbage collection to handle things.

Remember to always use `event.currentTarget` to refer to the element whose events you are listening to and not `event.target`. Sometimes the two are the same, for example when it's a button element with only text node children. But if the element has any markup you might end up trying to remove an event listener from a random `path` element in an embedded SVG icon and you're back to your old memory leaking ways.

But what if you have a more complex event handling problem? What if you have an actual mess on your hands, not the minor untidiness of the occasional one-and-done event handler?

3

Delegate, delegate, delegate

—*"How can I add an event listener to every input element and make that listener work on the correct context?"*

A regular task we run into as devs is having to handle events in multiple contexts on a page at the same time. Often people reach for something like `querySelectorAll` and simply add the event listener to potentially thousands of elements at a time, which, if your event handler captures external variables, can very quickly lead to a serious memory leak.

The tactic I tend to use here, and which is a perennial favourite in web dev, is *delegation*.

Instead of listening on the elements that are firing the events, you listen to a shared parent element that's a common root to them all and then check to see if the `event.target` (who's firing the event) matches the selector you're expecting. Then you use the `event.target` reference to that element to access its context. That way you can make sure you're operating in the immediate environment where the event took place.

```
// Let's add a fun function that logs the number of
// radio buttons in the each button group
function logRadioSiblings(event) {
  // `.matches` lets you check if an element
  // "matches" a selector
  if (event.target.matches('input[type="radio"]')) {
    // We want a glimpse of the input's context so we
    // find the closest `fieldset` parent.
    //
    // This is assuming we wrapped all related inputs
    // in a single fieldset as we should.
    const fieldset = event.target.closest("fieldset");
    console.log(fieldset.querySelectorAll('in-
        put[type="radio"]').length);
    // Logs however many radio buttons are in the fieldset
  }
}

// Add this event listener to the `main` element.
// You want to add the delegate listener to the closest pos-
        sible parent.
// The more 'layers' you have between the event sources and
        the event listener the
// likelier it is, as the app grows, that some unrelated com-
        ponent in your hierarchy
// calls `event.stopPropagation()` on you and everything
        breaks on you.
// "No matter what I do, the event never reaches my handler!"
// Been there; torn out the hair. Don't be me.
document.querySelector("main").addEventListener("change", lo-
        gRadioSibling);
```

That last comment in the code sample is an important issue. Events are how web apps work and it's really easy for one component in the hierarchy to stop an event and break things for the rest. If your form component adds an event listener on the document root but is embedded in a tab component that stops the propogation of the click event before it reaches the root, then your form's handler will never see it —even if the click event originated in the form! That's why you should always try to operate as close to the event sources as you can. Write the delegate as a parent component that only concerns itself with its child elements. Only add delegate listeners to the root element or document if you're implementing a fallback behaviour that other components are *supposed* to override.

Adding the delegate event handler to the nearest relevant component root also simplifies the selectors you need to write for `event.target.matches`.

(Shadow DOM was supposed to address this issue, amongst other things. Which it did by making everything much more complicated, less compatible, and broke a ton of APIs in the process, esp. ones that involve text selection. In my experience Shadow DOM has caused more problems than it created. It's a very useful tool when you need it —and when you need it you *really* need it—but it shouldn't be your default.)

The delegate pattern is extremely useful and it's tempting to use a single generic delegate system to handle all of your events. After all, that's essentially what React and jquery do. It's a long tradition.

These frameworks delegate for a good reason but it has also caused problems. The "*no matter what I do, the event never reaches my handler!*" issue is something a lot of React devs have encountered, not just dummies like me. It has caused so many problems that they changed how they did event delegation in React 17 to address it. Instead of setting the delegate listening function on the document root, which regularly lead to issues when you embedded two or more separate React components on a page, it is now set on the root container React renders to.

Essentially, they're adding the delegate listener to the closest possible parent element. Like I'm telling you to do. Don't listen to me; listen to the React team.

(There are a few reasons why some frameworks use delegate systems for all their event handling by default but a big one is that they need to support a much broader selection of browsers than you or I do. Historically, event handling hasn't exactly been uniform across the board. Delegation lets them address all of those differences in a single place.)

One *major* pitfall to overusing the delegate pattern is that it can make memory leaks likelier. As you saw in the first solution, browsers are pretty good at cleaning up event listeners provided you don't capture any external variables and attach it directly to the element that you're listening to. But if you're using a delegate you risk capturing your entire component in the event listener on an element that won't get garbage collected for the entire lifetime of the app (like the root document).

There are two problems that delegates *don't* help you with:

1. Listening to many events on a single element.
2. Any event handling that involves state management.

For the former problem, and occasionally the latter, I like to use the EventListener interface.

4

The EventListener

A frequent pattern in modern web dev is to wrap up state and event management into components. The component responds to events and manages current state in response to those events.

This is a frequent source of memory leaks as observed by Nolan Lawson in an excellent article on, well, memory leaks.

> Modern web app frameworks like React, Vue, and Svelte use a component-based model. Within this model, the most common way to introduce a memory leak is something like this:
>
> ```
> window.addEventListener("message",
>         this.onMessage.bind(this));
> ```
>
> That's it. That's all it takes to introduce a memory leak. If you call addEventListener on some global object (the window, the <body>, etc.) and then forget to clean it up with removeEventListener when the component is unmounted, then you've created a memory leak.

The problem here is that every time you call `this.onMessage.bind(this)` you get a new function, so `removeEventListener` won't work.

You can fix this in a few ways.

- A wrapper function that doesn't need binding and forwards the call to the component.
- You can add an arrow function as the class method, which is then the listener, but those come with a ton of their own issues.
- You could store the newly created bound function somewhere to remove later, but that gets unweildy quickly once you start using more components.

—*See, this is why web development sucks. Best practices just don't work for most projects. Just let the memory leak a bit. Won't hurt anybody.*

Don’t mistake common practices with best practices! My favourite tool for this problem, for example, has been a part of the web platform—widely supported even!—for over twenty years but very few devs seem to know of its existence: the venerable `EventListener` pattern.

What if I told you that you could use the component itself, directly, as an event listener both when adding and removing?

That’s what the platform supports!

```
// Normally you'd be subclassing whatever component system
        you're using
class MyComponent {
  handleEvent(event) {
    if (event.type === "click") {
      console.log("clicked!");
    }
  }
}
const myinstance = new MyComponent();
document.body.addEventListener("click", myinstance);
document.body.click();
// clicked!
document.body.removeEventListener("click", myinstance);
```

You can add objects as event listeners as long as they have a `handleEvent` method.

Any object will do. Any class instance will do. It doesn’t matter what framework you use or transpiler you rely on, if it results in a JavaScript object somewhere, you can give it a `handleEvent` and you can use it directly as an event listener.

This solves a lot of problems because most of our frameworks, tools, and web app architectures make it much easier to keep track of these components than it is to manage a bunch of anonymous or (semi-anonymous) event handling functions.

This gets really useful if you need the same component to respond to a lot of different events. You use `handleEvent` to route those calls to the appropriate method.

```
class MyComponent {
  handleEvent(event) {
    if (typeof this[`on${event.type}`] === "function") {
      this[`on${event.type}`](event);
    }
  }
  onclick(event) {
    console.log("clicked!");
  }
  onchange(event) {
    console.log("changed!");
  }
}
const myinstance = new MyComponent();
document.body.addEventListener("click", myinstance);
document.body.click();
// clicked!
document.body.removeEventListener("click", myinstance);
```

If you're managing state locally in that component this lets your event handling methods work with the component object, via `this`, instead of as a captured variable and without function binding.

But what happens when you need to handle events and manage shared state across a number of components? Y'know... the exact thing that is the root source of so, so many bugs on the web?

5

The Observable pattern and contract

This can get complicated.

The final pattern is also the one that I use the least, but when I need it I really need it. I only really began to use it properly when I first started to use Svelte, which has first class support for it. But it predates both Svelte and its use in the JavaScript community by several years: the Observer design pattern. It's one of the original "Gang of Four" design patterns and so dates back to the early 90s.

You can think of them as being to recurring events what promises are to single events. It's an abstraction that treats a stream of events as a value.

Or, in more understandable terms: it's an object you can subscribe to in order to get its current and future values.

```
const observable = new Observable.from(["bling", "blang",
        "bloom"]);
observable.subscribe((x) => {
  console.log(x);
  // Logs bling blang bloom
});
```

Now, of course, since this is JS and the web, you'll note that they couldn't just call it "the Observer design pattern".

Obviously.

Svelte calls them `stores` and has amazing, first class, support for reactive component rendering based on them.

Svelte makes it really simple for you to get the current and future values of an Observable/store. Prepend the variable name with a `$` in a component script and you'll not only get the current value but the script *will run again* for every future value of that store.

That's really handy.

The JS community in general calls them 'observables'. There's been an ongoing effort to add native support for them to the language for a while but, honestly, that isn't necessary. It was first really popularised by RXJS.

Well, "popularised" isn't really the right word. RXJS has a reputation for being complex, hard to understand, and hard to use.

A deserved reputation, to be honest. I've used RXJS in a project to great effect, it made an impossible problem possible, but getting your head around it isn't trivial. It's a great tool when you need it. Which is hopefully not that often.

In many ways RXJS was the worst thing to happen to Observables and if Svelte hadn't resurrected the pattern (or, more specifically, adjusted their own proposed store pattern to match that of plain Observables) it almost certainly would have just ended up being a niche solution you'd hope you never have to use.

Thankfully, you don't need to use RXJS to implement one. You don't even need to use the store tools that Svelte provides.

I find it most useful to adhere to the Svelte variant of the Observable contract even when I'm not working in Svelte because you never know.

Here's a from-scratch observable implementation that uses the browser's built-in EventTarget class to manage the subscribers. The Observable's current value is always a `CustomEvent`. If that bothers you then you can use Svelte's tools to create a `derived` store from this one that exposes the `CustomEvent.`'s detail property.

```
class Observable extends EventTarget {
  _value = new CustomEvent("value");
  subscribe(callback) {
    this.addEventListener("value", callback);
    return function unsubscribe() {
      this.removeEventListener("value", callback);
    };
  }
  get value() {
    return this._value;
  }

  set value(newvalue) {
    this._value = newvalue;
    this.dispatchEvent(new CustomEvent("value"), { detail:
        this._value });
  }
}
```

You could also mix this with the EventListener pattern if you want a state object that listens to a number of events in your app, boils them down to a single value, then informs its subscribers of the new value.

6

Of course there's more—so much more

—*I'm more confused now than when I started.*

Observables will do that to you. That's why they are last in the list and why doing nothing, if you can get away with it, is first. Step back. Focus on using the simpler methods. Only resort to the fancier ones when you really need to.

There are a lot more approaches to event handling in the front end that I haven't covered. The pub/sub pattern is similar to observables but with different use cases. I rarely use it. Even observables come in many varieties and implementations. (Some people like to use RXJS for *everything*.) You could even use async iterators if you want. These are just the methods I prefer and am familiar with.

Involved and complex patterns and tactics should alway be your *last* resort. The first rule of coding is that*reading* code is always more complex than *writing* code. Anything that you find hard to understand when writing is going to be opaque as hell when you come back to it later.

Always, *always* start with the simplest possible thing that works and *only* move onto more complex solutions when you have to.

Stick to that and you'll enjoy coding much more.