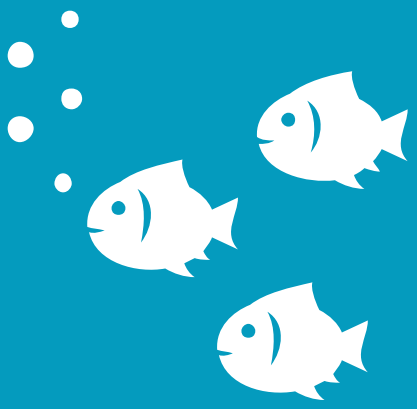
Pocket Guides  Go Make Things

9

WRITING PLUGINS

with Vanilla JavaScript



CHRIS FERDINANDI

Writing Plugins

By Chris Ferdinandi

Go Make Things, LLC

v2.1.0

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) and ECMA 6 (ES6) methods and APIs.

My goal for browser support is IE9 and above. Each function or technique mentioned in this guide includes specific browser support information. For methods and APIs that don't meet that standard, I also include information about polyfills—snippets of code that add support for features to browsers that don't natively offer it.

You'll never have to run a command line prompt, compile code, or learn a weird pseudo language (though you certain can if you want to).

Note: *You can extend support all the way back to IE7 with a polyfill service like polyfill.io*[1].

Using the code in this guide

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

```javascript 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-toggl
e')) 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 two or three lines 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-toggl
e')) 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-toggl
e')) 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 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-toggl
e')) 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 lets 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-t
oggle')) 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 will 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-t
oggle')) 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()`.

Feature Tests

If you're using any functions that are only supported by modern browsers, it's a good idea to check that they're supported before initializing your plugin.

In the example below, we're checking for `addEventListener` 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-accord
ion';

    };

    // ...

})();
```

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).
- Change the class that get's added to the `document` element on initialization.

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'
    };

    // ...

})();
```

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'
    };

    // ...

    // 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-accord
ion';

    };

    // Return our public APIs
    return publicAPIs;

})();
```

Merging user options with defaults

Next, we need a way to merge our user's options with the default values.

First, we'll create a variable called `settings`. Then, we'll use `extend()`[2], a helper method I wrote, to merge the user options and defaults together 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'
    };

    // Merge two or more objects together
    extend = function () {

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

        // 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 (obj.hasOwnProperty(prop)) {
                    // If property is an object, merge pr
operties
                    if (deep && Object.prototype.toString
.call(obj[prop]) === '[object Object]') {
                        extended[prop] = extend(extended[
prop], obj[prop]);
                    } else {
                        extended[prop] = obj[prop];
                    }
                }
            }
        };

        // Loop through each object and conduct a merge
        for (; i < arguments.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-accord
ion';

    };

    // Return our public APIs
    return publicAPIs;

})();
```

You may notice that we pass in `options` *or* an empty object when merging with the defaults.

```
extend(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 throughout 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 the `matches()` method instead. While browser support for this goes back to IE9, older browsers used a vendor-prefixed version, so we need to include a polyfill[3] to standardize behavior across browsers.

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 (simple version)
    // https://developer.mozilla.org/en-US/docs/Web/API/Element/matches#Polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches = Element.prototype.msMatchesSelector || Element.prototype.webkitMatchesSelector
```

```
atchesSelector || Element.prototype.webkitMatchesSelector
;
    }

    // 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'
    };

    // Merge two or more objects together
    extend = function () {

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

        // 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 (obj.hasOwnProperty(prop)) {
                    // If property is an object, merge pr
operties
                    if (deep && Object.prototype.toString
.call(obj[prop]) === '[object Object]') {
                        extended[prop] = extend(extended[
prop], obj[prop]);
                    } else {
                        extended[prop] = obj[prop];
                    }
                }
            }
        };

        // Loop through each object and conduct a merge
        for (; i < arguments.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.contentCl
ass)) {
            content.classList.remove(settings.contentClas
s);
            toggle.classList.remove(settings.toggleClass)
;
            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.selectorToggle
)) 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
        publicAPIs.closeAccordions();

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

    // 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',
    init: 'accordion-init'
});
```

Events and Callbacks

You can provide hooks developers can use to run code when key specific things happen in your plugin.

For example, when an accordion is opened, a developer may want to make an Ajax call to get content from another page and add it to the accordion. Or they may want to stop a video from playing when the accordion closes.

There are two ways to provide these hooks:

1. Callbacks
2. Events

They both achieve the same goal, but work a little differently.

Callbacks

A callback is a piece of code that runs at a specific time. In your plugin, you can let users pass callbacks in as an option.

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

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

    // Calbacks
    callbackOpen: function () {},
    callbackClose: function () {}
};
```

Then, you can run it at the appropriate time, and even pass in arguments that developers can use to in their callback.

```
// Run our accordion script
var runAccordion = function () {

    // ...

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

    // Run our callback
    // content = the accordion
    // event.target = the toggle
    settings.callbackOpen(content, event.target);

};
```

A developer might pass in a callback like this.

```
accordion.init({
    // Autoplay an HTML video when the accordion opens
    callbackOpen: function (content) {
        var video = content.querySelector('video');
        if (video) {
            video.play();
        }
    }
});
```

Events

Instead of passing in a callback, you can also emit a custom event that developers can listen for with `addEventListener()`.

The CustomEvent API provides a way to create custom events and attach data about the event. You can attach the event to the `window` for a global event (for example, after initializing the script). You can also attach the event to specific element (for example, the accordion being opened or closed).

Creating a custom event

Create a `new CustomEvent`, and pass in the name of the event. You can optionally pass in an object with options and details.

There are two standard options on events that you're likely to change. Both are booleans with a default of `false`.

- If `bubbles` is `true`, an event will "bubble up" or propagate through all of the element's parent elements[4].
- If `cancelable` is `true`, the event can be cancelled via `preventDefault()`.

```
// Dispatch a custom event
var event = new CustomEvent('accordionOpen', {
    bubbles: true
});
content.dispatchEvent(event);
```

You can also add additional details about the event under the `detail` property. These can be accessed under `event.detail` in your event listener.

```
// Dispatch a custom event
var event = new CustomEvent('accordionOpen', {
    bubbles: true,
    detail: {
        contentID: content.id
    }
});
content.dispatchEvent(event);
```

Browser Compatibility

The Custom Event API works in all modern browsers, but has no IE support. You should wrap your event in an `if` statement to check that it's supported.

```
// Dispatch a custom event
if (typeof window.CustomEvent === 'function') {
    var event = new CustomEvent('accordionOpen', {
        bubbles: true,
        detail: {
            contentID: content.id
        }
    });
    content.dispatchEvent(event);
}
```

You can also push support back to IE9 with a small polyfill.[5]

```
/**
 * CustomEvent() polyfill
 * https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent#Polyfill
 */
(function () {
    if (typeof window.CustomEvent === 'function') return false;

    function CustomEvent(event, params) {
        params = params || { bubbles: false, cancelable: false, detail: undefined };
        var evt = document.createEvent('CustomEvent');
        evt.initCustomEvent(event, params.bubbles, params.cancelable, params.detail);
        return evt;
    }

    CustomEvent.prototype = window.Event.prototype;
    window.CustomEvent = CustomEvent;
})();
```

A custom event example

Here's an example of a the custom event from above in our accordion script.

```
// Run our accordion script
var runAccordion = function () {

    // ...

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

    // Dispatch a custom event
    if (typeof window.CustomEvent === 'function') {
        var event = new CustomEvent('accordionOpen', {
            bubbles: true,
            detail: {
                contentID: content.id
            }
        });
        content.dispatchEvent(event);
    }

};
```

And a developer might listen for it like this.

```
// Listen for all accordionOpen events
window.addEventListener('accordionOpen', function (event)
 {
    // Autoplay an HTML video when the accordion opens
    var video = event.target.querySelector('video');
    if (video) {
        video.play();
    }
}, false);
```

Should you use callbacks, events, or both?

So, which one should you use, when should you use it, and why?

I personally use events exclusively in my new plugins, and have started converting my older ones over from callbacks to events. They just provide so much more flexibility.

Callbacks require you to pass in a function at the time of initialization. With events, you can create multiple event listeners in multiple scripts, and they can be added at any time (both before and after initializing your script).

The one big downside to events is their asynchronous nature.

After the event is emitted, the plugin continues to run, and any listener events will run once the listener picks them up. Callbacks, on the otherhand, stop the plugin from doing anything else until the callback is run.

If you have a plugin where you want or need to prevent the plugin from continuing until any external hooks are run, callbacks are a better choice. Otherwise, I'd use custom events.

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
```

```
        // Feature test
        var supports = 'querySelector' in document && 'addEventListener' in window;
        if (!supports) return;

        // 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 += ' ' + settings.init;

    };

    // ...

})();
```

Allow multiple instances of your plugin to run at once

Depending on what your plugin does, developers may want to run more than once instance of it at the same time.

For example, on a webpage with multiple accordions, a developer may want to use different selectors for some of them.

```
// Initialize with defaults
accordion.init();

// Initialize with a custom selector
accordion.init({
    selectorToggle: '[data-toggle]';
});
```

While you can pass in the selector as an option, you can only have once instance of our script running at a time.

In the example above, the custom selector initialization causes `accordion.destroy()` to run and destroys the first initialization. Even if you removed the automatic `destroy()`, our `init()` method merges our new options into the `settings` variable and overrides the ones used by the first initialization.

In otherwords, we can currently only run one instance of the plugin a time. Let's fix that.

Instantiating our script

We're going to change the way our script works. Instead of initializing the plugin like this:

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

Developers will do this:

```
var accordion = new Accordion('.accordion-toggle');
```

This creates an entirely new instance of the plugin, with its own settings and options. It's commonly called *instantiating* a plugin.

To make this work, we need to make one major change inside our plugin (and a few smaller ones).

Building a Constructor

Right now, our plugin returns an object with all of our public APIs.

We're going to move all of the variables and methods that are unique to each instance of our plugin into a function called a constructor. The constructor will return the object of APIs, and our plugin will return the constructor.

JavaScript convention is to capitalize the names of functions that are initialized this way, so we'll also rename our plugin from `accordion` to `Accordion`.

```
var Accordion = (function (options) {

    'use strict';

    // Shared variables and utility methods...

    // Our plugin constructor
    // Can be named anything you want
    var BuildAccordion = function (options) {

        var publicAPIs = {};

        // Unique variables and methods

        // Initialize the plugin
        publicAPIs.init(options);

        // Return the public APIs
        return publicAPIs;

    };

    // Return the constructor
    return BuildAccordion;

})();
```

Common convention with this approach is often (but not always) to pass in the selector as a standalone argument instead of as part of the

`options`. This is in part because you may be instantiating several instances of the plugin, each with their own selector.

With that in mind, our plugin will look something like this.

```
var Accordion = (function (selector, options) {

    'use strict';

    // Shared variables and utility methods...

    // Our plugin constructor
    // Can be named anything you want
    var BuildAccordion = function (selector, options) {

        var publicAPIs = {};

        // Unique variables and methods

        // Initialize the plugin
        publicAPIs.init(options);

        // Return the public APIs
        return publicAPIs;

    };

    // Return the constructor
    return BuildAccordion;

})();
```

Rewriting our accordion plugin

Now, let's move some stuff around in our accordion plugin to make it work with this approach.

Unique variables and methods

First, let's pull in any variables and methods that need to be unique to each instance of our plugin. These are are typically variables that hold instance-specific data (like our `settings` variable) or reference instance-specific data (the `isActive()` method uses our `settings` variable internally).

```
var Accordion = (function (selector, options) {

    'use strict';

    // Shared variables and utility methods...

    // Our plugin constructor
    // Can be named anything you want
    var BuildAccordion = function (selector, options) {

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

        // Check if the target content is already active
        var isActive = function (content, toggle) {
```

```
            if (content.classList.contains(settings.conte
ntClass)) {
                content.classList.remove(settings.content
Class);
                toggle.classList.remove(settings.toggleCl
ass);

                // Dispatch a custom event
                if (typeof window.CustomEvent === 'functi
on') {
                    var event = new CustomEvent('accordio
nClose', {
                        bubbles: true,
                        detail: {
                            toggle: toggle
                        }
                    });
                    content.dispatchEvent(event);
                }

                return true;
            }
        };

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

```
                items[i].classList.remove(activeClass);

                // Dispatch a custom event
                if (selector === settings.selectorContent
) {
                    if (typeof window.CustomEvent === 'fu
nction') {
                        var event = new CustomEvent('acco
rdionClose', {
                            bubbles: true,
                            detail: {
                                toggle: null
                            }
                        });
                        items[i].dispatchEvent(event);
                    }
                }
            }
        };

        // Close all accordions and toggles
        publicAPIs.closeAccordions = function () {
            closeItems(settings.selectorContent, settings
.contentClass);
            closeItems(settings.selectorToggle, settings.
toggleClass);
        };

        // Run our accordion script
        var runAccordion = function () {
```

```
        var runAccordion = function () {
            // Only run if the clicked link was an accord
ion toggle
            if (!event.target.matches(settings.selectorTo
ggle)) 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
            var expanded = isActive(content, event.target
);
            if (expanded) return;

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

            // Open our target content area and toggle li
nk
            content.classList.add(settings.contentClass);
            event.target.classList.add(settings.toggleCla
ss);
```

```
            // Dispatch a custom event
            if (typeof window.CustomEvent === 'function')
 {
                var customEvent = new CustomEvent('accord
ionOpen', {
                    bubbles: true,
                    detail: {
                        toggle: event.target
                    }
                });
                content.dispatchEvent(customEvent);
            }
        };

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

            // Remove event listener
            document.removeEventListener('click', runAcco
rdion, false);

            // Remove the initialization class
            document.documentElement.classList.remove(set
tings.init);

            // Reset settings
            settings = null;
        };
```

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

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

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

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

            // Add our initialization class
            document.documentElement.className += ' ' + s
ettings.init;
        };

        // Initialize the plugin
        publicAPIs.init(options);

        // Return the public APIs
        return publicAPIs;

    };
```

```
    // Return the constructor
    return BuildAccordion;

})();
```

Shared variables and methods

Variables and methods that don't contain any instance-specific data can and should stay outside of the constructor to reduce the amount of memory is used each time we create a new one.

This includes our `matches()` polyfill, the `default` settings, and the `extend()` helper method.

```
var Accordion = (function (selector, options) {

    'use strict';

    // Element.matches() polyfill (simple version)
    // https://developer.mozilla.org/en-US/docs/Web/API/Element/matches#Polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches = Element.prototype.msMatchesSelector || Element.prototype.webkitMatchesSelector;
    }

    // Defaults
```

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

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

    /*!
     * Merge two or more objects together.
     * (c) 2017 Chris Ferdinandi, MIT License, https://gomakethings.com
     * @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;

        // 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 (obj.hasOwnProperty(prop)) {
                    // If property is an object, merge pr
operties
                    if (deep && Object.prototype.toString
.call(obj[prop]) === '[object Object]') {
                        extended[prop] = extend(extended[
prop], obj[prop]);
                    } else {
                        extended[prop] = obj[prop];
                    }
                }
            }
        };

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

```
        return extended;

    };

    // Our plugin constructor
    // Can be named anything you want
    var BuildAccordion = function (selector, options) {

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

        // Check if the target content is already active
        var isActive = function (content, toggle) {
            if (content.classList.contains(settings.conte
ntClass)) {
                content.classList.remove(settings.content
Class);
                toggle.classList.remove(settings.toggleCl
ass);

                // Dispatch a custom event
                if (typeof window.CustomEvent === 'functi
on') {
                    var event = new CustomEvent('accordio
nClose', {
                        bubbles: true,
                        detail: {
                            toggle: toggle
                        }
```

```
                    });
                    content.dispatchEvent(event);
                }

                return true;
            }
        };

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

                // Dispatch a custom event
                if (selector === settings.selectorContent
) {
                    if (typeof window.CustomEvent === 'fu
nction') {
                        var event = new CustomEvent('acco
rdionClose', {
                            bubbles: true,
                            detail: {
                                toggle: null
                            }
                        });
                        items[i].dispatchEvent(event);
                    }
```

```
                }
            }
        };

        // Close all accordions and toggles
        publicAPIs.closeAccordions = function () {
            closeItems(settings.selectorContent, settings
.contentClass);
            closeItems(settings.selectorToggle, settings.
toggleClass);
        };

        // Run our accordion script
        var runAccordion = function () {
            // Only run if the clicked link was an accord
ion toggle
            if (!event.target.matches(settings.selectorTo
ggle)) 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
```

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

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

            // Open our target content area and toggle li
nk
            content.classList.add(settings.contentClass);
            event.target.classList.add(settings.toggleCla
ss);



            // Dispatch a custom event
            if (typeof window.CustomEvent === 'function')
 {
                var customEvent = new CustomEvent('accord
ionOpen', {
                    bubbles: true,
                    detail: {
                        toggle: event.target
                    }
                });
                content.dispatchEvent(customEvent);
            }
        };

        publicAPIs.destroy = function () {
            // Only run if settings is set
```

```
            if (!settings) return;

            // Remove event listener
            document.removeEventListener('click', runAcco
rdion, false);

            // Remove the initialization class
            document.documentElement.classList.remove(set
tings.init);

            // Reset settings
            settings = null;
        };

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

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

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

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

```
            // Add our initialization class
            document.documentElement.className += ' ' + s
ettings.init;
        };

        // Initialize the plugin
        publicAPIs.init(options);

        // Return the public APIs
        return publicAPIs;

    };

    // Return the constructor
    return BuildAccordion;

})();
```

Updating the selector

Finally, we should change all references to `settings.selectorToggle` to `selector` to account for our new approach. We can also remove that option from our `defaults`.

```
var Accordion = (function (selector, options) {

    'use strict';
```

```
    // Element.matches() polyfill (simple version)
    // https://developer.mozilla.org/en-US/docs/Web/API/Element/matches#Polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches = Element.prototype.msMatchesSelector || Element.prototype.webkitMatchesSelector;
    }

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

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

    /*!
     * Merge two or more objects together.
     * (c) 2017 Chris Ferdinandi, MIT License, https://gomakethings.com
     * @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 defa
```

```
     * @returns {Object}            Merged values of defa
ults and options
     */
    var extend = function () {

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

        // 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 (obj.hasOwnProperty(prop)) {
                    // If property is an object, merge pr
operties
                    if (deep && Object.prototype.toString
.call(obj[prop]) === '[object Object]') {
                        extended[prop] = extend(extended[
prop], obj[prop]);
                    } else {
                        extended[prop] = obj[prop];
                    }
```

```
                }
            }
        };

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

        return extended;

    };

    // Our plugin constructor
    // Can be named anything you want
    var BuildAccordion = function (selector, options) {

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

        // Check if the target content is already active
        var isActive = function (content, toggle) {
            if (content.classList.contains(settings.conte
ntClass)) {
                content.classList.remove(settings.content
Class);
                toggle.classList.remove(settings.toggleCl
ass);
```

```
                // Dispatch a custom event
                if (typeof window.CustomEvent === 'functi
on') {
                    var event = new CustomEvent('accordio
nClose', {
                        bubbles: true,
                        detail: {
                            toggle: toggle
                        }
                    });
                    content.dispatchEvent(event);
                }

                return true;
            }
        };

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

                // Dispatch a custom event
                if (selector === settings.selectorContent
) {
                    if (typeof window.CustomEvent === 'fu
```

```
                    if (typeof window.CustomEvent === 'fu
nction') {
                        var event = new CustomEvent('acco
rdionClose', {
                            bubbles: true,
                            detail: {
                                toggle: null
                            }
                        });
                        items[i].dispatchEvent(event);
                    }
                }
            }
        };

        // Close all accordions and toggles
        publicAPIs.closeAccordions = function () {
            closeItems(settings.selectorContent, settings
.contentClass);
            closeItems(selector, settings.toggleClass);
        };

        // Run our accordion script
        var runAccordion = function () {
            // Only run if the clicked link was an accord
ion toggle
            if (!event.target.matches(selector)) 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
            var expanded = isActive(content, event.target
);
            if (expanded) return;

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

            // Open our target content area and toggle li
nk
            content.classList.add(settings.contentClass);
            event.target.classList.add(settings.toggleCla
ss);



            // Dispatch a custom event
            if (typeof window.CustomEvent === 'function')
 {
                var customEvent = new CustomEvent('accord
ionOpen', {
                    bubbles: true,
                    detail: {
                        toggle: event.target
```

```
                    }
                });
                content.dispatchEvent(customEvent);
            }
        };

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

            // Remove event listener
            document.removeEventListener('click', runAcco
rdion, false);

            // Remove the initialization class
            document.documentElement.classList.remove(set
tings.init);

            // Reset settings
            settings = null;
        };

        // Initialize our plugin
        publicAPIs.init = function (options) {
            // Feature test
            var supports = 'querySelector' in document &&
 'addEventListener' 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', runAccordi
on, false);

            // Add our initialization class
            document.documentElement.className += ' ' + s
ettings.init;
        };

        // Initialize the plugin
        publicAPIs.init(options);


        // Return the public APIs
        return publicAPIs;

    };

    // Return the constructor
    return BuildAccordion;

})();
```

When to use constructors

This approach is a little bit more complicated than previous versions of our plugin.

If a plugin only needs to be (or only can be) initialized once on a page, use the simple plugin structure. Otherwise, use a constructor, as it provides a lot more flexibility for developers.

Universal Module Definition (UMD)

If you want your plugin to work with RequireJS, Node, WebPack, Browserify, and other module bundlers, you need wrap your code in something called a Universal Module Definition (UMD) pattern.

UMD merges two differing approaches to modules—AMD and CommonJS—with the global variable technique we've been using up to this point.

In the boiilerplate below, replace the `myPlugin` part of `root.myPlugin` with the name of the global variable you want to use (for example, `root.Accordion`).

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

    'use strict';

    // Your code...

});
```

The accordion plugin as UMD

For example, if we converted the constructor version of our plugin to UMD, it would look like this.

```
(function (root, factory) {
    if ( typeof define === 'function' && define.amd ) {
        define([], function () {
```

```
            return factory(root);
        });
    } else if ( typeof exports === 'object' ) {
        module.exports = factory(root);
    } else {
        root.Accordion = factory(root);
    }
})(typeof global !== 'undefined' ? global : typeof window
 !== 'undefined' ? window : this, function (window) {

    'use strict';

    // Element.matches() polyfill (simple version)
    // https://developer.mozilla.org/en-US/docs/Web/API/E
lement/matches#Polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches = Element.prototype.msM
atchesSelector || Element.prototype.webkitMatchesSelector
;
    }

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

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

```
    };

    /*!
     * Merge two or more objects together.
     * (c) 2017 Chris Ferdinandi, MIT License, https://gomakethings.com
     * @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;

        // 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 ) {
```

```
            for (var prop in obj) {
                if (obj.hasOwnProperty(prop)) {
                    // If property is an object, merge pr
operties
                    if (deep && Object.prototype.toString
.call(obj[prop]) === '[object Object]') {
                        extended[prop] = extend(extended[
prop], obj[prop]);
                    } else {
                        extended[prop] = obj[prop];
                    }
                }
            }
        };

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

        return extended;

    };

    // Our plugin constructor
    // Can be named anything you want
    var BuildAccordion = function (selector, options) {

        // Variables
```

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

        // Check if the target content is already active
        var isActive = function (content, toggle) {
            if (content.classList.contains(settings.conte
ntClass)) {
                content.classList.remove(settings.content
Class);
                toggle.classList.remove(settings.toggleCl
ass);

                // Dispatch a custom event
                if (typeof window.CustomEvent === 'functi
on') {
                    var event = new CustomEvent('accordio
nClose', {
                        bubbles: true,
                        detail: {
                            toggle: toggle
                        }
                    });
                    content.dispatchEvent(event);
                }

                return true;
            }
        };

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

```
        var closeItems = function (selector, activeClass)
 {
            var items = document.querySelectorAll(selecto
r);
            for (var i = 0; i < items.length; i++) {
                items[i].classList.remove(activeClass);

                // Dispatch a custom event
                if (selector === settings.selectorContent
) {
                    if (typeof window.CustomEvent === 'fu
nction') {
                        var event = new CustomEvent('acco
rdionClose', {
                            bubbles: true,
                            detail: {
                                toggle: null
                            }
                        });
                        items[i].dispatchEvent(event);
                    }
                }
            }
        };

        // Close all accordions and toggles
        publicAPIs.closeAccordions = function () {
            closeItems(settings.selectorContent, settings
.contentClass);
```

```
            closeItems(selector, settings.toggleClass);
        };

        // Run our accordion script
        var runAccordion = function () {
            // Only run if the clicked link was an accordion toggle
            if (!event.target.matches(selector)) 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
            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(settings.contentClass);
            event.target.classList.add(settings.toggleCla
```

```
ss);


            // Dispatch a custom event
            if (typeof window.CustomEvent === 'function')
 {
                var customEvent = new CustomEvent('accord
ionOpen', {
                    bubbles: true,
                    detail: {
                        toggle: event.target
                    }
                });
                content.dispatchEvent(customEvent);
            }
        };

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

            // Remove event listener
            document.removeEventListener('click', runAcco
rdion, false);

            // Remove the initialization class
            document.documentElement.classList.remove(set
tings.init);

            // Reset settings
```

```
            settings = null;
        };

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

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

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

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

            // Add our initialization class
            document.documentElement.className += ' ' + s
ettings.init;
        };

        // Initialize the plugin
        publicAPIs.init(options);


        // Return the public APIs
```

```
        // Return the public APIs
        return publicAPIs;

    };

    // Return the constructor
    return BuildAccordion;

});
```

You would instantiate it just like before.

```
var accordion = new Accordion('.accordion-toggle');
```

Putting it all together

To make this all tangible, let's work on a project together. We'll take a script that mirrors content from a textarea or input field into a preview window and convert it into a plugin.

The starter template and complete project code are included in the source code[6] on GitHub.

Getting Setup

The starter template includes a small amount of CSS to make the layout look nicer, but none of it is neccessary for the script to work.

The script listens for `keyup` and `paste` events that happen inside fields that have the `.mirror` class. It uses the `[data-mirror]` attribute to get the selector of the container to mirror the content in, and updates that container's `innerHTML`.

Markup

```
<textarea class="mirror" data-mirror="#content"></textare
a>
<div id="content"></div>
```

JavaScript

```
/**
 * Element.matches() polyfill (simple version)
 * https://developer.mozilla.org/en-US/docs/Web/API/Element/matches#Polyfill
 */
if (!Element.prototype.matches) {
    Element.prototype.matches = Element.prototype.msMatchesSelector || Element.prototype.webkitMatchesSelector;
}

// Setup our change events
document.addEventListener('keyup', function () {

    // Check if the keyup/paste event happened in a field we want to mirror
    var mirror = document.activeElement;
    if (!mirror.matches('.mirror')) return;

    // Get the container to mirror our content into
    var target = document.querySelector(mirror.getAttribute('data-mirror'));
    if (!target) return;

    // Copy our field content into the container
    target.innerHTML = mirror.value.replace(/\r?\n/g, '<br>');

}, false);
```

```
document.addEventListener('paste', function () {

    // Check if the keyup/paste event happened in a field
 we want to mirror
    var mirror = document.activeElement;
    if (!mirror.matches('.mirror')) return;

    // Get the container to mirror our content into
    var target = document.querySelector(mirror.getAttribu
te('data-mirror'));
    if (!target) return;

    // Set a 1ms timeout to account for paste event happe
ning before text is pasted in
    window.setTimeout(function () {

        // Copy our field content into the container
        target.innerHTML = mirror.value.replace(/\r?\n/g,
 '<br>');

    }, 1);

}, false);
```

Planning

Let's put together a quick plan for what we'd like to do.

1. Modularize our script.
2. Scope our code inside a function wrapper.
3. Add an initialization function.
4. Let users pass in options to configure things.
5. Create a public method to mirror content.
6. Add some custom events developers can hook into.
7. Add a destroy function.
8. Allow multiple instances of the plugin to run at once.

Modularize the code

The first thing we want to do is modularize our code.

Currently, we have two event listeners that do more or less the same thing. The only difference is that our `paste` event adds a 1ms timeout before duplicating the input value (neccessary because the `paste` event fires *before* content is pasted, oddly). A 1ms delay is imperceivable, so there's no reason we can't use that with our `keyup` events, too.

Let's move that code to a new function called `changeHandler()`, and call that in our event listeners.

```
/**
 * Handle changes to our inputs and textareas
 */
var changeHandler = function () {

    // Check if the keyup/paste event happened in a field
 we want to mirror
```

```
    var mirror = document.activeElement;
    if (!mirror.matches('.mirror')) return;

    // Get the container to mirror our content into
    var target = document.querySelector(mirror.getAttribu
te('data-mirror'));
    if (!target) return;

    // Set a 1ms timeout to account for paste event happe
ning before text is pasted in
    window.setTimeout(function () {

        // Copy our field content into the container
        target.innerHTML = mirror.value.replace(/\r?\n/g,
 '<br>');

    }, 1);

};

// Detect changes to our fields
document.addEventListener('keyup', changeHandler, false);
document.addEventListener('paste', changeHandler, false);
```

Scoping our code

Next, let's 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 into the boilerplate, and change `root.myPlugin` to `root.Mirror`.

```
/*!
 * Universal Module Definition (UMD) Boilerplate
 * (c) 2017 Chris Ferdinandi, MIT License, https://gomakethings.com
 */
 (function (root, factory) {
    if ( typeof define === 'function' && define.amd ) {
        define([], function () {
            return factory(root);
        });
    } else if ( typeof exports === 'object' ) {
        module.exports = factory(root);
    } else {
        root.Mirror = factory(root);
    }
 })(typeof global !== 'undefined' ? global : typeof window !== 'undefined' ? window : this, function (window) {

    'use strict';

    /**
     * Element.matches() polyfill (simple version)
     * https://developer.mozilla.org/en-US/docs/Web/API/Element/matches#Polyfill
     */
    if (!Element.prototype.matches) {
```

```
    if (!Element.prototype.matches) {
        Element.prototype.matches = Element.prototype.msMatchesSelector || Element.prototype.webkitMatchesSelector;
    }

    /**
     * Handle changes to our inputs and textareas
     */
    var changeHandler = function () {

        // Check if the keyup/paste event happened in a field we want to mirror
        var mirror = document.activeElement;
        if (!mirror.matches('.mirror')) return;

        // Get the container to mirror our content into
        var target = document.querySelector(mirror.getAttribute('data-mirror'));
        if (!target) return;

        // Set a 1ms timeout to account for paste event happening before text is pasted in
        window.setTimeout(function () {

            // Copy our field content into the container
            target.innerHTML = mirror.value.replace(/\r?\n/g, '<br>');

        }, 1);
```

```
    };

    // Detect changes to our fields
    document.addEventListener('keyup', changeHandler, fal
se);
    document.addEventListener('paste', changeHandler, fal
se);

});
```

Add an initialization method

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

We'll add a placeholder object for our public methods, move our event listeners to an `.init()` method, and return our public methods object.

```
/*!
 * Universal Module Definition (UMD) Boilerplate
 * (c) 2017 Chris Ferdinandi, MIT License, https://gomake
things.com
 */
(function (root, factory) {
    if ( typeof define === 'function' && define.amd ) {
        define([], function () {
            return factory(root);
```

```
        });
    } else if ( typeof exports === 'object' ) {
        module.exports = factory(root);
    } else {
        root.Mirror = factory(root);
    }
})(typeof global !== 'undefined' ? global : typeof windo
w !== 'undefined' ? window : this, function (window) {

    'use strict';

    /**
     * Element.matches() polyfill (simple version)
     * https://developer.mozilla.org/en-US/docs/Web/API/E
lement/matches#Polyfill
     */
    if (!Element.prototype.matches) {
        Element.prototype.matches = Element.prototype.msM
atchesSelector || Element.prototype.webkitMatchesSelector
;
    }


    // Hold our public methods
    var publicAPIs = {};

    /**
     * Handle changes to our inputs and textareas
     */
    var changeHandler = function () {
```

```
        // Check if the keyup/paste event happened in a f
ield we want to mirror
        var mirror = document.activeElement;
        if (!mirror.matches('.mirror')) return;

        // Get the container to mirror our content into
        var target = document.querySelector(mirror.getAtt
ribute('data-mirror'));
        if (!target) return;

        // Set a 1ms timeout to account for paste event h
appening before text is pasted in
        window.setTimeout(function () {

            // Copy our field content into the container
            target.innerHTML = mirror.value.replace(/\r?\
n/g, '<br>');

        }, 1);

    };

    /**
     * Initialize the plugin
     */
    publicAPIs.init = function () {

        // Detect changes to our fields
        document.addEventListener('keyup', changeHandler
```

```
        document.addEventListener('keyup', changeHandler,
 false);
        document.addEventListener('paste', changeHandler,
 false);

    };

    // Return our public methods
    return publicAPIs;

});
```

Now we need to initialize our script.

```
// Initialize the plugin
Mirror.init();
```

Add user options

Next, let's add some options that users can configure themselves.

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. We'll include options to change the selector, and the data attribute that holds our content selector.

```
// Default settings
var defaults = {
    selector: '.mirror',
    content: '[data-mirror]'
};
```

Next, we'll create a variable called `settings` that will hold our plugin settings. Then, we'll add the `extend()` helper method[7].

```
// Default settings
var defaults = {
    selector: '.mirror',
    content: 'data-mirror'
};

// Define settings variable
var settings;

/*!
 * Merge two or more objects together.
 * (c) 2017 Chris Ferdinandi, MIT License, https://gomakethings.com
 * @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 () {
    // ...
};
```

In our `init()` method, we'll pass `options` in as an argument, and merge them into `defaults` to create our `settings` object.

```
/**
 * Initialize the plugin
 * @param {Object} options User options
 */
publicAPIs.init = function (options) {

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

    // Detect changes to our fields
    document.addEventListener('keyup', changeHandler, fal
se);
    document.addEventListener('paste', changeHandler, fal
se);

};
```

And finally, we'll change any references to `.mirror` and `[data-mirror]` to our settings values in the `changeHandler()` function.

```
/**
 * Handle changes to our inputs and textareas
 */
var changeHandler = function () {

    // Check if the keyup/paste event happened in a field
 we want to mirror
    var mirror = document.activeElement;
    if (!mirror.matches(settings.selector)) return;

    // Get the container to mirror our content into
    var target = document.querySelector(mirror.getAttribu
te(settings.content));
    if (!target) return;

    // Set a 1ms timeout to account for paste event happe
ning before text is pasted in
    window.setTimeout(function () {

        // Copy our field content into the container
        target.innerHTML = mirror.value.replace(/\r?\n/g,
 '<br>');

    }, 1);

};
```

Create a public method to save content

To give this plugin even more flexibility, let's provide a public method for mirroring content.

Developers can call it from their own scripts and cause content to mirror even if a `keyup` or `paste` event hasn't happened (useful if setting an input value with JavaScript).

```
/**
 * Copy our field content into the container
 * @param {Node} mirror The field to mirror
 * @param {Node} target The field to mirror content into
 */
publicAPIs.mirror = function (mirror, target) {
    target.innerHTML = mirror.value.replace(/\r?\n/g, '<b
r>');
};
```

We'll call our new public `mirror()` method from our `setTimeout()` function.

```
// Set a 1ms timeout to account for paste event happening
 before text is pasted in
window.setTimeout(function () {
    publicAPIs.mirror(mirror, target);
}, 1);
```

Add custom events

Now let's add custom events to our plugin.

A developer may want to, for example, save data to `localStorage` or to a database via an API when a field changes. Custom events will give them a hook to do that.

First, we'll add the CustomEvents polyfill[8] to our script.

Then, in the `mirror()` method, we'll add a `mirrored` event. We'll apply the event to our field, set `bubbles` to `true`, and pass in the content area as a `detail`.

```
/**
 * Copy our field content into the container
 * @param {Node} mirror The field to mirror
 * @param {Node} target The field to mirror content into
 */
publicAPIs.mirror = function (mirror, target) {
    target.innerHTML = mirror.value.replace(/\r?\n/g, '<b
r>');

    // Dispatch a custom event
    if (typeof window.CustomEvent === 'function') {
        var customEvent = new CustomEvent('mirrored', {
            bubbles: true,
            detail: {
                content: target
            }
        });
        mirror.dispatchEvent(customEvent);
    }
};
```

To test this, let's set up a `mirrored` event listener, and log both `event.target` (the mirrored field) and `event.detail.content` (the content area) into the console.

```
// Initialize the plugin
Mirror.init();

// Listen for `mirrored` events
document.addEventListener('mirrored', function (event) {
    console.log(event.target);
    console.log(event.detail.content);
}, false);
```

Add a destroy function

Now we can add a function to destroy our initialization. We'll remove our event listeners, and wipe our any mirrored fields.

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

```
/**
 * Destroy initialization
 */
publicAPIs.destroy = function () {

    // Only run if settings exist
    if (!settings) return;

    // Remove event listeners
    document.removeEventListener('keyup', changeHandler,
false);
```

```
false);
    document.removeEventListener('paste', changeHandler,
false);

    // Reset any mirrored content
    var mirrors = document.querySelectorAll(settings.sele
ctor);
    for (var i = 0; i < mirrors.length; i++) {
        var content = document.querySelector(mirrors[i].g
etAttribute(settings.content));
        if (!content) continue;
        content.innerHTML = '';
    }

    // Reset settings
    settings = null;

};

/**
 * Initialize the plugin
 * @param {Object} options User options
 */
publicAPIs.init = function (options) {

    // Destroy any existing initialization
    publicAPIs.destroy();

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

```
    // Detect changes to our fields
    document.addEventListener('keyup', changeHandler, fal
se);
    document.addEventListener('paste', changeHandler, fal
se);

};
```

If you open up the console tab in developer tools and run `Mirror.destroy()`, any mirrored content should disappear, and typing into one of our fields won't do anything.

Allow multiple instances of the plugin to run at once

A developer may want to run multiple instances of the plugin with different settings. Let's switch to a constructor model to let them do so.

First, let's add a constructor function named `Mirror()` to the end of our script, and pass our options into it. Then we'll return it for external access.

```
//
// Constructor
//

var Mirror = function (options) {

};

// Return our constructor
return Mirror;
```

Next, let's move any unique variables and methods into the constructor. That includes our `publicAPIs` and `settings` variables, and our `mirror()`, `changeHandler()`, `destroy()`, and `init()` methods.

The rest of our variables and helper methods can be shared across instantiations.

```
/*!
 * Universal Module Definition (UMD) Boilerplate
 * (c) 2017 Chris Ferdinandi, MIT License, https://gomakethings.com
 */
 (function (root, factory) {
    if ( typeof define === 'function' && define.amd ) {
        define([], function () {
            return factory(root);
        });
    } else if ( typeof exports === 'object' ) {
```

```
        module.exports = factory(root);
    } else {
        root.Mirror = factory(root);
    }
})(typeof global !== 'undefined' ? global : typeof windo
w !== 'undefined' ? window : this, function (window) {

    'use strict';


    //
    // Polyfills
    //

    /**
     * Element.matches() polyfill (simple version)
     * https://developer.mozilla.org/en-US/docs/Web/API/E
lement/matches#Polyfill
     */
    if (!Element.prototype.matches) {
        Element.prototype.matches = Element.prototype.msM
atchesSelector || Element.prototype.webkitMatchesSelector
;
    }

    /**
     * CustomEvent() polyfill
     * https://developer.mozilla.org/en-US/docs/Web/API/C
ustomEvent/CustomEvent#Polyfill
     */
```

```
    (function () {

        if (typeof window.CustomEvent === "function") ret
urn false;

        function CustomEvent(event, params) {
            params = params || { bubbles: false, cancelab
le: false, detail: undefined };
            var evt = document.createEvent('CustomEvent')
;
            evt.initCustomEvent(event, params.bubbles, pa
rams.cancelable, params.detail);
            return evt;
        }

        CustomEvent.prototype = window.Event.prototype;

        window.CustomEvent = CustomEvent;
    })();



    //
    // Shared Variables
    //

    // Default settings
    var defaults = {
        selector: '.mirror',
        content: 'data-mirror'
    };
```

```
    };


    //
    // Shared Methods
    //

    /*!
     * Merge two or more objects together.
     * (c) 2017 Chris Ferdinandi, MIT License, https://go
makethings.com
     * @param   {Boolean}  deep     If true, do a deep (o
r recursive) merge [optional]
     * @param   {Object}   objects  The objects to merge
together
     * @returns {Object}           Merged values of defa
ults and options
     */
    var extend = function () {

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

        // 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 (obj.hasOwnProperty(prop)) {
                    // If property is an object, merge pr
operties
                    if (deep && Object.prototype.toString
.call(obj[prop]) === '[object Object]') {
                        extended[prop] = extend(extended[
prop], obj[prop]);
                    } else {
                        extended[prop] = obj[prop];
                    }
                }
            }
        };

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

        return extended;

    };
```

```
//
// Constructor
//

var Mirror = function (options) {


    //
    // Unique Variables
    //

    // Hold our public methods
    var publicAPIs = {};

    // Default settings
    var defaults = {
        selector: '.mirror',
        content: 'data-mirror'
    };

    // Define settings variable
    var settings;


    //
    // Unique Methods
    //

    /**
     * Copy our field content into the container
```

```
         * Copy our field content into the container
         * @param {Node} mirror The field to mirror
         * @param {Node} target The field to mirror conte
nt into
         */
        publicAPIs.mirror = function (mirror, target) {
            target.innerHTML = mirror.value.replace(/\r?\
n/g, '<br>');

            // Dispatch a custom event
            if (typeof window.CustomEvent === 'function')
 {
                var customEvent = new CustomEvent('mirror
ed', {
                    bubbles: true,
                    detail: {
                        content: target
                    }
                });
                mirror.dispatchEvent(customEvent);
            }
        };

        /**
         * Handle changes to our inputs and textareas
         */
        var changeHandler = function () {

            // Check if the keyup/paste event happened in
 a field we want to mirror
```

```
            var mirror = document.activeElement;
            if (!mirror.matches(settings.selector)) retur
n;

            // Get the container to mirror our content in
to
            var target = document.querySelector(mirror.ge
tAttribute(settings.content));
            if (!target) return;

            // Set a 1ms timeout to account for paste eve
nt happening before text is pasted in
            window.setTimeout(function () {
                publicAPIs.mirror(mirror, target);
            }, 1);

        };

        /**
         * Destroy initialization
         */
        publicAPIs.destroy = function () {

            // Only run if settings exist
            if (!settings) return;

            // Remove event listeners
            document.removeEventListener('keyup', changeH
andler, false);
            document.removeEventListener('paste', changeH
```

```
andler, false);

            // Reset any mirrored content
            var mirrors = document.querySelectorAll(setti
ngs.selector);
            for (var i = 0; i < mirrors.length; i++) {
                var content = document.querySelector(mirr
ors[i].getAttribute(settings.content));
                if (!content) continue;
                content.innerHTML = '';
            }

            // Reset settings
            settings = null;

        };

        /**
         * Initialize the plugin
         * @param {Object} options User options
         */
        publicAPIs.init = function (options) {

            // Destroy any existing initialization
            publicAPIs.destroy();

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

            // Detect changes to our fields
```

```
            // Detect changes to our fields
            document.addEventListener('keyup', changeHand
ler, false);
            document.addEventListener('paste', changeHand
ler, false);

        };


        // Return our public methods
        return publicAPIs;

    };


    // Return our constructor
    return Mirror;

});
```

Now, we want to change the way we instantiate our plugin.

We'll pass our selector directly into the `Mirror()` function as the first argument, and our `options` in as the second. We can remove `selector` from our `defaults`. We'll also need to pass it into our constructor, and change references to `settings.selector` in our script.

```
// Default settings
var defaults = {
```

```
    content: 'data-mirror'
};

// ...

//
// Constructor
//

var Mirror = function (selector, options) {

    // ...

    /**
     * Handle changes to our inputs and textareas
     */
    var changeHandler = function () {

        // Check if the keyup/paste event happened in a f
ield we want to mirror
        var mirror = document.activeElement;
        if (!mirror.matches(selector)) return;

        // Get the container to mirror our content into
        var target = document.querySelector(mirror.getAtt
ribute(settings.content));
        if (!target) return;

        // Set a 1ms timeout to account for paste event h
appening before text is pasted in
```

```
            window.setTimeout(function () {
                publicAPIs.mirror(mirror, target);
            }, 1);

        };

        // ...

}
```

Finally, we want to automatically initialize our plugin on instantiation. Let's call `publicAPIs.init()` right before returning our public APIs.

```
// Initialize the plugin
publicAPIs.init(options);

// Return our public methods
return publicAPIs;
```

Now, we can instantiate our plugin.

```
var mirror = new Mirror('.mirror');
```

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

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[9] 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://polyfill.io↩
2. https://github.com/cferdinandi/vanilla-javascript-cheat-sheet/blob/master/helper-methods/extend.js↩
3. https://github.com/cferdinandi/vanilla-javascript-cheat-sheet/blob/master/polyfills/matches.js↩
4. https://gomakethings.com/attaching-multiple-elements-to-a-single-event-listener-in-vanilla-js/↩
5. https://github.com/cferdinandi/vanilla-javascript-cheat-sheet/blob/master/polyfills/CustomEvent.js↩
6. https://github.com/cferdinandi/writing-plugins-source-code/↩
7. https://github.com/cferdinandi/vanilla-javascript-cheat-sheet/blob/master/helper-methods/extend.js↩
8. https://github.com/cferdinandi/vanilla-javascript-cheat-sheet/blob/master/polyfills/CustomEvent.js↩
9. https://github.com/cferdinandi/writing-plugins-source-code/↩