Log In Sign Up

Understanding Delegated JavaScript Events

OCT 20, 2014

James Holmes Follow | **Thoughts on Thoughts**

While I ended up using a CSS-only implementation for this pen, I started by writing it mostly using classes and JavaScript.

However, I had a conflict. I wanted to use Delegated Events but I also wanted to minimize the dependancies I wanted to inject. I didn't want to have to import *all* of jQuery for this little test, just to be able to use delegated events one bit.

Let's take a closer look at what exactly delegated events are, how they work, and various ways to implement them.

Ok, so what's the issue?

Let's look at a simplified example:

Let's say that I had a list of buttons and each time I clicked on one, then I want to mark that button as "active". If I click it again, then deactivate it.

So let's start with some HTML:

HTML

```
<ul class="toolbar">
  <li><button class="btn">Pencil</button></li>
  <li><button class="btn">Pen</button></li>
  <li><button class="btn">Eraser</button></li>
</ul>
```

I could use standard JavaScript event handler by doing something like this:

JavaScript

```
var buttons = document.querySelectorAll(".toolbar .btn");

for(var i = 0; i < buttons.length; i++) {
  var button = buttons[i];
  button.addEventListener("click", function() {
    if(!button.classList.contains("active"))
      button.classList.add("active");
    else
      button.classList.remove("active");
  });
}
```

And this looks good... but it wont' work... Not the way one might expect it to.

Bitten by closures

For those of you that have been doing functional JavaScript for a while, the problem is pretty obvious.

For the uninitiated, the handler function closes over the `button` variable. However, there is only *one* of them; it gets reassigned by each iteration of the loop.

The first time though the loop, it points a the first button. The next time, the second button. And so on. However, by the time that you click one of the elements, the loop has completed and the `button` variable will point at the *last* element iterated over. Not good.

What we really need is a stable scope for each function; let's refactor an extract a handler generator to give us a stable scope:

JavaScript

```
var buttons = document.querySelectorAll(".toolbar button");
var createToolbarButtonHandler = function(button) {
  return function() {
    if(!button.classList.contains("active"))
      button.classList.add("active");
    else
      button.classList.remove("active");
  };
};

for(var i = 0; i < buttons.length; i++) {
  buttons[i].addEventListener("click", createToolBarButtonHandler(buttons[i]));
}
```

Better! And now it actually works. We are using a function to create us a stable scope for `button`, so the `button` in the handler will always point at the element that we think it will.

So, what the problem?

This seems good and it will work for the most part. However, we can still do better.

First, we are making a lot of handlers. For each element that matches `.toolbar button` we create a function and attach it as an event listener. With the three buttons we have right now the allocations are negligible.

However, what if we had:

JavaScript

```
<ul class="toolbar">
  <li><button id="button_0001">Foo</button></li>
  <li><button id="button_0002">Bar</button></li>
  // ... 997 more elements ...
  <li><button id="button_1000">baz</button></li>
</ul>
```

It won't blow up, but it is far from ideal. We are allocating a bunch of function that we don't *have* to. Let's try to refactor so that we can *share* a single function that is attached *multiple times.*

Rather than closing over the `button` variable to keep track of which button we clicked on, we can use `event` object that is handed to each event handler as the first argument.

The `event` object contains some metadata about the event. In this case, we want the the `currentTarget` property of the event to get a reference to the element that was *actually clicked on*.

JavaScript

```
var buttons = document.querySelectorAll(".toolbar button");

var toolbarButtonHandler = function(e) {
  var button = e.currentTarget;
  if(!button.classList.contains("active"))
    button.classList.add("active");
  else
    button.classList.remove("active");
};

for(var i = 0; i < buttons.length; i++) {
  button.addEventListener("click", toolbarButtonHandler);
}
```

Great! This not only simplified down to a single function that is added multiple times, also made the code more readable by factoring out our generator function.

But, we can *still* do better.

Let's say we added some buttons dynamically into the list. Then we would also need to remember to wire up the event listeners directly to those dynamic elements. And we would have to hold onto a reference to that handler and reference from more places. That doesn't sound like fun.

Perhaps there is a different approach.

Let's start by getting a better understanding of how events work and how they move through the DOM.

Okay, how do (most) events work?

When the user clicks on an element, an event gets generated to notify the application of the user's intent. Events get dispatched in three phases:

Capturing
Target
Bubbling

> NOTE: Not *all* events bubble/capture, instead they are dispatched directly on the target, but most do.

The event starts outside the document and then descends though the DOM hierarchy to the `target` of the event. Once the event reaches it's target, it then turns around and heads back out the same way, until it exits the DOM.

Here is a full HTML example:

HTML

```
<html>
<body>
  <ul>
    <li id="li_1"><button id="button_1">Button A</button></li>
    <li id="li_2"><button id="button_2">Button B</button></li>
    <li id="li_3"><button id="button_3">Button C</button></li>
  </ul>
</body>
</html>
```

If the user clicks on `Button A`, then the event would travel like this like this:

```
START
| #document  \
| HTML        |
| BODY         } CAPTURE PHASE
| UL          |
| LI#li_1    /
| BUTTON     <-- TARGET PHASE
| LI#li_1    \
| UL          |
| BODY         } BUBBLING PHASE
| HTML        |
v #document  /
END
```

Notice that you can follow the path the event takes down to the element that gets clicked on. For any button we click on in our DOM, we can be sure that the event will bubble back out through our parent `ul` element. We can exploit this feature of the event dispatcher, combined with our defined hierarchy to simplify our implementation and implement Delegated Events.

Delegated Events

Delegated events are events that are attached to a parent element, but only get executed when the target of the event matches some criteria.

Let's look at a concrete example and switch back to our toolbar example DOM from before:

HTML

```
<ul class="toolbar">
  <li><button class="btn">Pencil</button></li>
  <li><button class="btn">Pen</button></li>
  <li><button class="btn">Eraser</button></li>
</ul>
```

So, since we know that any clicks on the button elements will get bubbled through the `UL.toolbar` element, let's put the event handler *there* instead. We'll have to adjust our handler a little bit from before;

JavaScript

```
var toolbar = document.querySelector(".toolbar");
toolbar.addEventListener("click", function(e) {
  var button = e.target;
  if(!button.classList.contains("active"))
```

```
      button.classList.add("active");
    else
      button.classList.remove("active");
  });
```

That cleaned up a lot of code, and we have no more loops! Notice that we use `e.target` instead of `e.currentTarget` as we did before. That is because we are listening for the event at a different level.

`e.target` is *actual* target of the event. Where the event is trying to get to, or where it came from, in the DOM.
`e.currentTarget` is the current element that is handling the event.
In our case `e.currentTarget` will be the `UL.toolbar`.

More Robust Delegated Events

Right now, we handle any click on any element that bubbles though `UL.toolbar`, but our matching strategy is a little too simple. What if we had more complicated DOM that included icons and items that were supposed to be non-clickable

HTML

```
<ul class="toolbar">
  <li><button class="btn"><i class="fa fa-pencil"></i> Pencil</button></li>
  <li><button class="btn"><i class="fa fa-paint-brush"></i> Pen</button></li>
  <li class="separator"></li>
  <li><button class="btn"><i class="fa fa-eraser"></i> Eraser</button></li>
</ul>
```

OOPS. Now, when we click on the `LI.separator` or the icons, we add the `active` class to *that* element. That's not cool. We need a way to filter our events so we only react to elements we care about, or if our `target` element is contained by an element we care about.

Let's make a little helper to handle that:

JavaScript

```
var delegate = function(criteria, listener) {
  return function(e) {
    var el = e.target;
    do {
      if (!criteria(el)) continue;
      e.delegateTarget = el;
      listener.apply(this, arguments);
      return;
    } while( (el = el.parentNode) );
    };
  };
```

This helper does two things, first it walks though each element and their parents to see if it matches a criteria function. If it does, then it adds a property to the event object called `delegateTarget`, which is the element that matched our filtering criteria. And then invokes the listener. If nothing matches, the no handlers are fired.

We can use it like this:

JavaScript

```
var toolbar = document.querySelector(".toolbar");
var buttonsFilter = function(elem) { return elem.classList && elem.classList.contains
var buttonHandler = function(e) {
  var button = e.delegateTarget;
  if(!button.classList.contains("active"))
    button.classList.add("active");
  else
    button.classList.remove("active");
};
toolbar.addEventListener("click", delegate(buttonsFilter, buttonHandler));
```

BOOM! That's what I'm talking about: A single event handler, attached to a single element that does all the work, but only does it on the elements that we care about and will react nicely to elements added or removed from the DOM dynamically.

Wrapping up

We've looked at the basics of how to implement event delegation in pure javascript in order to reduce the number of event handlers we need to generate or attach.

There are a few things I would do, if I were going to abstract this into a library, or use it for production level code:

Create helper functions to handle criteria matching in a unified functional way. Something like:

JavaScript

```
var criteria = {
  isElement: function(e) { return e instanceof HTMLElement; },
  hasClass: function(cls) {
    return function(e) {
      return criteria.isElement(e) && e.classList.contains(cls);
    }
  }
```

```
    // More criteria matchers
  };
```

A partial application helper would also be nice:

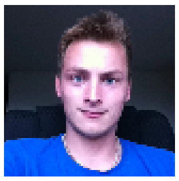
JavaScript

```
  var partialDelgate = function(criteria) {
    return function(handler) {
      return delgate(criteria, handler);
    }
  };
```

If you have any suggestions or improvments, drop a comment or send me a message! Happy coding!

32334 16 70

Comments



Brandon Brule (@brandonbrule) on OCTOBER 21, 2014

Great article to learn from, I didn't know most of that, really cool breakdown.

Quick nitpick - the first example has a syntax error.

for(var i = 0; i < buttons.length, i++)

Should be

for(var i = 0; i < buttons.length; i++) (semi colon instead of comma).

I'm curious why you didn't use 'this' when working with the eventListeners in your first example, is there something I'm missing with that?
http://codepen.io/brandonbrule/pen/635596e33bd79bb4a0cbb23f06c9f5c5

var buttons = document.querySelectorAll(".button");

for (var i = 0, len = buttons.length; i < len; i++){ buttons[i].addEventListener("click", function() {

```
  if( this.className === 'button active' ) {  this.className = 'button not-active
```

});}

Really neat that you can do it without the loop using e.target - That's really cool.

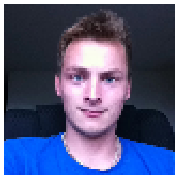
Thanks for the article, great write-up.

James Holmes (@32bitkid) on OCTOBER 21, 2014

Thanks Brandon, I fixed it! I typed this up during a layover between flights, so there might be a few more syntax errors.

As to your second question, I could have used `this` -- which for a event listener, will be the same as `e.currentTarget` -- but then I wouldn't have been able to talk about closures :) In the long run, I've found it better to use the event argument, rather than relying on `this` -- especially in the case of bound functions.

1



Brandon Brule (@brandonbrule) on OCTOBER 21, 2014

Ah gotcha, Thanks James, I appreciate the response and the article. You've converted me.

Any chance you have any examples of the bound functions giving undesired effects? Mostly curious here, I realize after reading your article how much blind faith I've put in those event listeners.

Thanks again James, and great read once again.

James Holmes (@32bitkid) on OCTOBER 22, 2014

@brandonbrule It comes up with CoffeeScript -- namely because the difference between => (fat arrow) is and –> (skinny arrow) is pretty subtle, but the same is applicable for "vanilla" javascript bound function.

Bound functions have a stable this, which means the invoker can't change the `this` context dynamically. Let's look at a CoffeeScript example:

CoffeeScript

```
class HandlerClass  boundHandler: => console.log(this)  unboundHandler: -> cons
```

Now if you called `attachBoundTo()` and clicked, the console would log an instance of HandlerClass, if you called `attachedUnboundTo()` and clicked then the console would log the HTML element you clicked on.

This is the same for bound functions vanilla javascript:

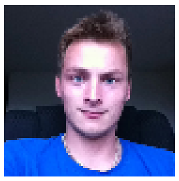
JavaScript

```
var unboundFunction = function() { console.log(this); };var boundFunction = unb
```

Again, clicking on `#foo` you `this` will dynamically be set in the handler to be the element you clicked on. However, clicking on `#bar` -- because its a bound function, its context cannot be changed by the invoker --`this` will always be the string `"baz"`.

Function binding can be a powerful tool to "stabilize" `this`, but it can lead to unexpected results, if you aren't careful (especially true in the case of CoffeeScript)

1



Brandon Brule (@brandonbrule) on OCTOBER 22, 2014

Wow! Would you look at that!

I've updated the pen above to include the bounded function example you showed me.

You just saved me a lot of potential headaches. That's really good to know, and very interesting.

Thanks for this James, I owe you.

1

mark fennell (@markfennell) on OCTOBER 27, 2014

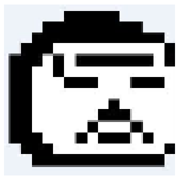
Why does it have to be so complicated when a few lines of javascript wrapped in a function can switch styles on and off? Perhaps there is another scenario in which delegated events would be useful...?

http://codepen.io/markfennell/pen/slblh

James Holmes (@32bitkid) on OCTOBER 27, 2014

@markfennell for a few lines, and you aren't taking into account code maintainability, then yea, something like that would be fine. This solution is for much larger, dynamic -- and difficult to manage -

- solutions.



zhu (@zhulongzheng) on OCTOBER 29, 2014

I think you mean

```
if(!button.classList.contains("active"))    button.classList.add("active");
```

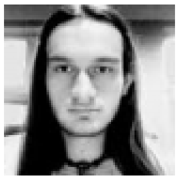
in the handler of the first example js code.

James Holmes (@32bitkid) on OCTOBER 29, 2014

@zhulongzheng yes, i do. thanks!

Justin D'Arcangelo (@justindarc) on OCTOBER 31, 2014

You can do away with the `while` loop by using `Element.prototype.matchesSelector` instead. If you need to support older browsers still, you can easily polyfill it.



Krzysztof Safjanowski (@safjanowski) on NOVEMBER 1, 2014

as `querySelectorAll` returns NodeList collection you should attach listener to first element of that collection`var toolbar = document.querySelectorAll(".toolbar"); toolbar[0].addEventListener("click", function(e) { var button = e.target; if(!button.classList.contains("active")) button.classList.add("active"); else button.classList.remove("active"); });`

What is wrong with `markdown` here?

James Holmes (@32bitkid) on NOVEMBER 1, 2014

@safjanowski thanks. another airport typo; I intended on switching those lines to `document.querySelector()` rather than `document.querySelectorAll()`. Thanks for the heads up!

Surendra (@svsdesigns) on NOVEMBER 4, 2014

Great Article! Yesterday i completed reading Chapter 13: Events from a book "Javascript for web developer", And luckily today i found your article it's really helpful.

In the third example where event object currentTarget property is used, i think you missed the index inside the loop. Currently you are adding attaching event to a variable button which is undefined. Also, i think you can use toggle() instead for checking, adding & removing class from classList.

Thanks :)

Alper Ortac (@alp82) on NOVEMBER 6, 2014

I really enjoyed your article and i agree with your reasoning. Nevertheless there is one thing that bugs me:

When having a single event handler for the toolbar with a delegate function, we have potentially much more events to care about. Even if they are ignored when they do not match the criteria, the event is still fired. The criteria could do something resource intensive or causing a redraw (like getting the width of the element) which would result in worse performance than the initial approach.

What do you think about that matter?

Ben Adams (@benaadams) on MAY 27, 2015

Can use matches to test the criteria e.g. `!el.matches(criteria)`

Its mostly supported but you can shim it like so:

```
if ( !Element.prototype.matches ) {    Element.prototype.matches = Element.prot
```

Thiago Lagden (@lagden) on OCTOBER 17, 2015

Very nice!! I liked the criteria approach!But I prefer individuals handlers...I did a pen to play: http://codepen.io/lagden/pen/meqrQV?editors=001:)

LEAVE A COMMENT MARKDOWN SUPPORTED. DOUBLE-CLICK NAMES TO ADD TO COMMENT.

You must be logged in to comment.

Demo or it didn't happen.