Pocket Guides  Go Make Things

WRITING PLUGINS

with Vanilla JavaScript

CHRIS FERDINANDI

Writing Plugins

By Chris Ferdinandi

Go Make Things, LLC

v2.0.0

Table of Contents

Intro

In this guide, you'll learn:

- How to write modular code.
- How to scope code so that it can be dropped into any project.
- How to expose public functions and APIs in your plugin.
- How to let users pass in their own options and settings.
- How to make your plugins work with module bundlers like WebPack and Browserify.

A quick word about browser compatibility

This guide makes heavy use of ECMAScript 5 (more commonly known as ES5) methods and APIs.

That generally means browser support begins with IE9 and above. Each function or technique mentioned in this guide includes specific browser support information, as some do provide further backwards compatibility.

Let's get started…

Getting Started

The title of this book is a lie.

By definition, plugins extend the functionality of a library or framework. They're dependent on another body of code to work. jQuery plugins add features to jQuery, for example.

Since we're working with vanilla JS, there are no dependencies. What we're going to write aren't plugins. They're components. They're modules. Standalone pieces of code you can drop in and out of any project.

That said, "plugin" is still the commonly used term, so I'll continue to call what we're writing plugins throughout the guide.

Converting a script to a plugin

To help make the concepts in this guide stick better, we'll be converting a simple accordion script into a plugin.

It uses anchor links with the `.accordion-toggle` to show and hide content with the `.accordion` class. The content ID matches the `href` of the anchor links. It uses a CSS class to toggle content visibility.

HTML

```
<a class="accordion-toggle" href="#content-1">Show More 1
</a>

<div class="accordion" id="content-1">
    <p>Some content...</p>
</div>

<a class="accordion-toggle" href="#content-2">Show More 2
</a>

<div class="accordion" id="content-2">
    <p>Some more content...</p>
</div>
```

CSS

```
.accordion {
    display: none;
}

.accordion.active {
    display: block;
}
```

JavaScript

```js document.addEventListener('click', function (event) {

// Only run if the clicked link was an accordion toggle
```

```
if ( !event.target.classList.contains('accordion-toggle')
 ) return;

// Get the target content
var content = document.querySelector(event.target.hash);
if ( !content ) return;

// Prevent default link behavior
event.preventDefault();

// If the content is already expanded, collapse it and qu
it
if ( content.classList.contains('active') ) {
    content.classList.remove('active');
    event.target.classList.remove('active');
    return;
}

// Get all accordion content, loop through it, and close
it
var accordions = document.querySelectorAll('.accordion');
for (var i = 0; i < accordions.length; i++) {
    accordions[i].classList.remove('active');
}

// Get all toggle links, loop through them, and close the
m
var toggles = document.querySelectorAll('.accordion-toggl
e');
for (var n = 0; n < toggles.length; n++) {
```

```
    toggles[n].classList.remove('active');
}

// Open our target content area and toggle link
content.classList.add('active');
event.target.classList.add('active');

}, false); ``
```

Alright. Let’s get started!

Modular Code

When you're first learning JavaScript, it's common to write your scripts as one long chunk of code.

For example, let's take another look at our accordion script. All of the code is one giant function. Some of the code is repetitive, copy/pasted with one or two tweaks.

```
document.addEventListener('click', function (event) {

    // Only run if the clicked link was an accordion togg
le
    if ( !event.target.classList.contains('accordion-togg
le') ) return;

    // Get the target content
    var content = document.querySelector(event.target.has
h);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // If the content is already expanded, collapse it an
d quit
    if ( content.classList.contains('active') ) {
        content.classList.remove('active');
        event.target.classList.remove('active');
        return;
```

```
    }

    // Get all accordion content, loop through it, and cl
ose it
    var accordions = document.querySelectorAll('.accordio
n');
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove('active');
    }

    // Get all toggle links, loop through them, and close
 them
    var toggles = document.querySelectorAll('.accordion-t
oggle');
    for (var n = 0; n < toggles.length; n++) {
        toggles[n].classList.remove('active');
    }

    // Open our target content area and toggle link
    content.classList.add('active');
    event.target.classList.add('active');

}, false);
```

We can make this script easier to debug and maintain, and more DRY (an acronymn for Don't Repeat Yourself), by breaking it up into smaller, more modular parts.

Modularizing the Code

My approach: anything that's more than a line or two of code gets moved into its own function. Let's modularize our accordion script a little.

1. Run our accordion script.

We'll move all of that code into it's own function that we'll call whenever a `click` event happens.

```
// Run our accordion script
var runAccordion = function () {
    // Only run if the clicked link was an accordion togg
le
    if ( !event.target.classList.contains('accordion-togg
le') ) return;

    // Get the target content
    var content = document.querySelector(event.target.has
h);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // If the content is already expanded, collapse it an
d quit
    if ( content.classList.contains('active') ) {
        content.classList.remove('active');
```

```
        event.target.classList.remove('active');
        return;
    }

    // Get all accordion content, loop through it, and cl
ose it
    var accordions = document.querySelectorAll('.accordio
n');
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove('active');
    }

    // Get all toggle links, loop through them, and close
 them
    var toggles = document.querySelectorAll('.accordion-t
oggle');
    for (var n = 0; n < toggles.length; n++) {
        toggles[n].classList.remove('active');
    }

    // Open our target content area and toggle link
    content.classList.add('active');
    event.target.classList.add('active');
};

// Listen for click events
document.addEventListener('click', runAccordion, false);
```

2. Check if it's already expanded.

Next, we'll move the code to check if our content is already expanded into it's own function, `isActive()`.

Currently, if the content is already open, we `return` afterwards to stop our script. That won't work with our code in it's function (we'll only be ending our new modular function, not `runAccordion`).

Instead, if the content *is* expanded, we'll return `true`. When we run `isActive()`, we'll set it as a variable. If it returns `true`, we'll `return` inside `runAccordion()`.

```
// Check if the target content is already active
var isActive = function (content, toggle) {
    if ( content.classList.contains('active') ) {
        content.classList.remove('active');
        toggle.classList.remove('active');
        return true;
    }
};

// Run our accordion script
var runAccordion = function () {
    // Only run if the clicked link was an accordion togg
le
    if ( !event.target.classList.contains('accordion-togg
le') ) return;

    // Get the target content
```

```
    var content = document.querySelector(event.target.has
h);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // If the content is already expanded, collapse it an
d quit
    var expanded = isActive(content, event.target);
    if ( expanded ) return;

    // Get all accordion content, loop through it, and cl
ose it
    var accordions = document.querySelectorAll('.accordio
n');
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove('active');
    }

    // Get all toggle links, loop through them, and close
 them
    var toggles = document.querySelectorAll('.accordion-t
oggle');
    for (var n = 0; n < toggles.length; n++) {
        toggles[n].classList.remove('active');
    }

    // Open our target content area and toggle link
    content.classList.add('active');
```

```
    event.target.classList.add('active');
};

// Listen for click events
document.addEventListener('click', runAccordion, false);
```

3. Close all accordions.

Finally, we'll move our code to close all accordions and accordion toggles into their own function, `closeAccordions()`.

We'll pass in either the appropriate selector as an argument, and let the function do the heavy lifting. This let's us remove some repeated code from our script.

```
// Check if the target content is already active
var isActive = function (content, toggle) {
    if ( content.classList.contains('active') ) {
        content.classList.remove('active');
        toggle.classList.remove('active');
        return true;
    }
};

// Close all accordions or accordion toggles
var closeAccordions = function (selector) {
    var items = document.querySelectorAll(selector);
    for (var i = 0; i < items.length; i++) {
```

```
            items[i].classList.remove('active');
        }
};

// Run our accordion script
var runAccordion = function () {
    // Only run if the clicked link was an accordion togg
le
    if ( !event.target.classList.contains('accordion-togg
le') ) return;

    // Get the target content
    var content = document.querySelector(event.target.has
h);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // If the content is already expanded, collapse it an
d quit
    var expanded = isActive(content, event.target);
    if ( expanded ) return;

    // Close all accordion content and toggles
    closeAccordions('.accordion');
    closeAccordions('.accordion-toggle');

    // Open our target content area and toggle link
    content.classList.add('active');
```

```
    event.target.classList.add('active');
};

// Listen for click events
document.addEventListener('click', runAccordion, false);
```

While our script is actually a few lines longer (mostly due to inline comments and documentation), it's now more readable, and less us quickly scan `runAccordion()` to understand how the script works.

Scoping Our Code

The biggest issue with our new, modular script is that all of our code is in the global scope.

If another script has functions named `isActive()` or `closeAccordions()`, for example, they'll conflict with our functions in unexpected ways.

We want to pull our code out of the global scope. The simplest way to do that is by wrapping it in an IIFE, or Immediately Invoked Function Expression.

```
;(function (window, document, undefined) {

    'use strict';

    // Code goes here...

})(window, document);
```

An IIFE is an unnamed function that runs immediately. By wrapping you code in a function, you keep it out of the global scope.

You'll notice I'm also including `use strict;` in my IIFE. This tells browsers to be less forgiving with bugs and errors, which sounds like a bad thing but helps us write better code.

Here's what our script looks like now.

```
;(function (window, document, undefined) {
```

```
;(function (window, document, undefined) {

    'use strict';

    // Check if the target content is already active
    var isActive = function (content, toggle) {
        if ( content.classList.contains('active') ) {
            content.classList.remove('active');
            toggle.classList.remove('active');
            return true;
        }
    };

    // Close all accordions or accordion toggles
    var closeAccordions = function (selector) {
        var items = document.querySelectorAll(selector);
        for (var i = 0; i < items.length; i++) {
            items[i].classList.remove('active');
        }
    };

    // Run our accordion script
    var runAccordion = function () {
        // Only run if the clicked link was an accordion
toggle
        if ( !event.target.classList.contains('accordion-
toggle') ) return;

        // Get the target content
        var content = document.querySelector(event.target
```

```
.hash);
        if ( !content ) return;

        // Prevent default link behavior
        event.preventDefault();

        // If the content is already expanded, collapse i
t and quit
        var expanded = isActive(content, event.target);
        if ( expanded ) return;

        // Close all accordion content and toggles
        closeAccordions('.accordion');
        closeAccordions('.accordion-toggle');

        // Open our target content area and toggle link
        content.classList.add('active');
        event.target.classList.add('active');
    };

    // Listen for click events
    document.addEventListener('click', runAccordion, fals
e);

})(window, document);
```

As soon as this loads on the page, it will run.

Initializing Your Plugin

You may not always want your code to run as soon as it's loaded. You may want to explicitly initialize by doing something like this:

```
accordion.init();
```

You might also want to be able to run some of the other functions on demand, not just when the script normally runs them. For example, you might want to be able to open an accordion, or close all accordions and toggles, from another script.

```
accordion.closeAccordions();
```

This can make your plugin so much more flexible and future-friendly. On several of my scripts, I often get requests for features that don't exist in the core plugin. Through these public functions, many of the requested features can be bolted on without having to touch the core plugin code at all.

To implement these features, we'll use what's known as a Revealing Module Pattern.

The Revealing Module Pattern

With a revealing module pattern, you assign an IIFE to a named function.

```
var myPlugin = (function () {

    'use strict;'

    // Your code...

})();
```

Inside the IIFE, we'll create an object. Any functions want to use outside of the plugin we'll be assigned as keys in the object, which we'll return at the end of the plugin.

```
var myPlugin = (function () {

    'use strict;'

    // Public APIs
    var publicAPIs = {};

    // Private function
    // This can only be run inside of the accordion() fun
ction
    var someFunction = function () {
        // Do stuff...
    };

    // Public function
    // This can be run from other scripts
    publicAPIs.publicFunction = function () {
        // Do other stuff.
    };

    // Return our public APIs
    return publicAPIs;

})();
```

Converting our Accordion Plugin to a Revealing Module Pattern

Let's convert our accordion plugin to a revealing module pattern. For our purposes, we want to:

1. Initialize our plugin before running it, which we'll do by moving our event listener into a public `init()` function.
2. Make `closeAccordions()` a public API. For this, we'll:
    - Rename `closeAccordions()` to `closeItems()`.
    - Create a new `closeAccordions()` function that calls `closeItems()` with the selectors for our content and toggles.
    - Call `publicAPIs.closeAccordions()` in `runAccordion()`.

```
var accordion = (function () {

    'use strict;'

    // Public APIs
    var publicAPIs = {};

    // Check if the target content is already active
    var isActive = function (content, toggle) {
        if ( content.classList.contains('active') ) {
            content.classList.remove('active');
            toggle.classList.remove('active');
            return true;
        }
    };

    // Close all items with a matching selector
```

```
    var closeItems = function (selector) {
        var items = document.querySelectorAll(selector);
        for (var i = 0; i < items.length; i++) {
            items[i].classList.remove('active');
        }
    };

    // Close all accordions and toggles
    publicAPIs.closeAccordions = function () {
        closeItems('.accordion');
        closeItems('.accordion-toggle');
    }

    // Run our accordion script
    var runAccordion = function () {
        // Only run if the clicked link was an accordion
toggle
        if ( !event.target.classList.contains('accordion-
toggle') ) return;

        // Get the target content
        var content = document.querySelector(event.target
.hash);
        if ( !content ) return;

        // Prevent default link behavior
        event.preventDefault();

        // If the content is already expanded, collapse i
t and quit
```

```
t and quit
        var expanded = isActive(content, event.target);
        if ( expanded ) return;

        // Close all accordion content and toggles
        publicAPIs.closeAccordions();

        // Open our target content area and toggle link
        content.classList.add('active');
        event.target.classList.add('active');
    };

    // Initialize our plugin
    publicAPIs.init = function () {
        // Listen for click events
        document.addEventListener('click', runAccordion,
false);
    };

    // Return our public APIs
    return publicAPIs;

})();
```

One thing you might notice is that we're using our own public methods inside private ones—for example, when we run `publicAPIs.closeAccordions()` inside `runAccordion()`.

You might also notice that we're running a private method —`closeItems()`—inside our public `publicAPIs.closeAccordions()` method.

This approach is extremely flexible, letting you expose only the functions you want to for public use.

To use the our plugin, you would now run `accordion.init()`. You can also close all accordions dynamically from any other script by running `accordion.closeAccordions()`.

Cutting the Mustard

If you're using any functions that are only supported by modern browsers, it's a good idea to check that they're support (an approach known as "cutting the mustard") before initializing your plugin.

In the example below, we're checking for event listener and `querySelector` support.

```
var accordion = (function () {

    'use strict;'

    // ...

    // Initialize our plugin
    publicAPIs.init = function () {
        // Feature test
        var supports = 'querySelector' in document && 'ad
dEventListener' in window;
        if ( !supports ) return;

        // Listen for click events
        document.addEventListener('click', runAccordion,
false);
    };

    // ...

})();
```

Adding an initialization class

One little thing I like to do in my plugins is add an initialization class. This is a class that gets added to the `<html>` element after the plugin has initialized. I can hook into in my CSS to make style changes based on

whether or not a plugin is running.

```
var accordion = (function () {

    'use strict;'

    // ...

    // Initialize our plugin
    publicAPIs.init = function () {
        // Feature test
        var supports = 'querySelector' in document && 'ad
dEventListener' in window;
        if ( !supports ) return;

        // Listen for click events
        document.addEventListener('click', runAccordion,
false);

        // Add our initialization class
        document.documentElement.className += ' .js-accor
dion';
    };

    // ...

})();
```

User Options

To make your plugin for flexible, it's a good idea to let users configure options. Looking at our accordion script, we might want to let users:

- Change the selectors for our accordions and toggles.
- Change the class that get's added and removed to something other than `.active` (to avoid conflicts with other styles).
- Run callback functions before and after our accordions toggle.

Setting up defaults

Someone using your plugin shouldn't have to configure every options. In fact, it should work out-of-the-box without any configuration at all.

The first thing we need to do is setup defaults. We'll create a `defaults` object to hold these values.

```
var accordion = (function () {

    'use strict;'

    // Public APIs
    var publicAPIs = {};

    // Defaults
    var defaults = {
        // Selectors
        selectorToggle: '.accordion-toggle',
        selectorContent: '.accordion',

        // Classes
        toggleClass: 'active',
        contentClass: 'active',
        init: 'js-accordion',

        // Callbacks
        before: function () {},
        after: function () {}
    };

    // ...

})();
```

Passing in options

Next, we want to provide a way for plugin users to pass in their own options that override our defaults. We'll add an `options` argument to our `init()` function.

```
var accordion = (function () {

    'use strict;'

    // Public APIs
    var publicAPIs = {};

    // Defaults
    var defaults = {
        // Selectors
        selectorToggle: '.accordion-toggle',
        selectorContent: '.accordion',

        // Classes
        toggleClass: 'active',
        contentClass: 'active',
        init: 'js-accordion',

        // Callbacks
        before: function () {},
        after: function () {}
    };

    ''
```

```
    // ...

    // Initialize our plugin
    publicAPIs.init = function (options) {
        // Feature test
        var supports = 'querySelector' in document && 'ad
dEventListener' in window;
        if ( !supports ) return;

        // Listen for click events
        document.addEventListener('click', runAccordion,
false);

        // Add our initialization class
        document.documentElement.className += ' .js-accor
dion';
    };

    // Return our public APIs
    return publicAPIs;

})();
```

Merging user options with defaults

Now, we need a way to merge our user's options with the default values. First, we'll create a variable called `settings`. Next, we'll use my `extend()` helper function [1] to merge the user options and defaults and

assign them to the `settings` variable.

```
var accordion = (function () {

    'use strict;'

    // Variables
    var publicAPIs = {}; // Our public APIs
    var settings; // Settings

    // Defaults
    var defaults = {
        // Selectors
        selectorToggle: '.accordion-toggle',
        selectorContent: '.accordion',

        // Classes
        toggleClass: 'active',
        contentClass: 'active',
        init: 'js-accordion',

        // Callbacks
        before: function () {},
        after: function () {}
    };

    /**
     * Merge two or more objects together.
     * @param {Boolean}  deep     If true, do a deep (or
recursive) merge [optional]
```

```
recursive) merge (optional)
     * @param {Object}   objects  The objects to merge to
gether
     * @returns {Object}          Merged values of defaul
ts and options
     */
    var extend = function () {

        // Variables
        var extended = {};
        var deep = false;
        var i = 0;
        var length = arguments.length;

        // Check if a deep merge
        if ( Object.prototype.toString.call( arguments[0]
 ) === '[object Boolean]' ) {
            deep = arguments[0];
            i++;
        }

        // Merge the object into the extended object
        var merge = function ( obj ) {
            for ( var prop in obj ) {
                if ( Object.prototype.hasOwnProperty.call
( obj, prop ) ) {
                    // If deep merge and property is an o
bject, merge properties
                    if ( deep && Object.prototype.toStrin
g.call(obj[prop]) === '[object Object]' ) {
```

```
                        extended[prop] = extend( true, ex
tended[prop], obj[prop] );
                    } else {
                        extended[prop] = obj[prop];
                    }
                }
            }
        };

        // Loop through each object and conduct a merge
        for ( ; i < length; i++ ) {
            var obj = arguments[i];
            merge(obj);
        }

        return extended;

    };

    // ...

    // Initialize our plugin
    publicAPIs.init = function (options) {
        // Feature test
        var supports = 'querySelector' in document && 'ad
dEventListener' in window;
        if ( !supports ) return;

        // Merge user options with the defaults
        settings = extend( defaults, options || {} );
```

```
        // Listen for click events
        document.addEventListener('click', runAccordion,
false);

        // Add our initialization class
        document.documentElement.className += ' .js-accor
dion';
    };

    // Return our public APIs
    return publicAPIs;

})();
```

You may notice that we pass `options` *or* an empty object when merging with the defaults (`options || {}`). If the user doesn't provide any options at all, the `options` value will be `null` and cause an error, so we provide an empty object as a fallback.

Reference our merged settings

Finally, we need to reference our new merged `settings` variable through the script.

Since our selectors may not be a class, we can no longer rely on `classList.contains()` to check if an accordion toggle as clicked. We need to use `matches()` instead. Browser support for this one is a bit

flakey, so we should include a polyfill.[2]

We also now need to pass a class into the `closeItems()` function, since the active class could vary between accordion toggles and content.

```
var accordion = (function () {

    'use strict;'

    // Element.matches() polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches =
            Element.prototype.matchesSelector ||
            Element.prototype.mozMatchesSelector ||
            Element.prototype.msMatchesSelector ||
            Element.prototype.oMatchesSelector ||
            Element.prototype.webkitMatchesSelector ||
            function(s) {
                var matches = (this.document || this.owne
rDocument).querySelectorAll(s),
                    i = matches.length;
                while (--i >= 0 && matches.item(i) !== th
is) {}
                return i > -1;
            };
    }

    // Variables
    var publicAPIs = {}; // Our public APIs
    var settings; // Settings
```

```
    // Defaults
    var defaults = {
        // Selectors
        selectorToggle: '.accordion-toggle',
        selectorContent: '.accordion',

        // Classes
        toggleClass: 'active',
        contentClass: 'active',
        init: 'js-accordion',

        // Callbacks
        before: function () {},
        after: function () {}
    };

    /**
     * Merge two or more objects together.
     * @param {Boolean}  deep     If true, do a deep (or recursive) merge [optional]
     * @param {Object}   objects  The objects to merge together
     * @returns {Object}          Merged values of defaults and options
     */
    var extend = function () {

        // Variables
        var extended = {};
```

```
        var deep = false;
        var i = 0;
        var length = arguments.length;

        // Check if a deep merge
        if ( Object.prototype.toString.call( arguments[0]
 ) === '[object Boolean]' ) {
            deep = arguments[0];
            i++;
        }

        // Merge the object into the extended object
        var merge = function ( obj ) {
            for ( var prop in obj ) {
                if ( Object.prototype.hasOwnProperty.call
( obj, prop ) ) {
                    // If deep merge and property is an o
bject, merge properties
                    if ( deep && Object.prototype.toStrin
g.call(obj[prop]) === '[object Object]' ) {
                        extended[prop] = extend( true, ex
tended[prop], obj[prop] );
                    } else {
                        extended[prop] = obj[prop];
                    }
                }
            }
        };
```

```
        // Loop through each object and conduct a merge
        for ( ; i < length; i++ ) {
            var obj = arguments[i];
            merge(obj);
        }

        return extended;

    };

    // Check if the target content is already active
    var isActive = function (content, toggle) {
        if ( content.classList.contains(settings.contentC
lass) ) {
            settings.before(content, toggle);
            content.classList.remove(settings.contentClas
s);
            toggle.classList.remove(settings.toggleClass)
;
            settings.after(content, toggle);
            return true;
        }
    };

    // Close all items with a matching selector
    var closeItems = function (selector, activeClass) {
        var items = document.querySelectorAll(selector);
        for (var i = 0; i < items.length; i++) {
            items[i].classList.remove(activeClass);
        }
```

```
    };

    // Close all accordions and toggles
    publicAPIs.closeAccordions = function () {
        closeItems(settings.selectorContent, settings.con
tentClass);
        closeItems(settings.selectorToggle, settings.togg
leClass);
    }

    // Run our accordion script
    var runAccordion = function () {
        // Only run if the clicked link was an accordion
toggle
        if ( !event.target.matches(settings.selectorToggl
e) ) return;

        // Get the target content
        var content = document.querySelector(event.target
.hash);
        if ( !content ) return;

        // Prevent default link behavior
        event.preventDefault();

        // If the content is already expanded, collapse i
t and quit
        var expanded = isActive(content, event.target);
        if ( expanded ) return;
```

```
            // Run our callback before
            settings.before(content, event.target);

            // Close all accordion content and toggles
            publicAPIs.closeAccordions();

            // Open our target content area and toggle link
            content.classList.add(settings.contentClass);
            event.target.classList.add(settings.toggleClass);

            // Run our callback after
            settings.after(content, event.target);
        };

        // Initialize our plugin
        publicAPIs.init = function (options) {
            // Feature test
            var supports = 'querySelector' in document && 'ad
dEventListener' in window;
            if ( !supports ) return;

            // Merge user options with the defaults
            settings = extend( defaults, options || {} );

            // Listen for click events
            document.addEventListener('click', runAccordion,
false);

            // Add our initialization class
```

```
        document.documentElement.className += ' ' + setti
ngs.init;
    };

    // Return our public APIs
    return publicAPIs;

})();
```

Initializing the plugin with options

Now we're ready to initialize our plugin with user options. Here's an example.

```
accordion.init({
    toggleSelector: '[data-accordion-toggle]',
    contentClass: 'is-open',
    after: function (content, toggle) {
        console.log(content);
        console.log(toggle);
    }
});
```

Destroying the Plugin Initialization

Sometimes, it's helpful to provide a way for users to destroy your plugin after it's been initialized.

This becomes a public function users can run that resets the `settings` variable, removes any event listeners, and restores any changes you've made to their original state.

```
var accordion = (function () {

    'use strict;'

    // ...

    publicAPIs.destroy = function () {
        // Only run if settings is set
        if ( !settings ) return;

        // Remove event listener
        document.removeEventListener('click', runAccordio
n, false);

        // Remove the initialization class
        document.documentElement.classList.remove(setting
s.init);

        // Reset settings
        settings = null;
    };

    // ...

})();
```

For good measure, you should also call it whenever you run your `publicAPIs.init()` function.

```
var accordion = (function () {

    'use strict;'

    // ...

    publicAPIs.destroy = function () {
        // Only run if settings is set
        if ( !settings ) return;

        // Remove event listener
        document.removeEventListener('click', runAccordio
n, false);

        // Remove the initialization class
        document.documentElement.classList.remove(setting
s.init);

        // Reset settings
        settings = null;
    };

    // Initialize our plugin
    publicAPIs.init = function (options) {
        // Feature test
        var supports = 'querySelector' in document && 'ad
dEventListener' in window;
        if ( !supports ) return;

        // Destroy any previous initializations
```

```
        // Destroy any previous initializations
        publicAPIs.destroy();

        // Merge user options with the defaults
        settings = extend( defaults, options || {} );

        // Listen for click events
        document.addEventListener('click', runAccordion,
false);

        // Add our initialization class
        document.documentElement.className += ' ' + setti
ngs.init;
    };

    // ...

})();
```

Universal Module Definition (UMD)

If you want your plugin to work with RequireJS, Node, WebPack, Browserify, and other module bundlers, you should use a Universal Module Definition (UMD) pattern.

UMD merges two differing approaches to modules—AMD and CommonJS—with the global variable technique you use for revealing module patterns.

Replace `myPlugin` with the name of the global variable you want to use. In UMD, `window` becomes `root`. It's helpful to set `window` as a variable to avoid messing that up.

```
(function (root, factory) {
    if ( typeof define === 'function' && define.amd ) {
        define([], factory(root));
    } else if ( typeof exports === 'object' ) {
        module.exports = factory(root);
    } else {
        root.myPlugin = factory(root);
    }
})(typeof global !== 'undefined' ? global : this.window ||
| this.global, function (root) {

    'use strict';

    // Redefine window
    var window = root;

    // Your code...

});
```

The accordion plugin as UMD

```
(function (root, factory) {
    if ( typeof define === 'function' && define.amd ) {
        define([], factory(root));
    } else if ( typeof exports === 'object' ) {
        module.exports = factory(root);
```

```
    } else {
        root.accordion = factory(root);
    }
})(typeof global !== 'undefined' ? global : this.window ||
| this.global, function (root) {

    'use strict';

    // Element.matches() polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches =
            Element.prototype.matchesSelector ||
            Element.prototype.mozMatchesSelector ||
            Element.prototype.msMatchesSelector ||
            Element.prototype.oMatchesSelector ||
            Element.prototype.webkitMatchesSelector ||
            function(s) {
                var matches = (this.document || this.owne
rDocument).querySelectorAll(s),
                    i = matches.length;
                while (--i >= 0 && matches.item(i) !== th
is) {}
                return i > -1;
            };
    }

    // Variables
    var window = root; // Redefine window
    var publicAPIs = {}; // Our public APIs
```

```
    var settings; // Settings

    // Defaults
    var defaults = {
        // Selectors
        selectorToggle: '.accordion-toggle',
        selectorContent: '.accordion',

        // Classes
        toggleClass: 'active',
        contentClass: 'active',
        init: 'js-accordion',

        // Callbacks
        before: function () {},
        after: function () {}
    };

    /**
     * Merge two or more objects together.
     * @param {Boolean}  deep     If true, do a deep (or recursive) merge [optional]
     * @param {Object}   objects  The objects to merge together
     * @returns {Object}          Merged values of defaults and options
     */
    var extend = function () {

        // Variables
```

```
        var extended = {};
        var deep = false;
        var i = 0;
        var length = arguments.length;

        // Check if a deep merge
        if ( Object.prototype.toString.call( arguments[0]
 ) === '[object Boolean]' ) {
            deep = arguments[0];
            i++;
        }

        // Merge the object into the extended object
        var merge = function ( obj ) {
            for ( var prop in obj ) {
                if ( Object.prototype.hasOwnProperty.call
( obj, prop ) ) {
                    // If deep merge and property is an o
bject, merge properties
                    if ( deep && Object.prototype.toStrin
g.call(obj[prop]) === '[object Object]' ) {
                        extended[prop] = extend( true, ex
tended[prop], obj[prop] );
                    } else {
                        extended[prop] = obj[prop];
                    }
                }
            }
        };
```

```
        // Loop through each object and conduct a merge
        for ( ; i < length; i++ ) {
            var obj = arguments[i];
            merge(obj);
        }

        return extended;

    };

    // Check if the target content is already active
    var isActive = function (content, toggle) {
        if ( content.classList.contains(settings.contentC
lass) ) {
            settings.before(content, toggle);
            content.classList.remove(settings.contentClas
s);
            toggle.classList.remove(settings.toggleClass)
;
            settings.after(content, toggle);
            return true;
        }
    };

    // Close all items with a matching selector
    var closeItems = function (selector, activeClass) {
        var items = document.querySelectorAll(selector);
        for (var i = 0; i < items.length; i++) {
            items[i].classList.remove(activeClass);
        }
```

```
        }
    };

    // Close all accordions and toggles
    publicAPIs.closeAccordions = function () {
        closeItems(settings.selectorContent, settings.con
tentClass);
        closeItems(settings.selectorToggle, settings.togg
leClass);
    }

    // Run our accordion script
    var runAccordion = function () {
        // Only run if the clicked link was an accordion
toggle
        if ( !event.target.matches(settings.selectorToggl
e) ) return;

        // Get the target content
        var content = document.querySelector(event.target
.hash);
        if ( !content ) return;

        // Prevent default link behavior
        event.preventDefault();

        // If the content is already expanded, collapse i
t and quit
        var expanded = isActive(content, event.target);
        if ( expanded ) return;
```

```
            // Run our callback before
            settings.before(content, event.target);

            // Close all accordion content and toggles
            publicAPIs.closeAccordions();

            // Open our target content area and toggle link
            content.classList.add(settings.contentClass);
            event.target.classList.add(settings.toggleClass);

            // Run our callback after
            settings.after(content, event.target);
        };

        // Initialize our plugin
        publicAPIs.init = function (options) {
            // Feature test
            var supports = 'querySelector' in document && 'ad
dEventListener' in window;
            if ( !supports ) return;

            // Merge user options with the defaults
            settings = extend( defaults, options || {} );

            // Listen for click events
            document.addEventListener('click', runAccordion,
false);

            // Add our initialization class
```

```
        document.documentElement.className += ' ' + setti
ngs.init;
    };

    // Return our public APIs
    return publicAPIs;

});
```

And then you can still just call `accordion.init()` to initialize it.

Putting it all together

To make this all tangible, let's work on a project together. We'll take a simple accordion script and convert it into a plugin.

The starter template and complete project code are included in the source code[3] on GitHub.

Getting Setup

Our script includes a small amount of CSS to show and hide content using `display: none` and `display: block`. It also includes an event listener and a few modern JavaScript methods.

CSS

```
.accordion-content {
    display: none;
}

.accordion-content.active {
    display: block;
}
```

JavaScript

```
// Listen for click events
```

```
// Listen for click events
document.addEventListener('click', function (event) {

    // Only run if the clicked link was an accordion toggle
    if ( !event.target.classList.contains('accordion-toggle') ) return;

    // Get the target content
    var content = document.querySelector(event.target.hash);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // If the content is already expanded, collapse it and quit
    if ( content.classList.contains('active') ) {
        content.classList.remove('active');
        return;
    }

    // Get all accordion content, loop through it, and close it
    var accordions = document.querySelectorAll('.accordion-content.active');
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove('active');
    }
```

```
    // Open our target content area
    content.classList.add('active');

}, false);
```

Markup

```
<a class="accordion-toggle" href="#content">Show Content<
/a>

<div class="accordion-content" id="content">
    The content
</div>
```

Planning

Let's take a quick inventory of what we'd like to accomplish.

1. Scope our code inside a function wrapper.
2. Add an initialization function.
3. Modularize our script.
4. Add a destroy function to under our initialization.
5. Let users pass in options to configure things.
6. Add developer hooks.

Scoping our code

The first thing we want to do is add a functional wrapper around our code to keep it out of the global scope.

To maximize compatibility with module loaders, let's use a UMD wrapper. We'll paste our current code in, and change `accordion` from the boilerplate to `accordion`.

```
(function (root, factory) {
    if ( typeof define === 'function' && define.amd ) {
        define([], factory(root));
    } else if ( typeof exports === 'object' ) {
        module.exports = factory(root);
    } else {
        root.accordion = factory(root);
    }
})(typeof global !== 'undefined' ? global : this.window |
| this.global, function (root) {

    'use strict';

    // Redefine window
    var window = root;

    // Listen for clicks on the document
    document.addEventListener('click', function (event) {

        // Bail if our clicked element doesn't have the .
accordion-toggle class
```

```
        if ( !event.target.classList.contains('accordion-
toggle') ) return;

        // Get the target content
        var content = document.querySelector(event.target
.hash);
        if ( !content ) return;

        // Prevent default link behavior
        event.preventDefault();

        // If the content is already expanded, collapse i
t and quit
        if ( content.classList.contains('active') ) {
            content.classList.remove('active');
            return;
        }

        // Get all accordion content, loop through it, an
d close it
        var accordions = document.querySelectorAll('.acco
rdion-content.active');
        for (var i = 0; i < accordions.length; i++) {
            accordions[i].classList.remove('active');
        }

        // Open our target content area
        content.classList.add('active');

    }, false);
```

```
    }, false);

});
```

Add an initialization method

Now, let's add an initialization function, so that it only runs if we explicitly call it.

Let's add a placeholder object for our public methods, move our event listener to an `.init()` method, and return our public methods object.

```
(function (root, factory) {
    if ( typeof define === 'function' && define.amd ) {
        define([], factory(root));
    } else if ( typeof exports === 'object' ) {
        module.exports = factory(root);
    } else {
        root.accordion = factory(root);
    }
})(typeof global !== 'undefined' ? global : this.window ||
| this.global, function (root) {

    'use strict';


    //
    // Variables
    //
```

```
    //

    var window = root; // Redefine window
    var publicMethods = {}; // Placeholder for public met
hods


    //
    // Methods
    //

    /**
     * Initialize our script
     */
    publicMethods.init = function () {

        // Listen for clicks on the document
        document.addEventListener('click', function (even
t) {

            // Bail if our clicked element doesn't have t
he .accordion-toggle class
            if ( !event.target.classList.contains('accord
ion-toggle') ) return;

            // Get the target content
            var content = document.querySelector(event.ta
rget.hash);
            if ( !content ) return;
```

```
            // Prevent default link behavior
            event.preventDefault();

            // If the content is already expanded, collap
se it and quit
            if ( content.classList.contains('active') ) {
                content.classList.remove('active');
                return;
            }

            // Get all accordion content, loop through it
, and close it
            var accordions = document.querySelectorAll('.
accordion-content.active');
            for (var i = 0; i < accordions.length; i++) {
                accordions[i].classList.remove('active');
            }

            // Open our target content area
            content.classList.add('active');

        }, false);

    };


    //
    // Return our public methods
    //
```

```
    return publicMethods;

});

// Initialize our script
accordion.init();
```

Let's also add an initialization class, and make the CSS that hides our content dependant on it's presence.

JavaScript

```
/**
 * Initialize our script
 */
publicMethods.init = function () {

    // Listen for clicks on the document
    document.addEventListener('click', function (event) {

        // Bail if our clicked element doesn't have the .accordion-toggle class
        if ( !event.target.classList.contains('accordion-toggle') ) return;

        // Get the target content
        var content = document.querySelector(event.target.hash);
        if ( !content ) return;
```

```
        // Prevent default link behavior
        event.preventDefault();

        // If the content is already expanded, collapse i
t and quit
        if ( content.classList.contains('active') ) {
            content.classList.remove('active');
            return;
        }

        // Get all accordion content, loop through it, an
d close it
        var accordions = document.querySelectorAll('.acco
rdion-content.active');
        for (var i = 0; i < accordions.length; i++) {
            accordions[i].classList.remove('active');
        }

        // Open our target content area
        content.classList.add('active');

        // Add an initialization class
        document.documentElement.className += ' js-accord
ion';

    }, false);

};
```

CSS

```
.js-accordion .accordion-content {
    display: none;
}

.accordion-content.active {
    display: block;
}
```

Modularize our script

Next, let's break our script up into some smaller parts.

First, let's pull everything in our event listener our into a standalone function. This will make it possible for us to remove the event listener later. We'll pass in the `event` as an argument into our function.

```
//
// Methods
//

/**
 * Function to run on click
 * @param  {Event} event The click event
 */
```

```
 */
var clickHandler = function (event) {

    // Bail if our clicked element doesn't have the .acco
rdion-toggle class
    if ( !event.target.classList.contains('accordion-togg
le') ) return;

    // Get the target content
    var content = document.querySelector(event.target.has
h);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // If the content is already expanded, collapse it an
d quit
    if ( content.classList.contains('active') ) {
        content.classList.remove('active');
        return;
    }

    // Get all accordion content, loop through it, and cl
ose it
    var accordions = document.querySelectorAll('.accordio
n-content.active');
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove('active');
    }
```

```
    // Open our target content area
    content.classList.add('active');
};

/**
 * Initialize our script
 */
publicMethods.init = function () {

    // Listen for clicks on the document
    document.addEventListener('click', clickHandler, fals
e);

    // Add an initialization class
    document.documentElement.className += ' js-accordion'
;

};
```

Then, let's move all of our code that actually toggles visibility into it's own function that we'll call within our `clickHandler()`. We'll need to pass in the `content` as an argument.

```
//
// Methods
//

/**
```

```
 * Toggle accordion content visibility
 * @param  {Node} content  The content to show or hide
 */
var toggleAccordion = function (content) {

    // If the content is already expanded, collapse it an
d quit
    if ( content.classList.contains('active') ) {
        content.classList.remove('active');
        return;
    }

    // Get all accordion content, loop through it, and cl
ose it
    var accordions = document.querySelectorAll('.accordio
n-content.active');
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove('active');
    }

    // Open our target content area
    content.classList.add('active');

};

/**
 * Function to run on click
 * @param  {Event} event The click event
 */
var clickHandler = function (event) {
```

```
    // Bail if our clicked element doesn't have the .acco
rdion-toggle class
    if ( !event.target.classList.contains('accordion-togg
le') ) return;

    // Get the target content
    var content = document.querySelector(event.target.has
h);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // Toggle accordion content visibility
    toggleAccordion(content);
};

/**
 * Initialize our script
 */
publicMethods.init = function () {

    // Listen for clicks on the document
    document.addEventListener('click', clickHandler, fals
e);

    // Add an initialization class
    document.documentElement.className += ' js-accordion'
;
```

```
,

};
```

Add a destroy function

Now we can add a function to destroy our initialization. This is useful when we want to cancel our accordion or reinitialize it with some new options.

This will be a public method, and we'll call it in our initialization method.

```
/**
 * Destroy our script
 */
publicMethods.destroy = function () {

    // Remove our event listener
    document.removeEventListener('click', clickHandler, f
alse);

    // Remove our initialization class
    document.documentElement.classList.remove('js-accordi
on');

};

/**
```

```
 * Initialize our script
 */
publicMethods.init = function () {

    // Destroy any existing initializations
    publicMethods.destroy();

    // Listen for clicks on the document
    document.addEventListener('click', clickHandler, fals
e);

    // Add an initialization class
    document.documentElement.className += ' js-accordion'
;

};
```

If you open up the console tab in developer tools and run `accordion.destroy()`, all of the hidden content should become visible.

Add user options

Next, let's add some options that users can configure themselves—things like selectors and initialization classes.

To get started, let's set up our defaults as an object with key/value pairs, and add a `null` variable to hold our settings globally within our plugin.

```
//
// Variables
//

var window = root; // Redefine window
var publicMethods = {}; // Placeholder for public methods
var settings; // Settings placeholder

// Defaults
var defaults = {
    selectorToggle: '.accordion-toggle',
    selectorContent: '.accordion-content',
    activeClass: 'active',
    initClass: 'js-accordion'
};
```

Now, let's add a function to merge user settings into our defaults.

```
//
// Methods
//

/**
 * Merge two or more objects together.
 * @param {Boolean}  deep     If true, do a deep (or recu
rsive) merge [optional]
 * @param {Object}   objects  The objects to merge togeth
er
 * @returns {Object}          Merged values of defaults a
```

```
nd options
 */
var extend = function () {

    // Variables
    var extended = {};
    var deep = false;
    var i = 0;
    var length = arguments.length;

    // Check if a deep merge
    if ( Object.prototype.toString.call( arguments[0] ) =
== '[object Boolean]' ) {
        deep = arguments[0];
        i++;
    }

    // Merge the object into the extended object
    var merge = function ( obj ) {
        for ( var prop in obj ) {
            if ( Object.prototype.hasOwnProperty.call( ob
j, prop ) ) {
                // If deep merge and property is an objec
t, merge properties
                if ( deep && Object.prototype.toString.ca
ll(obj[prop]) === '[object Object]' ) {
                    extended[prop] = extend( true, extend
ed[prop], obj[prop] );
                } else {
                    extended[prop] = obj[prop];
```

```
                    extended[prop] = obj[prop];
                }
            }
        }
    };

    // Loop through each object and conduct a merge
    for ( ; i < length; i++ ) {
        var obj = arguments[i];
        merge(obj);
    }

    return extended;

};
```

In our `publicMethods.init()` function, we'll add an argument for options. We'll also merge `options` into `defaults` and set them to the `settings` variable. We'll omit the `var` before `settings` to modify our global variable instead of creating a new one.

When merging, we'll pass our options in as `options || {}` so that if the user doesn't provide any options, we'll fall back to an empty object. This prevents our `extend` method from throwing an error.

```
/**
 * Initialize our script
 */
publicMethods.init = function (options) {

    // Destroy any existing initializations
    publicMethods.destroy();

    // Merge options into defaults
    settings = extend(defaults, options || {});

    // Listen for clicks on the document
    document.addEventListener('click', clickHandler, fals
e);

    // Add an initialization class
    document.documentElement.className += ' js-accordion'
;

};
```

We also want to reset the `settings` variable in our `publicMethods.destroy()` function. We can use that variable as a test to determine if the script is already initialized or not, and bail if this is the first initialization.

```
/**
 * Destroy our script
 */
publicMethods.destroy = function () {

    // Check if the script is initialized
    if (!settings) return;

    // Remove our event listener
    document.removeEventListener('click', clickHandler, f
alse);

    // Remove our initialization class
    document.documentElement.classList.remove('js-accordi
on');

    // Reset our settings
    settings = null;

};
```

Using our settings in the script

Finally, we need to use our merged settings in our script instead of our hand-coded values.

```
/**
 * Toggle accordion content visibility
```

```
 * @param  {Node} content  The content to show or hide
 */
var toggleAccordion = function (content) {

    // If the content is already expanded, collapse it an
d quit
    if ( content.classList.contains(settings.activeClass)
 ) {
        content.classList.remove(settings.activeClass);
        return;
    }

    // Get all accordion content, loop through it, and cl
ose it
    var accordions = document.querySelectorAll(settings.s
electorContent + '.' + settings.activeClass);
    for (var i = 0; i < accordions.length; i++) {
        accordions[i].classList.remove(settings.activeCla
ss);
    }

    // Open our target content area
    content.classList.add(settings.activeClass);

};

/**
 * Function to run on click
 * @param  {Event} event The click event
```

```
 */
var clickHandler = function (event) {

    // Bail if our clicked element doesn't have the .acco
rdion-toggle class
    if ( !event.target.classList.contains(settings.select
orToggle) ) return;

    // Get the target content
    var content = document.querySelector(event.target.has
h);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // Toggle accordion content visibility
    toggleAccordion(content);
};

/**
 * Destroy our script
 */
publicMethods.destroy = function () {

    // Check if the script is initialized
    if (!settings) return;

    // Remove our event listener
    document.removeEventListener('click', clickHandler, f
```

```
alse);

    // Remove our initialization class
    document.documentElement.classList.remove(settings.in
itClass);

    // Reset our settings
    settings = null;

};

/**
 * Initialize our script
 */
publicMethods.init = function (options) {

    // Destroy any existing initializations
    publicMethods.destroy();

    // Merge options into defaults
    settings = extend(defaults, options || {});

    // Listen for clicks on the document
    document.addEventListener('click', clickHandler, fals
e);

    // Add an initialization class
    document.documentElement.className += ' ' + settings.
initClass;
```

```
};
```

Beyond class-based selectors

The one snag here is with our `clickHandler()` function. Right now, it relies on `classList.contains()` to check if the clicked element is one of our accordion toggles.

Our selector is `.accordion-toggle`. That leading `.` will cause our `if` statement to fail, as `classList.contains()` uses just the class name, not the CSS selector associated with classes. It also doesn't account for other types of selectors, like data attributes (`[data-accordion-toggle]`) or selectors with multiple classes (`.main .accordion-toggle`);

Instead, we want to use `.matches()`, which will return true whenever the selector(s) would match on the element in question.

```
/**
 * Function to run on click
 * @param  {Event} event The click event
 */
var clickHandler = function (event) {

    // Bail if our clicked element doesn't have the .acco
rdion-toggle class
    if ( !event.target.matches(settings.selectorToggle) )
 return;

    // ...

};
```

The `.matches()` method has spotty browser support, and in some supporting browsers requires a vendor prefix (like `webkitMatches()`). Fortunately, a simple polyfill extends this to all browsers that support `querySelector`.

Let's add this globally within our script.

```
if (!Element.prototype.matches) {
    Element.prototype.matches =
        Element.prototype.matchesSelector ||
        Element.prototype.mozMatchesSelector ||
        Element.prototype.msMatchesSelector ||
        Element.prototype.oMatchesSelector ||
        Element.prototype.webkitMatchesSelector ||
        function(s) {
            var matches = (this.document || this.ownerDoc
ument).querySelectorAll(s),
                i = matches.length;
            while (--i >= 0 && matches.item(i) !== this)
{}
            return i > -1;
        };
}
```

Adding developer hooks

To make our plugin as flexible as possible, we also want to add hooks developers can use to extend functionality.

First, let's make the `toggleAccordion()` method public, so that users can run it from other scripts. For example, imagine if a user wanted to dynamically open a specific piece of accordion content if a link in the navigation is clicked.

Don't forget to change our function call in the `clickHandler()`

method.

```
/**
 * Toggle accordion content visibility
 * @param  {Node} content  The content to show or hide
 */
publicMethods.toggleAccordion = function (content) {
    // ...
};

/**
 * Function to run on click
 * @param  {Event} event The click event
 */
var clickHandler = function (event) {

    // ...

    // Toggle accordion content visibility
    publicMethods.toggleAccordion(content);

};
```

Next, let's add callback functions that can be run before and after accordion content is toggled. These let developers add additional functionality to our plugin without having to touch the core code.

We'll include these as empty functions in our `defaults` variable, and add them to our `clickHandler` function. Developers can pass their callbacks in with the `options` object.

We'll also pass in the toggle element and content area as arguments, for maximum flexibility.

```
// Defaults
var defaults = {

    // Selectors
    selectorToggle: '.accordion-toggle',
    selectorContent: '.accordion-content',
    activeClass: 'active',
    initClass: 'js-accordion',

    // Callbacks
    callbackBefore: function () {},
    callbackAfter: function () {}

};

// ...

/**
 * Function to run on click
 * @param  {Event} event The click event
 */
var clickHandler = function (event) {
```

```
    // Bail if our clicked element doesn't have the .acco
rdion-toggle class
    if ( !event.target.matches(settings.selectorToggle) )
 return;

    // Get the target content
    var content = document.querySelector(event.target.has
h);
    if ( !content ) return;

    // Prevent default link behavior
    event.preventDefault();

    // Run callback before toggling the accordion
    settings.callbackBefore(event.target, content);

    // Toggle accordion content visibility
    publicMethods.toggleAccordion(content);

    // Run callback after toggling the accordion
    settings.callbackAfter(event.target, content);

};
```

And with those last two changes, you've made a plugin that's incredibly flexible and developer-friendly. Congrats!

About the Author

Hi, I'm Chris Ferdinandi. I help people learn JavaScript.

I love pirates, puppies, and Pixar movies, and live near horse farms in rural Massachusetts. I run Go Make Things with Bailey Puppy, a lab-mix from Tennessee.

You can find me:

- On my website at GoMakeThings.com.
- By email at chris@gomakethings.com.
- On Twitter at @ChrisFerdinandi.

1. https://github.com/cferdinandi/extend↩
2. https://developer.mozilla.org/en-US/docs/Web/API/Element/matches#Polyfill↩
3. https://github.com/cferdinandi/writing-plugins-source-code/↩