Pocket Guides  Go Make Things

DOM INJECTION & TRAVERSAL

with Vanilla JavaScript

CHRIS FERDINANDI

DOM Injection and Traversal

By Chris Ferdinandi

Go Make Things, LLC

v2.1.0

Table of Contents

Intro

In this guide, you'll learn:

- How to detect when the viewport is ready.
- How to manipulation HTML.
- How to add and remove elements from the DOM.
- How to traverse up and down the DOM.
- How to detect when elements are in the viewport.
- How to calculate distances in the viewport.

A quick word about browser compatibility

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

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

Let's get started…

DOM Ready

How to detect when the DOM is ready before running code.

Note: *If you're loading your scripts in the footer (which you generally should be for performance reasons), the* `ready()` *method isn't really needed. It's just a habit from the "load everything in the header" days.*

Vanilla JavaScript provides a native way to do this: the `DOMContentLoaded` event for `addEventListener`.

But... if the DOM is already loaded by the time you call your event listener, the event never happens and your function never runs.

Below is a super lightweight helper method that does the same thing as jQuery's `ready()` method. This helper method does two things:

1. Check to see if the document is already `interactive` or `complete`. If so, it runs your function immediately.
2. Otherwise, it adds a listener for the `DOMContentLoaded` event.

```
/**
 * Run event after DOM is ready
  * @param  {Function} fn Callback function
 */
var ready = function ( fn ) {

    // Sanity check
    if ( typeof fn !== 'function' ) return;

    // If document is already loaded, run method
    if ( document.readyState === 'interactive' || documen
t.readyState === 'complete' ) {
        return fn();
    }

    // Otherwise, wait until document is loaded
    document.addEventListener( 'DOMContentLoaded', fn, fa
lse );

};

// Example
ready(function() {
    // Do stuff...
});
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

HTML

Use `.innerHTML` to get and set HTML content in an element.

```
var elem = document.querySelector( '#some-elem' );

// Get HTML content
var html = elem.innerHTML;

// Set HTML content
elem.innerHTML = 'We can dynamically change the HTML. We can even include HTML elements like <a href="#">this link</a>.';

// Add HTML to the end of an element's existing content
elem.innerHTML += ' Add this after what is already there. ';

// Add HTML to the beginning of an element's existing content
elem.innerHTML = 'We can add this to the beginning. ' + elem.innerHTML;

// You can inject entire elements into other ones, too
elem.innerHTML += '<p>A new paragraph</p>';
```

Browser Compatibility

Works in all modern browsers, and IE9 and above. **IE9 Exception:** Tables and selects require IE10 and above.[1]

DOM Injection

How to add and remove elements in the DOM.

Injecting an element into the DOM

Injecting an element into the DOM requires us to combine a few JavaScript methods.

1. Get the element you want to add our new element before or after.
2. Create our new element using the `createElement()` method.
3. Add content to our element with `innerHTML`.
4. Add any other attributes we want to our element (an ID, classes, etc.).
5. Insert the element using the `insertBefore()` method.

```
// Get the element you want to add your new element befor
e or after
var target = document.querySelector( '#some-element' );

// Create the new element
// This can be any valid HTML element: p, article, span,
etc...
var div = document.createElement( 'div' );

// Add content to the new element
div.innerHTML = 'Your content, markup, etc.';

// You could also add classes, IDs, and so on
// div is a fully manipulatable DOM Node

// Insert the element before our target element
// The first argument is our new element.
// The second argument is our target element.
target.parentNode.insertBefore( div, target );

// Insert the element after our target element
// Instead of passing in the target, you pass in target.n
extSibling
target.parentNode.insertBefore( div, target.nextSibling )
;
```

Browser Compatibility

Works in all modern browsers, and IE9 and above. **IE9 Exception:** Adding content to tables and selects with `innerHTML` require IE10 and above.[2]

Removing an element from the DOM

You can easily hide an element in the DOM by setting it's `style.display` to `none`.

```
var elem = document.querySelector('#some-element');
elem.style.display = 'none';
```

To *really* remove an element from the DOM, we can use the `removeChild()` method. This method is called against our target element's parent, which we can get with `parentNode`.

```
var elem = document.querySelector('#some-element');
elem.parentNode.removeChild( elem );
```

Browser Compatibility

Works in all modern browsers, and at least IE6.

Traversing Up the DOM

How to traverse up the DOM.

parentNode

Use `parentNode` to get the parent of an element.

```
var elem = document.querySelector( '#some-elem' );
var parent = elem.parentNode;
```

You can also string them together to go several levels up.

```
var levelUpParent = elem.parentNode.parentNode;
```

Browser Compatibility

Works in all modern browsers, and at least back to IE6.

closest()

Use `closest()` to get the closest parent up the DOM tree that matches against a selector.

```
var elem = document.querySelector( '#some-elem' );
var closestSandwich = elem.closest( '[data-sandwich]' );
```

Note: *This method can also be used in event listeners to determine if the* `event.target` *is inside of a particular element or not (for example, did a click happen inside of a dropdown menu?).*

Browser Compatibility

The `closest()` method works in most modern browsers but has pretty spotty backwards compatibility. Fortunately, a lightweight polyfill adds functionality to any browser that supports `querySelectorAll()` (so, IE8 and up).

```
if (window.Element && !Element.prototype.closest) {
    Element.prototype.closest = function(s) {
        var matches = (this.document || this.ownerDocumen
t).querySelectorAll(s),
            i,
            el = this;
        do {
            i = matches.length;
            while (--i >= 0 && matches.item(i) !== el) {}
;
        } while ((i < 0) && (el = el.parentElement));
        return el;
    };
}
```

getParents()

`getParents()` is a helper method I wrote that returns an array of parent elements, optionally matching against a selector. It's a vanilla JavaScript equivalent to jQuery's `.parents()` method.

It starts with the element you've passed in itself, so pass in `elem.parentNode` to skip to the first parent element instead.

```
/**
 * Get all of an element's parent elements up the DOM tre
e
```

```
 * @param  {Node}   elem     The element
 * @param  {String} selector Selector to match against [optional]
 * @return {Array}           The parent elements
 */
var getParents = function ( elem, selector ) {

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

    // Setup parents array
    var parents = [];

    // Get matching parent elements
    for ( ; elem && elem !== document; elem = elem.parent
```

```
Node ) {

        // Add matching parents to array
        if ( selector ) {
            if ( elem.matches( selector ) ) {
                parents.push( elem );
            }
        } else {
            parents.push( elem );
        }

    }

    return parents;

};

// Example
var elem = document.querySelector( '#some-elem' );
var parents = getParents( elem.parentNode );
var parentsWithWrapper = getParents( elem.parentNode, '.w
rapper' );
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

getParentsUntil()

`getParentsUntil()` is a helper method I wrote that returns an array of parent elements until a matching parent is found, optionally matching against a selector. It's a vanilla JavaScript equivalent to jQuery's `.parentsUntil()` method.

It starts with the element you've passed in itself, so pass in `elem.parentNode` to skip to the first parent element instead.

```
/**
 * Get all of an element's parent elements up the DOM tre
e until a matching parent is found
 * @param  {Node}   elem     The element
 * @param  {String} parent   The selector for the parent
to stop at
 * @param  {String} selector The selector to filter again
st [optionals]
 * @return {Array}           The parent elements
 */
var getParentsUntil = function ( elem, parent, selector )
 {

    // Element.matches() polyfill
    if (!Element.prototype.matches) {
        Element.prototype.matches =
            Element.prototype.matchesSelector ||
            Element.prototype.mozMatchesSelector ||
            Element.prototype.msMatchesSelector ||
            Element.prototype.oMatchesSelector ||
```

```
            Element.prototype.webkitMatchesSelector ||
            function(s) {
                var matches = (this.document || this.ownerDocument).querySelectorAll(s),
                    i = matches.length;
                while (--i >= 0 && matches.item(i) !== this) {}
                return i > -1;
            };
    }

    // Setup parents array
    var parents = [];

    // Get matching parent elements
    for ( ; elem && elem !== document; elem = elem.parentNode ) {

        if ( parent ) {
            if ( elem.matches( parent ) ) break;
        }

        if ( selector ) {
            if ( elem.matches( selector ) ) {
                parents.push( elem );
            }
            break;
        }
```

```
            parents.push( elem );

        }

        return parents;

    };

    // Examples
    var elem = document.querySelector( '#some-element' );
    var parentsUntil = getParentsUntil( elem, '.some-class' )
    ;
    var parentsUntilByFilter = getParentsUntil( elem, '.some-
    class', '[data-something]' );
    var allParentsUntil = getParentsUntil( elem );
    var allParentsExcludingElem = getParentsUntil( elem.paren
    tNode );
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Traversing Down the DOM

querySelector() and querySelectorAll()

The `querySelector()` and `querySelectorAll()` APIs aren't limited to just running on the `document`. They can be run on any element to search only for elements inside of it.

```
var elem = document.querySelector( '#some-elem' );

// Find the first element inside `#some-elem` that has a
`[data-snack]` attribute
var snack = elem.querySelector( '[data-snack]' );

// Get all divs inside `#some-elem`
var divs = elem.querySelectorAll( 'div' );
```

Browser Compatibility

Same as `querySelectorAll` and `querySelector`.

children

While `querySelector()` and `querySelectorAll()` search through all levels within a nested DOM/HTML structure, you may want to just get immediate decedants of a particular element. Use `children` for this.

```
var elem = document.querySelector( '#some-elem' );
var decendants = wrapper.children;
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Traversing Sideways in the DOM

`getSiblings` is a helper method I wrote that gets the siblings of an element in the DOM. For example: if you had a list item (`<li>`) and wanted to grab all of the other items in the list.

```
/**
 * Get all siblings of an element
 * @param  {Node}  elem The element
 * @return {Array}      The siblings
 */
var getSiblings = function ( elem ) {
    var siblings = [];
    var sibling = elem.parentNode.firstChild;
    for ( ; sibling; sibling = sibling.nextSibling ) {
        if ( sibling.nodeType === 1 && sibling !== elem )
 {
            siblings.push( sibling );
        }
    }
    return siblings;
};

// Example
var elem = document.querySelector( '#some-element' );
var siblings = getSiblings( elem );
```

Browser Compatibility

Works in all modern browsers, and IE6 and above.

The Viewport

How to detect viewport dimensions, and check if an element is within it.

Get the viewport height

There are two methods to get the viewport height: `window.innerHeight` and `document.documentElement.clientHeight`. The former is more accurate. The latter has better browser support.

To get the best of both worlds, try `innerHeight` first, and fallback to `clientHeight` if not supported.

```
var viewportHeight = window.innerHeight || document.docum
entElement.clientHeight;
```

Browser Compatibility

`innerHeight` works in all modern browsers, and IE9 and above.
`clientHeight` works in all modern browsers, and IE6 and above.

Get the viewport width

There are two methods to get the viewport width: `window.innerWidth` and `document.documentElement.clientWidth`. The former is more accurate. The latter has better browser support.

To get the best of both worlds, try `innerWidth` first, and fallback to `clientWidth` if not supported.

```
var viewportWidth = window.innerWidth || document.documen
tElement.clientWidth;
```

Browser Compatibility

`innerWidth` works in all modern browsers, and IE9 and above. `clientWidth` works in all modern browsers, and IE6 and above.

Check if an element is in the viewport or not

`isInViewport` is a helper method I wrote to check if an element is in the viewport or not. It returns `true` if the element is in the viewport, and `false` if it's not.

```
/**
 * Determine if an element is in the viewport
 * @param  {Node}    elem The element
 * @return {Boolean}      Returns true if element is in t
he viewport
 */
var isInViewport = function ( elem ) {
    var distance = elem.getBoundingClientRect();
    return (
        distance.top >= 0 &&
        distance.left >= 0 &&
        distance.bottom <= (window.innerHeight || documen
t.documentElement.clientHeight) &&
        distance.right <= (window.innerWidth || document.
documentElement.clientWidth)
    );
};

// Example
var elem = document.querySelector( '#some-element' );
isInViewport( elem ); // Boolean: returns true/false
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Distances

How to get the distances between elements in the DOM.

Get the currently scrolled distance from the top of the page

Use `pageYOffset` to get the distance the user has scrolled from the top of the page.

```
var distance = window.pageYOffset;
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Get an element's distance from the top of the page

`getOffsetTop` is a helper method I wrote to get an element's distance from the top of the document.

```
/**
 * Get an element's distance from the top of the Document
.
 * @private
 * @param  {Node} elem The element
 * @return {Number}    Distance from the top in pixels
 */
var getOffsetTop = function ( elem ) {
    var location = 0;
    if (elem.offsetParent) {
        do {
            location += elem.offsetTop;
            elem = elem.offsetParent;
        } while (elem);
    }
    return location >= 0 ? location : 0;
};

// Example
var elem = document.querySelector( '#some-element' );
var distance = getOffsetTop( elem );
```

Browser Compatibility

Works in all modern browsers, and IE9 and above.

Get the distance between two elements in the

DOM

We can use `getOffsetTop` to calculate the distance from the top of the viewport for our two elements, and then use simple match to figure out how far apart they are.

```
var elem1 = document.querySelector( '#first-element' );
var elem2 = document.querySelector( '#second-element' );

// Distance between them
// If negative, elem1 is below elem2
var distance = getOffsetTop( elem2 ) - getOffsetTop( elem
1 );

// To always get a positive number
var distancePositive = Math.abs( getOffsetTop( elem2 ) -
getOffsetTop( elem1 ) );
```

Putting it all together

To make this all tangible, let's work on a project together. We'll build a script that lazy loads images after they enter the viewport.

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

Getting Setup

I've dropped some placeholder markup into the template to help you get started.

Throughout the page copy, I've included empty `<figure>` elements that will eventually contain our lazy loaded images. Each one has the `.lazy-load` class on it, which we'll use as a hook in our JavaScript to get the images that should be lazy loaded.

We also need to include some information about the image itself, which we can do with data attributes. The `[data-image]` attribute includes a URL to the image to be lazy loaded, and the `[data-caption]` attribute (if included) is the caption we'd like to include with our image.

```
<figure class="lazy-load" data-caption="A lifeguard stati
on on a deserted beach" data-image="img/beach.jpg"></figu
re>
```

I've also included some lightweight CSS to make our images responsive, and our captions look good.

```
figure {
    margin: 0 0 1.4em;
}

caption {
    color: #808080;
    display: block;
    font-size: 0.8em;
    font-style: italic;
    padding: 0.25em 0 0.5em;
    text-align: center;
}

img {
    height: auto;
    max-width: 100%;
}
```

Alright, let's get coding!

Getting all of the images

The first thing we need to do is get all of the images that we want to lazy load. We'll use `querySelectorAll()` for that.

```
var images = document.querySelectorAll('.lazy-load');
```

Whenever the visitor scrolls, we want to loop through each of our

images, and check to see if it's in the viewport. We'll use `addEventListener()` to listen for scroll events, with a simple `for` loop to loop through each of our image placeholders.

```
var images = document.querySelectorAll('.lazy-load');

window.addEventListener('scroll', function (event) {
    for (var i = 0; i < images.length; i++) {
        // Check if image is in viewport
    }
}, false);
```

To check if our image is in the viewport, we'll use the `isInViewport()` helper method I shared earlier in this guide.

```
// Get all lazy load images
var images = document.querySelectorAll('.lazy-load');

// Determine if an element is in the viewport
var isInViewport = function ( elem ) {
    var distance = elem.getBoundingClientRect();
    return (
        distance.top >= 0 &&
        distance.left >= 0 &&
        distance.bottom <= (window.innerHeight || documen
t.documentElement.clientHeight) &&
        distance.right <= (window.innerWidth || document.
documentElement.clientWidth)
    );
```

```
};

// Listen for scroll events
window.addEventListener('scroll', function (event) {

    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++) {

        // Check if the image is in the viewport
        if (isInViewport(images[i])) {
            // Load the image
            console.log(images[i]);
        }

    }

}, false);
```

If you reload the page, open up the console in dev tools, and scroll, you'll see our image placeholders showing up as the come into the viewport.

Loading the image

If the image is in the viewport, we want to load it into the DOM. First, though, let's make sure our image has a `[data-image]` attribute on it. If not, we can just quit and move on to the next image in our loop.

```
// Listen for scroll events
window.addEventListener('scroll', function (event) {

    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++) {

        // Check if the image is in the viewport
        if (isInViewport(images[i])) {

            // Make sure the image has a data-image attri
bute
            if (!images[i].hasAttribute('data-image'))) c
ontinue;

        }

    }

}, false);
```

To load the image, we can use `innerHTML` to inject it into the placeholder `<figure>` element.

```
// Listen for scroll events
window.addEventListener('scroll', function (event) {

    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++) {

        // Check if the image is in the viewport
        if (isInViewport(images[i])) {

            // Make sure the image has a data-image attri
bute
            if (!images[i].hasAttribute('data-image')) co
ntinue;

            // Load the image
            images[i].innerHTML = '<img src="' + images[i
].getAttribute('data-image') + '">';

        }

    }

}, false);
```

Loading the caption

Next, let's check if there's a caption included, and load that, too.

We'll use `hasAttribute()` to see if a caption is included. If it is, we'll climb down the DOM to find our image. Then, we'll create a `<caption>` element and and use `insertBefore` to add it after the image.

Note: *This is not the most efficient way to do this. A better way would be to add the image and caption at the same time using* `innerHTML`*. But, I wanted to make sure you got to work with as many of the techniques covered in this guide as possible.*

```
// Listen for scroll events
window.addEventListener('scroll', function (event) {

    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++) {

        // Check if the image is in the viewport
        if (isInViewport(images[i])) {

            // Make sure the image has a data-image attri
bute
            if (!images[i].hasAttribute('data-image')) co
ntinue;

            // Load the image
            images[i].innerHTML = '<img src="' + images[i
].getAttribute('data-image') + '">';

            // Add a caption if one exists
            if (images[i].hasAttribute('data-caption')) {
                var img = images[i].querySelector('img');
```

```
                var caption = document.createElement('cap
tion');
                caption.innerHTML = images[i].getAttribut
e('data-caption');
                images[i].insertBefore(caption, img.nextS
ibling);
            }

        }

    }

}, false);
```

Only load each image once

There's a small problem with our code. Right now, it reloads the image and caption over and over again as you scroll through a page.

You can see this in action by logging our image in the console.

```
// Listen for scroll events
window.addEventListener('scroll', function (event) {

    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++) {

        // Check if the image is in the viewport
```

```
        if (isInViewport(images[i])) {

            // Make sure the image has a data-image attri
bute
            if (!images[i].hasAttribute('data-image')) co
ntinue;

            // Load the image
            images[i].innerHTML = '<img src="' + images[i
].getAttribute('data-image') + '">';

            // Add a caption if one exists
            if (images[i].hasAttribute('data-caption')) {
                var img = images[i].querySelector('img');
                var caption = document.createElement('cap
tion');
                caption.innerHTML = images[i].getAttribut
e('data-caption');
                images[i].insertBefore(caption, img.nextS
ibling);
            }

            // Log the image load in the console
            console.log(images[i]);

        }

    }
```

```
}, false);
```

See how the same images are showing up multiple times? We don't want that. Let's add a `.loaded` class to our placeholder after we load the image.

```
// Listen for scroll events
window.addEventListener('scroll', function (event) {

    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++) {

        // Check if the image is in the viewport
        if (isInViewport(images[i])) {

            // Make sure the image has a data-image attri
bute
            if (!images[i].hasAttribute('data-image')) co
ntinue;

            // Load the image
            images[i].innerHTML = '<img src="' + images[i
].getAttribute('data-image') + '">';

            // Add a caption if one exists
            if (images[i].hasAttribute('data-caption')) {
                var img = images[i].querySelector('img');
                var caption = document.createElement('cap
tion');
                caption.innerHTML = images[i].getAttribut
```

```
e('data-caption');
                images[i].insertBefore(caption, img.nextS
ibling);
            }

            // Add a .loaded class to our image placehold
er
            images[i].classList.add('loaded');

        }

    }

}, false);
```

Now, where we check to make sure our placeholder has a `[data-image]` attribute, we can also check to see if it has the `.loaded` class.

```
// Listen for scroll events
window.addEventListener('scroll', function (event) {

    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++) {

        // Check if the image is in the viewport
        if (isInViewport(images[i])) {

            // Make sure the image has a data-image attri
```

```
bute and hasn't already been loaded
            if (!images[i].hasAttribute('data-image') ||
images[i].classList.contains('loaded')) continue;

            // Load the image
            images[i].innerHTML = '<img src="' + images[i
].getAttribute('data-image') + '">';

            // Add a caption if one exists
            if (images[i].hasAttribute('data-caption')) {
                var img = images[i].querySelector('img');
                var caption = document.createElement('cap
tion');
                caption.innerHTML = images[i].getAttribut
e('data-caption');
                images[i].insertBefore(caption, img.nextS
ibling);
            }

            // Add a .loaded class to our image placehold
er
            images[i].classList.add('loaded');

        }

    }

}, false);
```

Loading images on page load

One last thing.

If an image is above the fold on page load, but the visitor hasn't scrolled yet, it won't load. We want to run the same loop we use on scroll when the page is first loaded.

First, let's pull it out into its own function so that we're not writing the same code twice.

```
// Load our images
var loadImages = function () {

    // Loop through each lazy load image
    for (var i = 0; i < images.length; i++) {

        // Check if the image is in the viewport
        if (isInViewport(images[i])) {

            // Make sure the image has a data-image attri
bute and hasn't already been loaded
            if (!images[i].hasAttribute('data-image') ||
images[i].classList.contains('loaded')) continue;

            // Load the image
            images[i].innerHTML = '<img src="' + images[i
].getAttribute('data-image') + '">';

            // Add a caption if one exists
```

```
            if (images[i].hasAttribute('data-caption')) {
                var img = images[i].querySelector('img');
                var caption = document.createElement('cap
tion');
                caption.innerHTML = images[i].getAttribut
e('data-caption');
                images[i].insertBefore(caption, img.nextS
ibling);
            }

            // Add a .loaded class to our image placehold
er
            images[i].classList.add('loaded');

        }

    }

};
```

Then, we can call that function in our event listener.

```
// Listen for scroll events
window.addEventListener('scroll', loadImages, false);
```

Now, let's also call that function the first time our script runs.

```
// Listen for scroll events
window.addEventListener('scroll', loadImages, false);

// Load images on page load
loadImages();
```

Now, any images in the viewport on page load will show up instantly. The rest will load when they're scrolled into the viewport.

Congratulations! You just created an image lazy loader using a handful of DOM injection and traversal techniques.

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. http://quirksmode.org/dom/html/↩
2. http://quirksmode.org/dom/html/↩
3. https://github.com/cferdinandi/dom-injection-source-code/↩