Engineer

DOM

the live tree of objects a browser builds from a page

What it is

The DOM, short for Document Object Model, is the tree of objects a browser builds from a page and keeps in memory for as long as the page is open. The HTML parser reads the markup, and its output is a document object: every element, every run of text and every comment becomes a node, and each node knows its parent, its children and its neighbours. Scripts read and change the page through that tree, and what the browser draws is worked out from it.

The WHATWG maintains the model in the DOM Standard, whose own summary names three things it covers without tying them to any one platform: node trees, events, and the aborting of activities in progress. The name is older than the standard. In its original sense it meant an interface for reaching into a document and changing it; today most people also say “the DOM” for the tree itself, and both meanings are in daily use. The HTML Standard builds on top of it: HTML defines the elements and what they do, the DOM Standard defines what a node, an element, an attribute and an event are.

The tree is not the source file. A page’s HTML is text that arrived over the network. The DOM is what the parser made of that text, with every parse error handled by rules the HTML Standard spells out, and then changed by every script that has run since. The parser keeps details that are easy to forget. The whitespace between two tags can become a text node of its own, so the first child of an element is not always the first element inside it, although the standard’s own example notes that not every stretch of whitespace turns into a text node.

Nodes come in a handful of kinds. The document sits at the root, with a doctype, elements, text nodes and comments beneath it. A document fragment is a light container that sits outside the page’s tree, which makes it a handy place to

assemble a block before inserting it in one move. An element can also host a shadow tree, a separate tree of its own that the document's ordinary queries do not reach into.

Lists of nodes come in two kinds, and the difference matters. The standard's rule is that a collection is live unless stated otherwise: `childNodes` and the HTMLCollection from `getElementsByClassName()` change as the tree changes. `querySelectorAll()` is the stated exception and returns a static list, a snapshot taken at the moment of the call.

The same standard defines how events travel. An event is dispatched at a target node and passes through three phases: capturing, on the way down from the top of the tree through the target's ancestors; at the target itself; and bubbling, back up through the ancestors. The `eventPhase` property reports where it is, as 1, 2 or 3, and 0 when the event is not being dispatched. A listener added with `addEventListener()` takes options: `once` removes it after its first call, `passive` promises that it will not call `preventDefault()`, and `signal` ties it to an `AbortController`, so that aborting the controller removes the listener.

Use the concept in a decision

Put "What is the DOM?" to work in a real decision. We can help relate the concept to the system you are building or buying. Interfaces that load quickly, work on a phone and behave the same in every browser your customers happen to use.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Directed full-stack GIS map development — full-stack mapping, from database to browser

Directed full-stack GIS map development Get in touch →

Why it matters

Every script that changes a page works through the DOM, whatever language or framework it was written in. A framework adds a layer of its own on top, but what the reader finally sees is nodes in this tree. Next.js describes hydration as React attaching event handlers to the DOM, which is how HTML sent from the server becomes a page that responds to clicks. A team that knows the tree underneath can debug what the framework actually did, rather than what it was asked to do.

The tree is also where text and markup part ways. A string assigned to `textContent` replaces an element's children with a single text node and is never read as markup. The same string assigned to `innerHTML` is handed to the HTML parser, and the HTML Standard notes that those setters remove nothing dangerous by default. Which of the two a page uses decides whether a visitor's display name is shown as text or run as code.

What a reader experiences depends on more than the tree. Styles decide whether a node is visible, where it sits and whether a pointer can reach it, and none of that is visible to a tool that only walks the nodes. That gap is where automated checks are most often wrong, and it is the reason to test a page as rendered as well as the document behind it.

Finally, the DOM is the contract between a page and everything that reads it after the server has finished. Tests that click through a page, browser extensions and assistive software all start from the live tree. The other side of that contract is easy to miss: a crawler that does not run JavaScript sees only the HTML that was sent, and none of the nodes that scripts add later.

What it looks like in practice

The script below, written for this article, fills a list of orders from data, keeps a single click listener on the list rather than one per item, and can remove that listener later with one call. `openOrder` stands for whatever the page does when an order is chosen.

```
const list = document.querySelector("#orders");
const controller = new AbortController();

function render(orders) {
  const items = orders.map((order) => {
    const li = document.createElement("li");
    li.dataset.id = order.id;
    li.textContent = `${order.number}: ${order.status}`;
    return li;
  });
  list.replaceChildren(...items);
}

list.addEventListener(
  "click",
  (event) => {
    const li = event.target.closest("li");
```

```
    if (li) openOrder(li.dataset.id);
  },
  { signal: controller.signal },
);

// When the view goes away:
controller.abort();
```

Each part maps onto the standard. `createElement()` makes an element that belongs to no parent until it is inserted. `textContent` puts the order number in as text, so a status that happens to contain angle brackets stays a status. `replaceChildren()` removes the old items and inserts the new ones in a single operation. The one listener on the list serves every item because a click on an item bubbles up to the list, and `closest()` walks from whatever was clicked to the nearest `li` . Aborting the controller removes the listener without the page having to keep a reference to the function.

The tree also explains a class of defects that this site's own review turned up. In an accessibility review carried out in August 2026, the site passed its automated rules and still had eleven defects, every one of them down to how the page rendered rather than to what its markup said. Eight footer links stayed in the tab order behind `pointer-events` , a property that removes pointer interaction and nothing else. To anything reading the document they were ordinary links; a pointer never reached them, and the keyboard still stopped on every one. A scroll-driven animation clipped the footer off three pages. Of the four rules the review left behind, one reads as a warning against trusting the tree alone: visibility has to be established from the rendered pixels, not merely from the document tree.

Where it goes wrong

A loop over a live collection skips every other item. The HTMLCollection from `getElementsByClassName()` is live, so a loop that removes the class from each element by index shrinks the collection under itself and steps over every second element. Iterate a static list instead: `querySelectorAll()` returns one, and `Array.from()` turns a live collection into a plain array first.

```
const pending = document.getElementsByClassName("pending");
for (let i = 0; i < pending.length; i++) {
  pending[i].classList.remove("pending"); // skips every other one
}
```

```
for (const el of document.querySelectorAll(".pending")) {
  el.classList.remove("pending"); // static list: all of them
}
```

Listeners pile up. A view that adds its listeners every time it renders, and never removes them, runs each handler once per render: after three visits, one click sends three requests. Removing a listener by hand takes the very same function object that was added, and an arrow function written inline at the call leaves nothing to pass back. Pass a `signal` from one `AbortController` per view, as in the example above, and a single `abort()` removes every listener the view added; use `once: true` for a listener that should fire only one time.

Text goes in as markup. Building a string of HTML from data and assigning it to `innerHTML` gives the data to the parser, with no sanitizing by default. A display name that carries markup, an image tag with an error handler for instance, arrives on the page as a real element with its handler attached. Text belongs in `textContent` or in a text node made with `createTextNode()`. Where markup really must come from a string, the HTML Standard defines `setHTML()`, which sanitizes as it parses and removes unsafe content.

An observer feeds itself. A `MutationObserver` reports changes to the nodes it watches, and its callback runs after each batch of changes. A callback that writes to the subtree it observes, to add a counter or correct a class, creates new records and is called again, and the page can lock up in a loop of its own making. Watch only what has to be watched, with `attributeFilter` and without `subtree` where possible, have the callback check whether its change is already in place before writing, or call `disconnect()` before the write and `observe()` again after it.

Further reading

- DOM Standard — the WHATWG's definition of nodes, trees, live and static collections, events, `AbortController` and `MutationObserver`.
- Parsing HTML documents — how the HTML parser turns a stream of characters into a document object, parse errors included.
- Dynamic markup insertion — `innerHTML`, `setHTML()` and `setHTMLUnsafe()`, and which of them sanitize.
- Server and Client Components — the Next.js account of hydration: HTML from the server, then event handlers attached to the DOM.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is the DOM?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/dom/