

[Technologies ▾](#)[References & Guides ▾](#)[Feedback ▾](#)[Sign in !\[\]\(d66ff64371a51729ac8c1cdaa685ba6f\_img.jpg\)](#)

# Manipulating documents

[← Previous](#)[↑ Overview: Client-side web APIs](#)[Next →](#)

When writing web pages and apps, one of the most common things you'll want to do is manipulate the document structure in some way. This is usually done by using the Document Object Model (DOM), a set of APIs for controlling HTML and styling information that makes heavy use of the `Document` object. In this article we'll look at how to use the DOM in detail, along with some other interesting APIs that can alter your environment in interesting ways.

**Prerequisites:** Basic computer literacy, a basic understanding of HTML, CSS, and JavaScript — including JavaScript objects.

**Objective:** To gain familiarity with the core DOM APIs, and the other APIs commonly associated with DOM and document manipulation

# The important parts of a web browser

Web browsers are very complicated pieces of software with a lot of moving parts, many of which can't be controlled or manipulated by a web developer using JavaScript. You might think that such limitations are a bad thing, but browsers are locked down for good reasons, mostly centering around security. Imagine if a web site could get access to your stored passwords or other sensitive information, and log into websites as if it were you?

Despite the limitations, Web APIs still give us access to a lot of functionality that enable us to do a great many things with web pages. There are a few really obvious bits you'll reference regularly in your code — consider the following diagram, which represents the main parts of a browser directly involved in viewing web pages:



- The window is the browser tab that a web page is loaded into; this is represented in JavaScript by the `Window` object. Using methods available on this object you can do things like return the window's size (see `Window.innerWidth` and `Window.innerHeight`), manipulate the document loaded into that window, store data specific to that document on the client-side (for example using a local database or other storage mechanism), attach an event handler to the current window, and more.
- The navigator represents the state and identity of the browser (i.e. the user-agent) as it exists on the web. In JavaScript, this is represented by the `Navigator` object. You

can use this object to retrieve things like geolocation information, the user's preferred language, a media stream from the user's webcam, etc.

- The document (represented by the DOM in browsers) is the actual page loaded into the window, and is represented in JavaScript by the `Document` object. You can use this object to return and manipulate information on the HTML and CSS that comprises the document, for example get a reference to an element in the DOM, change its text content, apply new styles to it, create new elements and add them to the current element as children, or even delete it altogether.

In this article we'll focus mostly on manipulating the document, but we'll show a few other useful bits besides.

---

## The document object model

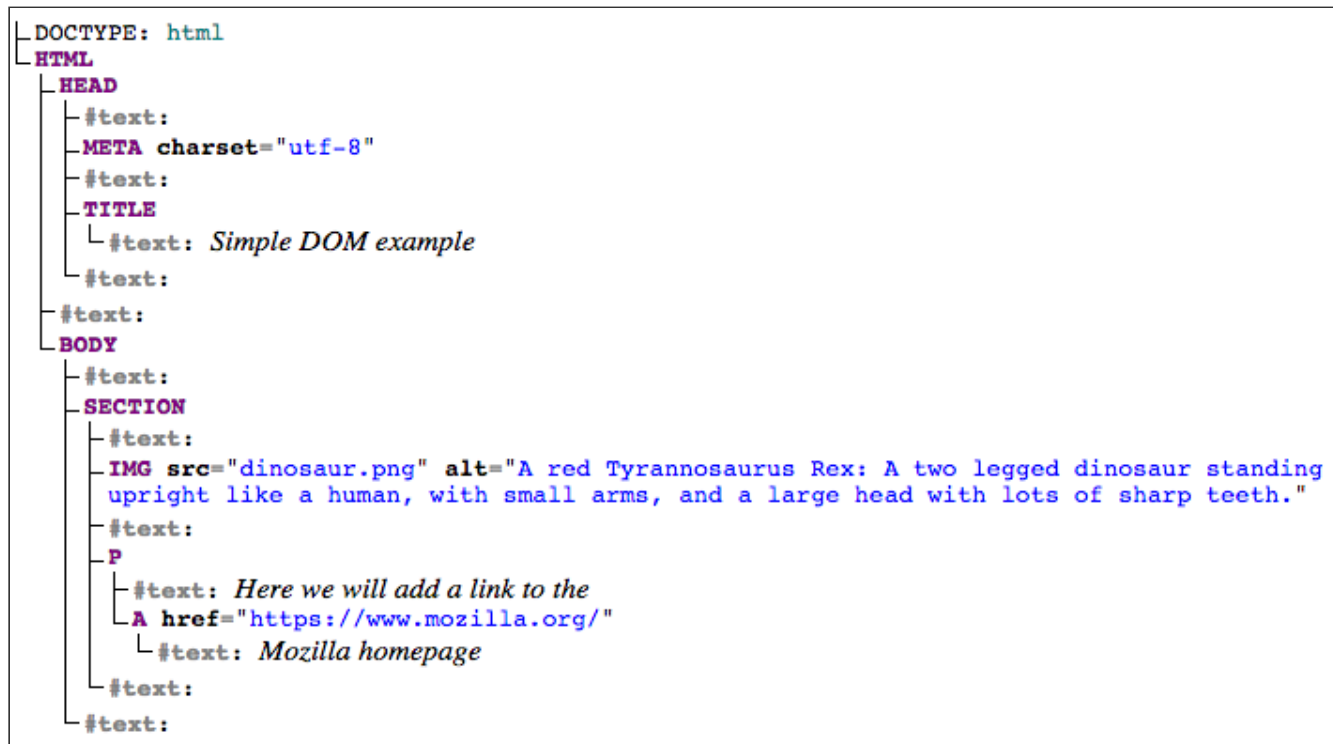
The document currently loaded in each one of your browser tabs is represented by a document object model. This is a "tree structure" representation created by the browser that enables the HTML structure to be easily accessed by programming languages — for example the browser itself uses it to apply styling and other information to the correct elements as it renders a page, and developers like you can manipulate the DOM with JavaScript after the page has been rendered.


We have created a simple example page at [dom-example.html](#) ([see it live also](#)). Try opening this up in your browser — it is a very simple page containing a `<section>` element inside which you can find an image, and a paragraph with a link inside. The HTML source code looks like this:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8">
5      <title>Simple DOM example</title>
6    </head>
7    <body>
8      <section>
9        Here we will add a link to the <a href="https://www.mozilla.org
11       </section>
12
```

```
13 | </body>
    | </html>
```

The DOM on the other hand looks like this:



 **Note:** This DOM tree diagram was created using Ian Hickson's [Live DOM viewer](#).

You can see here that each element and bit of text in the document has its own entry in the tree — each one is called a **node**. You will also encounter various terms used to describe the type of node, and their position in the tree in relation to one another:

- **Element node:** An element, as it exists in the DOM.
- **Root node:** The top node in the tree, which in the case of HTML is always the HTML node (other markup vocabularies like SVG and custom XML will have different root elements).
- **Child node:** A node *directly* inside another node. For example, IMG is a child of SECTION in the above example.
- **Descendant node:** A node *anywhere* inside another node. For example, IMG is a child of SECTION in the above example, and it is also a descendant. IMG is not a child of BODY, as it is two levels below it in the tree, but it is a descendant of BODY.
- **Parent node:** A node which has another node inside it. For example, BODY is the parent node of SECTION in the above example.

- **Sibling nodes:** Nodes that sit on the same level in the DOM tree. For example, `IMG` and `P` are siblings in the above example.
- **Text node:** A node containing a text string.

It is useful to familiarize yourself with this terminology before working with the DOM, as a number of the code terms you'll come across make use of them. You may have also come across them if you have studied CSS (e.g. descendant selector, child selector).

---

## Active learning: Basic DOM manipulation

To start learning about DOM manipulation, let's begin with a practical example.

1. Take a local copy of the [dom-example.html](#) page and the [image](#) that goes along with it.
2. Add a `<script></script>` element just above the closing `</body>` tag.
3. To manipulate an element inside the DOM, you first need to select it and store a reference to it inside a variable. Inside your script element, add the following line:

```
1 | var link = document.querySelector('a');
```

4. Now we have the element reference stored in a variable, we can start to manipulate it using properties and methods available to it (these are defined on interfaces like `HTMLAnchorElement` in the case of `<a>` element, its more general parent interface `HTMLElement`, and `Node` — which represents all nodes in a DOM). First of all, let's change the text inside the link by updating the value of the `Node.textContent` property. Add the following line below the previous one:

```
1 | link.textContent = 'Mozilla Developer Network';
```

5. We should also change the URL the link is pointing to, so that it doesn't go to the wrong place when it is clicked on. Add the following line, again at the bottom:

```
1 | link.href = 'https://developer.mozilla.org';
```

Note that, as with many things in JavaScript, there are many ways to select an element and store a reference to it in a variable. `Document.querySelector()` is the recommended modern approach, which is convenient because it allows you to select elements using CSS selectors. The above `querySelector()` call will match the first `<a>` element that appears

in the document. If you wanted to match and do things to multiple elements, you could use `Document.querySelectorAll()`, which matches every element in the document that matches the selector, and stores references to them in an array-like object called a `NodeList`.

There are older methods available for grabbing element references, such as:

- `Document.getElementById()`, which selects an element with a given `id` attribute value, e.g. `<p id="myId">My paragraph</p>`. The ID is passed to the function as a parameter, i.e. `var elementRef = document.getElementById('myId')`.
- `Document.getElementsByTagName()`, which returns an array containing all the elements on the page of a given type, for example `<p>s`, `<a>s`, etc. The element type is passed to the function as a parameter, i.e. `var elementRefArray = document.getElementsByTagName('p')`.

These two work in older browsers than the modern methods like `querySelector()`, but are not as convenient. Have a look and see what others you can find!

## Creating and placing new nodes

The above has given you a little taste of what you can do, but let's go further and look at how we can create new elements.

1. Going back to the current example, let's start by grabbing a reference to the our `<section>` element — add the following code at the bottom of your existing script (do the same with the other lines too):

```
1 | var sect = document.querySelector('section');
```

2. Now let's create a new paragraph using `Document.createElement()` and give it some text content in the same way as before:

```
1 | var para = document.createElement('p');  
2 | para.textContent = 'We hope you enjoyed the ride.';
```

3. You can now append the new paragraph at the end of the section using `Node.appendChild()`:

```
1 | sect.appendChild(para);
```

4. Finally for this part, let's add a text node to the paragraph the link sits inside, to round off the sentence nicely. First we will create the text node using

```
Document.createTextNode():
```

```
1 | var text = document.createTextNode(' - the premier source for web
```

5. Now we'll grab a reference to the paragraph the link is inside, and append the text node to it:

```
1 | var linkPara = document.querySelector('p');  
2 | linkPara.appendChild(text);
```

That's most of what you need for adding nodes to the DOM — you'll make a lot of use of these methods when building dynamic interfaces (we'll look at some examples later).

## Moving and removing elements

There may be times when you want to move nodes, or delete them from the DOM altogether. This is perfectly possible.

If we wanted to move the paragraph with the link inside it to the bottom of the section, we could simply do this:

```
1 | sect.appendChild(linkPara);
```

This moves the paragraph down to the bottom of the section. You might have thought it would make a second copy of it, but this is not the case — `linkPara` is a reference to the one and only copy of that paragraph. If you wanted to make a copy and add that as well, you'd need to use `Node.cloneNode()` instead.

Removing a node is pretty simple as well, at least when you have a reference to the node to be removed and its parent. In our current case, we just use `Node.removeChild()`, like this:

```
1 | sect.removeChild(linkPara);
```

It gets slightly more complex when you want to remove a node based only on a reference to itself, which is fairly common. There is no method to tell a node to remove itself, so you'd have to do the following.

```
1 | linkPara.parentNode.removeChild(linkPara);
```

Have a go at adding the above lines to your code.

## Manipulating styles

It is possible to manipulate CSS styles via JavaScript in a variety of ways.

To start with, you can get a list of all the stylesheets attached to a document using `Document.styleSheets`, which returns an array of `CSSStyleSheet` objects. You can then add/remove styles as wished. However, we're not going to expand on those features because they are a somewhat archaic and difficult way to manipulate style. There are much easier ways.

The first way is to add inline styles directly onto elements you want to dynamically style. This is done with the `HTMLElement.style` property, which contains inline styling information for each element in the document. You can set properties of this object to directly update element styles.

1. As an example, try adding these lines to our ongoing example:

```
1 | para.style.color = 'white';  
2 | para.style.backgroundColor = 'black';  
3 | para.style.padding = '10px';  
4 | para.style.width = '250px';  
5 | para.style.textAlign = 'center';
```

2. Reload the page and you'll see that the styles have been applied to the paragraph. If you look at that paragraph in your browser's Page Inspector/DOM inspector, you'll see that these lines are indeed adding inline styles to the document:

```
1 | <p style="color: white; background-color: black; padding: 10px; v
```

**Note:** Notice how the JavaScript property versions of the CSS styles are written in lower camel case whereas the CSS versions are hyphenated (e.g. `backgroundColor` versus `background-color`). Make sure you don't get these mixed up, otherwise it won't work.

There is another common way to dynamically manipulate styles on your document, which we'll look at now.

1. Delete the previous five lines you added to the JavaScript.



2. Add the following inside your HTML `<head>`:

```
1 | <style>
2 | .highlight {
3 |   color: white;
4 |   background-color: black;
5 |   padding: 10px;
6 |   width: 250px;
7 |   text-align: center;
8 | }
9 | </style>
```

3. Now we'll turn to a very useful method for general HTML manipulation — `Element.setAttribute()` — this takes two arguments, the attribute you want to set on the element, and the value you want to set it to. In this case we will set a class name of `highlight` on our paragraph:

```
1 | para.setAttribute('class', 'highlight');
```

4. Refresh your page, and you'll see no change — the CSS is still applied to the paragraph, but this time by giving it a class that is selected by our CSS rule, not as inline CSS styles.

Which method you choose is up to you; both have their advantages and disadvantages. The first method takes less setup and is good for simple uses, whereas the second method is more purist (no mixing CSS and JavaScript, no inline styles, which are seen as a bad practice). As you start building larger and more involved apps, you will probably start using the second method more, but it is really up to you.

At this point, we haven't really done anything useful! There is no point using JavaScript to create static content — you might as well just write it into your HTML and not use JavaScript. It is more complex than HTML, and creating your content with JavaScript also has other issues attached to it (such as not being readable by search engines).

In the next couple of sections we will look at a couple of more practical uses of DOM APIs.

 **Note:** You can find our [finished version](#) of the `dom-example.html` demo on GitHub ([see it live also](#)).

# Active learning: Getting useful information from the Window object

So far we've only really looked at using `Node` and `Document` features to manipulate documents, but there is no reason why you can't get data from other sources and use it in your UI. Think back to our simple [maps-example.html](#) demo from the last article — there we retrieved some location data and used it to display a map of your area. You just have to make sure your data is in the right format; JavaScript makes it easier than many other languages, being weakly typed — for example numbers will convert to strings automatically when you want to print them to the screen.

In this example we will solve a common problem — making sure your application is as big as the window it is viewed in, whatever size it is. This is often useful in situations like games, where you want to use as much of the screen area as possible to play the game in.

To start with, make a local copy of our [window-resize-example.html](#) and [bgtile.png](#) demo files. Open it and have a look — you'll see that we've got a `<div>` element covering a small part of the screen, which has got a background tile applied to it. We'll use that to represent our app UI area.

1. First of all, let's grab a reference to the div, and then grab the width and height of the viewport (the inner window, where your document is displayed) and store them in variables — these two values are handily contained in the `Window.innerWidth` and `Window.innerHeight` properties. Add the following lines inside the existing `<script>` element:

```
1 | var div = document.querySelector('div');
2 | var WIDTH = window.innerWidth;
3 | var HEIGHT = window.innerHeight;
```

2. Next, we'll dynamically alter the width and height of the div to equal that of the viewport. Add the following two lines below your first ones:

```
1 | div.style.width = WIDTH + 'px';
2 | div.style.height = HEIGHT + 'px';
```

3. Save and try refreshing your browser — you should now see the div become as big as your viewport, whatever size of screen you are using. If you now try resizing your window to make it bigger, you'll see that the div stays the same size — we are only setting it once.

4. How about we use an event so that the div resizes as we resize the window? The `Window` object has an event available on it called `resize`, which is fired every time the window is resized — let's access that via the `Window.onresize` event handler and rerun our sizing code each time it changes. Add the following to the bottom of your code:

```
1 | window.onresize = function() {  
2 |     WIDTH = window.innerWidth;  
3 |     HEIGHT = window.innerHeight;  
4 |     div.style.width = WIDTH + 'px';  
5 |     div.style.height = HEIGHT + 'px';  
6 | }
```

 **Note:** If you get stuck, have a look at our [finished window resize example](#) ([see it live](#) also).

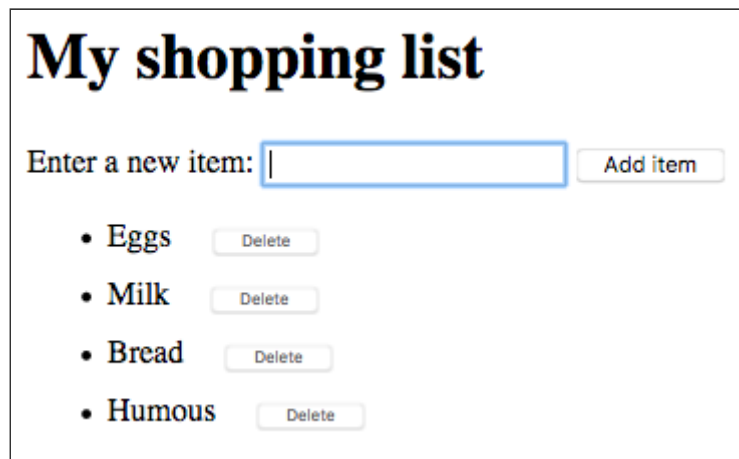
---

## Active learning: A dynamic shopping list

To round off the article, we'd like to set you a little challenge — we want to make a simple shopping list example that allows you to dynamically add items to the list using a form input and button. When you add an item to the input and press the button:

- The item should appear in the list.
- Each item should be given a button that can be pressed to delete that item off the list.
- The input should be emptied and focused ready for you to enter another item.

The finished demo will look something like this:



**My shopping list**

Enter a new item:  Add item

- Eggs Delete
- Milk Delete
- Bread Delete
- Humous Delete

To complete the exercise, follow the steps below, and make sure that the list behaves as described above.

1. To start with, download a copy of our [shopping-list.html](#) starting file and make a copy of it somewhere. You'll see that it has some minimal CSS, a list with a label, input, and button, and an empty list and `<script>` element. You'll be making all your additions inside the script.
2. Create three variables that hold references to the list (`<ul>`), `<input>`, and `<button>` elements.
3. Create a function that will run in response to the button being clicked.
4. Inside the function body, start off by storing the current value of the input element in a variable.
5. Next, empty the input element by setting its value to an empty string — `''`.
6. Create three new elements — a list item (`<li>`), `<span>`, and `<button>`, and store them in variables.
7. Append the span and the button as children of the list item.
8. Set the text content of the span to the input element value you saved earlier, and the text content of the button to 'Delete'.
9. Append the list item as a child of the list.
10. Attach an event handler to the delete button, so that when clicked it will delete the entire list item it is inside.
11. Finally, use the `focus()` method to focus the input element ready for entering the next shopping list item.

 **Note:** If you get really stuck, have a look at our [finished shopping list](#) ([see it running live also](#).)

# Summary

We have reached the end of our study of document and DOM manipulation. At this point you should understand what the important parts of a web browser are with respect to controlling documents and other aspects of the user's web experience. Most importantly, you should understand what the Document Object Model is, and how to manipulate it to create useful functionality.

---

## See also

There are lots more features you can use to manipulate your documents. Check out some of our references and see what you can discover:

- [Document](#)
- [Window](#)
- [Node](#)
- [HTMLElement](#), [HTMLInputElement](#), [HTMLImageElement](#), etc.

(See our [Web API index](#) for the full list of Web APIs documented on MDN!)

[← Previous](#)[↑ Overview: Client-side web APIs](#)[Next →](#)

---

## In this module

- [Introduction to web APIs](#)
- [Manipulating documents](#)
- [Fetching data from the server](#)
- [Third party APIs](#)
- [Drawing graphics](#)
- [Video and audio APIs](#)
- [Client-side storage](#)

