Pocket Guides  Go Make Things

DOM MANIPULATION

with Vanilla JavaScript

CHRIS FERDINANDI

DOM Manipulation

By Chris Ferdinandi
Go Make Things, LLC
v4.0.0

Table of Contents

Intro

In this guide, you'll learn:

- How to get elements in the DOM.
- How to loop through arrays, objects, NodeLists, and other *iterable* items.
- How to get, set, and remove classes.
- How to manipulate, remove, and update styles.
- How to get, set, and remove attributes.
- How to listen for events in the DOM.
- Techniques for improving event listener performance.

A quick word about browser compatibility

This guide focuses on methods and APIs that are supported in all modern browsers. That means the latest versions of Edge, Chrome, Firefox, Opera, Safari, and the latest mobile browsers for Android and iOS.

Using the code in this guide

Let's get started!

Selectors

How to get elements in the DOM.

document.querySelectorAll()

Find all matching elements on a page. You can use any valid CSS selector.

```
// Get all button elements
let buttons =
document.querySelectorAll('button');

// Get all elements with the .bg-red class
let elemsRed =
document.querySelectorAll('.bg-red');

// Get all elements with the [data-snack]
attribute
let elemsSnacks =
document.querySelectorAll('[data-snack]');
```

document.querySelector()

Find the first matching element on a page.

```
// The first button
let button =
document.querySelector('button');

// The first element with the .bg-red class
let red = document.querySelector('.bg-red');

// The first element with a data attribute
of snack equal to carrots
let carrots = document.querySelector('[data-
snack="carrots"]');
```

If an element isn't found, `querySelector()` returns `null`. If you try to do something with the nonexistent element, an error will get thrown. You should check that a matching element was found before using it.

```
// An element that doesn't exist
let none = document.querySelector('.bg-
orange');

// Verify element exists before doing
anything with it
if (none) {
    // Do something...
}
```

Element.matches()

Check if an element would be selected by a particular selector or set of selectors. Returns `true` if the element is a match, and `false` when it's not.

```
// Check if the first .bg-red element has
the [data-snack attribute]
let red = document.querySelector('.bg-red');
if (red.matches('[data-snack]')) {
    console.log('Yummy snack!');
} else {
    console.log('No snacks');
}
```

Type-specific selector methods

There are other selector methods that target elements by specific type.

The `document.getElementById()` method gets elements by their ID, predates IE6. The `document.getElementsByName()` method returns a NodeList of elements with matching `[name]` attributes. It also has deep backwards compatibility.

If you wanted to get all elements of a certain type, you could use `document.getElementsByTagName()`, which works back to IE6. And the new kid on the block,

`document.getElementsByClassName()`, gets all elements that match a specific class. It works in IE9 and up.

I don't recommend using any of them.

I'm lazy. I don't like to think about which selector is the right one to use. The `document.querySelector()` and `document.querySelectorAll()` methods do everything those other methods do and more.

The toughest decision I have to make is whether I need all matching elements or just the first one.

Loops

How to loop through arrays, objects, and array-like objects.

for

Loop through arrays, NodeLists, and other array-like objects.

```
let sandwiches = ['turkey', 'tuna', 'ham',
'pb&j'];

// logs 0, "tuna", 1, "ham", 2, "turkey", 3,
"pb&j"
for (let i = 0; i < sandwiches.length; i++)
{
    console.log(i); // index
    console.log(sandwiches[i]); // value
}
```

- In the first part of the loop, before the first semicolon, we set a counter variable (typically `i`, but it can be anything) to `0`.
- The second part, between the two semicolons, is the test we check against after each iteration of the loop. In this case, we want to make sure the counter value is less than the total number of items in our array. We do this by checking the `.length` of our array.
- Finally, after the second semicolon, we specify what to run

after each loop. In this case, we're adding `1` to the value of `i` with `i++`.

We can then use `i` to grab the current item in the loop from our array.

for...of

Loop over *iterable objects*. That includes strings, arrays, and other array-like objects such as NodeLists, HTMLCollections, and HTMLFormControlsCollection, but *not* plain objects (`{}`).

In a `for...of` loop, you define a *variable* to represent the current item `of` the iterable that you're looping through. Inside the *block* (the stuff between curly brackets), you can use that variable to reference the current item.

```
let sandwiches = ['turkey', 'tuna', 'ham',
'pb&j'];

// logs "tuna", "ham", "turkey", "pb&j"
for (let sandwich of sandwiches) {
    console.log(sandwich);
}
```

for...in

Loop over plain objects (`{}`).

The first part, `key`, is a variable that gets assigned to the object key on each loop. The second part (in the example below, `lunch`), is the object to loop over.

In a `for...in` loop, you define a *variable* to represent the key `in` the object that you're looping through. Inside the *block* (the stuff between curly brackets), you can use that variable to get the key name and the value of that key.

```
let lunch = {
    sandwich: 'ham',
    snack: 'chips',
    drink: 'soda',
    desert: 'cookie',
    guests: 3,
    alcohol: false,
};

// logs "sandwich", "ham", "snack", "chips",
"drink", "soda", "desert", "cookie",
"guests", 3, "alcohol", false
for (let key in lunch) {
    console.log(key); // key
    console.log(lunch[key]); // value
}
```

Skipping and ending loops

You can skip to the next item in a loop using `continue`, or end the loop altogether with `break`. These work with `for`, `for...of`, and `for...in` loops.

```
/**
 * Skipping a loop
 */
let sandwiches = ['turkey', 'tuna', 'ham',
'pb&j'];

// logs "turkey", "tuna", "turkey", "pb&j"
for (let sandwich of sandwiches) {

    // Skip to the next item in the loop
    if (sandwich === 'ham') continue;

    console.log(sandwich);

}

/**
 * Breaking a loop
 */
let lunch = {
    sandwich: 'ham',
    snack: 'chips',
    drink: 'soda',
```

```
    desert: 'cookie',
    guests: 3,
    alcohol: false,
};

// logs "sandwich", "ham", "snack", "chips"
for (let key in lunch) {
    if (key === 'drink') break;
    console.log(lunch[key]);
}
```

Array.forEach() and NodeList.forEach()

The `Array.forEach()` and `NodeList.forEach()` methods provide a simpler way to iterate over arrays and NodeLists while still having access to the index.

You pass a callback function into the `forEach()` method. The callback itself accepts three arguments: the current item in the loop, the index of the current item in the loop, and the array itself. All three are optional, and you can name them anything you want.

```
let sandwiches = ['turkey', 'tuna', 'ham',
'pb&j'];

// logs 0, "tuna", 1, "ham", 2, "turkey", 3,
"pb&j"
sandwiches.forEach(function (sandwich,
index) {
    console.log(index); // index
    console.log(sandwich); // value
});
```

Unlike with `for`, `for...of`, and `for...in` loops, you can't end a `forEach()` callback function before it's looped through all items. You can `return` to end the current loop (like you would with `continue`), but there's no way to `break` the loop.

Because of that, I generally prefer using a `for...of` loop unless I explicitly need the `index`.

```
// Skip "ham"
// logs 0, "tuna", 2, "turkey", 3, "pb&j"
sandwiches.forEach(function (sandwich,
index) {
    if (sandwich === 'ham') return;
    console.log(index); // index
    console.log(sandwich); // value
});
```

Classes

How to add, remove, toggle, and check for classes on an element.

Element.classList

The `Element.classList` API provides a simple way to add, remove, toggle, and check for classes on an element.

Use the `add()` method to add a class, the `remove()` method to remove a class, the `toggle()` method to toggle a class on or off, and the `contains()` method to check if a class exists.

```
let elem =
document.querySelector('#sandwich');

// Add the .turkey class
elem.classList.add('turkey');

// Remove the .tuna class
elem.classList.remove('tuna');

// Toggle the .tomato class on or off
// (Add the class if it's not already on the
element, remove it if it is.)
elem.classList.toggle('tomato');

// Check if an element has the .mayo class
if (elem.classList.contains('mayo')) {
    console.log('add mayo!');
}
```

Element.className

Get all of the classes on an element as a string, add a class or classes, or completely replace or remove all classes.

```
let elem = document.querySelector('div');

// Get all of the classes on an element
let elemClasses = elem.className;

// Add a class to an element
elem.className += ' vanilla-js';

// Completely replace all classes on an
element
elem.className = 'new-class';
```

Styles

How to get and set styles (as in, CSS) for an element.

Vanilla JavaScript uses camel cased versions of the attributes you would use in CSS. The Mozilla Developer Network provides a comprehensive list of available attributes and their JavaScript counterparts.

Inline Styles

Get and set inline styles for an element with the `Element.style` property.

The `Element.style` property is a read-only object. You can get and set individual style properties on it using camelCase style names as properties on the `Element.style` object.

```
<p id="sandwich" style="background-color:
green; color: white;">
    Sandwich
</p>
```

```
let sandwich =
document.querySelector('#sandwich');

// Get a style
// If this style is not set as an inline
style directly on the element, it returns an
empty string
let bgColor =
sandwich.style.backgroundColor; // this will
return "green"
let fontWeight = sandwich.style.fontWeight;
// this will return ""

// Set the background-color style property
sandwich.style.backgroundColor = 'purple';
```

You can also get and set a string representation of the entire inline `style` property on the element itself with the `Element.style.cssText` property.

```
// Get the styles on an element
// returns "background-color: green; color: white;"
let styles = sandwich.style.cssText;

// Completely replace the inline styles on an element
sandwich.style.cssText = 'font-size: 2em; font-weight: bold;';

// Add additional styles
sandwich.style.cssText += 'color: purple;';
```

Computed Styles

The `window.getComputedStyle()` method gets the actual computed style of an element. This factors in browser default stylesheets as well as external styles being used on the page.

```
let sandwich = document.querySelector('#sandwich');
let bgColor = window.getComputedStyle(sandwich).backgroundColor;
```

Attributes & Properties

How to get, set, and remove attributes for an element.

Element.getAttribute(), Element.setAttribute(), Element.removeAttribute(), and Element.hasAttribute()

Get, set, remove, and check for the existence of attributes (including data attributes) on an element.

If an attribute does not exist on an element, the `Element.getAttribute()` method returns `null`.

```
let elem = document.querySelector('#lunch');

// Get the value of the [data-sandwich]
attribute
let sandwich = elem.getAttribute('data-
sandwich');

// Set a value for the [data-sandwich]
attribute
elem.setAttribute('data-sandwich',
'turkey');

// Remove the [data-chips] attribute
elem.removeAttribute('data-chips');

// Check if an element has the `[data-
drink]` attribute
if (elem.hasAttribute('data-drink')) {
    console.log('Add a drink!');
}
```

Properties

HTML elements have dozens of properties that you can access directly.

Some of them are *read only*, meaning you can get their value but not set it. Others can be used to both read and set values. You can find a full list on the Mozilla Developer Network.

```
let elem = document.querySelector('#main');

// Get the ID of the element
// returns "main"
let id = elem.id;

// Set the ID of the element
elem.id = 'secondary';

// Get the parentNode of the element
// This property is read-only
let parent = elem.parentNode;
```

Attributes vs. Properties

In JavaScript, an element has attributes and properties. The terms are often used interchangeably, but they're actually two separate things.

An *attribute* is the *initial state* when rendered in the DOM. A *property* is the *current state.*

In most cases, attributes and properties are kept in-sync automatically. For example, when you use `setAttribute()` to update an ID attribute, the `id` property is updated as well.

```
<p>Hello</p>
```

```
let p = document.querySelector('p');

// Update the ID
p.setAttribute('id', 'first-paragraph');

// These both return "first-paragraph"
let id1 = p.getAttribute('id');
let id2 = p.id;
```

However, user-changeable form properties—noteably, `value`, `checked`, and `selected`—are *not* automatically synced.

```
<label for="greeting">Greeting</label>
<input type="text" id="greeting">
```

```
let greeting =
document.querySelector('#greeting');

// Update the value
greeting.setAttribute('value', 'Hello
there!');

// If you haven't made any updates to the
field, these both return "Hello there!"
// If you HAVE updated the field, val1
returns whatever was typed in the field
instead
let val1 = greeting.value;
let val2 = greeting.getAttribute('value');
```

If you try to update the `value` property directly, that *will* update the UI.

```
greeting.value = 'Hello there!';
```

This allows you to choose different approaches depending on whether you want to overwrite user updates or not.

If you want to update a field, but *only if* the user hasn't made any changes, use `Element.setAttribute()`. If you want to overwrite anything they've done, use the `value` property.

Event Listeners

How to listen for browser events and run callback functions when they happen.

EventTarget.addEventListener()

Listen for events on an element. You can find a full list of available events on the Mozilla Developer Network.

Run the `EventTarget.addEventListener()` method on the element you want to listen for events on. It accepts two arguments: the event to listen for, and a callback function to run when the event happens.

You can pass the `event` into the callback function as an argument. The `event.target` property is the element that triggered the event. The `event` object has other properties as well, many of them specific to the type of event that occurred.

```
let btn = document.querySelector('#click-
me');

btn.addEventListener('click', function
(event) {
    console.log(event); // The event details
    console.log(event.target); // The
clicked element
});
```

Multiple Targets

The `EventTarget.addEventListener()` method only be attached to an individual element. You can't attach it to an array or node list of matching elements like you might in jQuery or other frameworks.

```
// This won't work!
let btns =
document.querySelectorAll('.click-me');

btns.addEventListener('click', function
(event) {
    console.log(event); // The event details
    console.log(event.target); // The
clicked element
});
```

For performance reasons, you also **should not** loop over each element and attach an even listener to it.

```
/**
 * This works, but it's bad for performance
 * DON'T DO IT!
 */
let btns =
document.querySelectorAll('.click-me');

for (let btn of btns) {
    btn.addEventListener('click', function
(event) {
        console.log(event); // The event
details
        console.log(event.target); // The
clicked element
    });
}
```

Fortunately, there's a really easy *and* performant way to get a jQuery-like experience: *event delegation* or *event bubbling*.

Instead of listening for an event on specific elements, you attach your listener to a parent element that your elements are contained within, such as the `window` or `document`. Events that happens on elements inside it *bubble up*. We can then check to see if the item that triggered the event has a matching selector.

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

    // If the clicked element has the
`.click-me` class, it's a match!
    if (event.target.matches('.click-me')) {
        // Do something...
    }

});
```

Yes, it is actually better for performance to listen to all clicks on the document than have a bunch of individual event listeners.

As a side benefit, you can dynamically load matching elements to the DOM after the event listener is already set up and it will still work.

Capturing events that don't bubble

Certain events, like `focus`, don't bubble. In order to use event delegation with events that don't bubble, you can set an optional third argument on the `EventTarget.addEventListener()` method, called `useCapture`, to `true`.

```
// Listen for all focus events in the
document
document.addEventListener('focus', function
(event) {
    // Run functions whenever an element in
the document comes into focus
}, true);
```

You can determine if `useCapture` should be set to `true` or `false` by looking at the event details page on the Mozilla Developer Network (like this one for the `focus` event).

If *Bubbles* in the chart at the top of the page is "No," you need to set `useCapture` to `true` to use event delegation.

Multiple Events

In vanilla JavaScript, each event type requires it's own event listener. Unfortunately, you *can't* pass in multiple events to a single listener like you might in jQuery and other frameworks.

```
/**
 * This won't work!
 */
window.addEventListener('click, scroll',
function (event) {
    console.log(event); // The event details
    console.log(event.target); // The
clicked element
});
```

Instead, create a named function and pass that into your event listener. This lets you avoid writing the same code over and over again, and keeps your code more DRY.

For named callback functions, *do not* include the parentheses (`()`) on the function.

The `event` object is automatically passed in as an argument. You can determine which type of event triggered the callback function with the `event.type` property.

```
// Setup our function to run on various
events
function logTheEvent (event) {
    console.log('The following event
happened: ' + event.type);
}

// Add our event listeners
document.addEventListener('click',
logTheEvent);
window.addEventListener('scroll',
logTheEvent);
```

Putting it all together

To make this all tangible, let's work on a project together. We'll build a script that let's users toggle the visibility of password fields in a form.

Getting Setup

The template has some starting markup: a form with two password fields and some buttons.

In the form, there's a button with the `[data-password]` attribute that will be used to toggle the password field visibility. That button has two additional attributes.

The `[type="button"]` attribute prevents the button from submitting the form when clicked.

```
<form>

    <label for="current-pw">Current
Password</label>
    <input type="password" id="current-pw">

    <label for="new-pw">New Password</label>
    <input type="password" id="new-pw">

    <p><button type="button" data-
password>Show Passwords</button></p>

    <p><button>Change Password</button></p>

</form>
```

I've also added some default CSS so that we can focus on the JavaScript.

Alright, let's get started.

Listening for clicks

The first thing we want to do is detect clicks on our `[data-password]` button. Let's use the `document.querySelector()` method to get button and save it to the `toggle` variable.

```
// Get the password toggle
let toggle = document.querySelector('[data-
password]');
```

Next, we'll use the `addEventListener()` method to listen for `click` events on it, and do things when the button is clicked.

```
// Get the password toggle
let toggle = document.querySelector('[data-
password]');

// Listen for clicks on the toggle button
toggle.addEventListener('click', function
(event) {
    // Do stuff...
});
```

Determining whether to show or hide passwords

When the button is clicked, we need to determine if we should show or hide passwords. One simple way to do that is to check if the button is currently selected or not.

The `[aria-pressed]` attribute is used to tell screen readers (software that people with visual impairments use to interact with web pages) if a state-based button like this one is pressed

or not. It's exactly what we need!

The `[aria-pressed]` attribute has a value of `true` when the button is selected, and `false` when it's not. Let's start by adding it to our button.

```
<button type="button" data-password aria-
pressed="false">Show Passwords</button>
```

Inside our event listener's callback function, we can check the value of the `[aria-pressed]` attribute to determine if the button is currently active or not.

We'll use the `event.target` to get the button that triggered the click event. We could use our `toggle` variable, but I want to show the different options you have.

We'll use the `getAttribute()` method to get the value of `[aria-pressed]`.

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

    // Get the value of the [aria-pressed]
attribute
    let pressed =
event.target.getAttribute('aria-pressed');

});
```

Next, we'll use the *strict equals operator* (===) to check if `pressed` has a value of `false`.

If it does, we need to show the password fields and update the value of `[aria-pressed]` to `true`. If not, we need to hide the fields and change the value to `false`.

We can use the `setAttribute()` method to set the `[aria-pressed]` attribute.

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

    // Get the value of the [aria-pressed]
attribute
    let pressed =
event.target.getAttribute('aria-pressed');

    // If button isn't pressed yet, press it
and show fields
    // Otherwise, unpress it and hide the
fields
    if (pressed === 'false') {
        event.target.setAttribute('aria-
pressed', 'true');
        // Show the fields...
    } else {
        event.target.setAttribute('aria-
pressed', 'false');
        // Hide the fields...
    }

});
```

Showing and hiding passwords

Now, we're ready to actually show and hide our password fields. Let's use the `document.querySelectorAll()` method to get all fields with the `[type="password"]` attribute.

```
// Get the password toggle and password
fields
let toggle = document.querySelector('[data-
password]');
let fields =
document.querySelectorAll('[type="password"]
');
```

We'll use a `for...of` loop to loop through each of our `fields`, and the `type` property to update the field type as needed.

If the password should be visible, we'll change the`type` to `text`. If it should be hidden, we'll change it back to`password`.

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

    // Get the value of the [aria-pressed]
attribute
    let pressed =
event.target.getAttribute('aria-pressed');

    // If button isn't pressed yet, press it
and show fields
    // Otherwise, unpress it and hide the
fields
    if (pressed === 'false') {
        event.target.setAttribute('aria-
pressed', 'true');
        for (let field of fields) {
            field.type = 'text';
        }
    } else {
        event.target.setAttribute('aria-
pressed', 'false');
        for (let field of fields) {
            field.type = 'password';
        }
    }

});
```

Now, the passwords will show or hide based on the button state.

There's just one last thing to do: style the button so users can visually tell if it's selected or not.

Styling the button

One really cool thing about attributes is that they can be used to style elements just like classes and IDs.

Since the `[aria-pressed]` attribute already holds information about whether or not the button is selected, it makes sense to use it to style the button visually as well. Let's give a blue background with white text when it's active.

```
/**
 * Active Button Style
 */
[aria-pressed="true"] {
    background-color: #0088cc;
    color: #ffffff;
}
```

Congratulations! You just created a show password script using a variety of DOM manipulation techniques.

About the Author

Hi, I'm Chris Ferdinandi. I believe there's a simpler, more resilient way to make things for the web.

I'm the author of the Vanilla JS Pocket Guide series, creator of the Vanilla JS Academy training program, and host of the Vanilla JS Podcast. My developer tips newsletter is read by thousands of developers each weekday.

I love pirates, puppies, and Pixar movies, and live near horse farms in rural Massachusetts.

You can find me:

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