



Document Object Model

Frontend Junior Program - 2022

Solutions that might fix the problem without breaking anything



Essential

Hoping This Works

ORLY?

@ThePracticalDev

Agenda

- 1 Intro
- 2 History – why do we need to manipulate the DOM?
- 3 Document Object Model
- 4 Accessing the DOM
- 5 Modifying the DOM structure

Document Object Model (DOM)

The Document Object Model (DOM) is a representation — a model — of a document and its content.

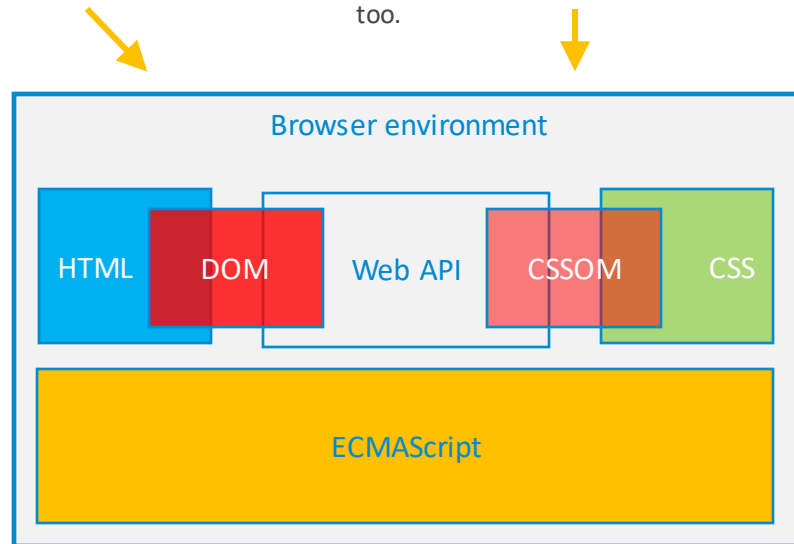
It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects.

DOM is a web [standard](#).

But why do we need to access the HTML markup in the first place? To understand that, we need a bit of a history lesson...

It is 2021 - you probably won't work with the [Document Object Model](#) directly, yet it is something you need to know.

Also, we have a [CSS Object Model](#), for manipulating CSS from JavaScript. We deliberately won't touch this, as you should not touch this in production code, too.

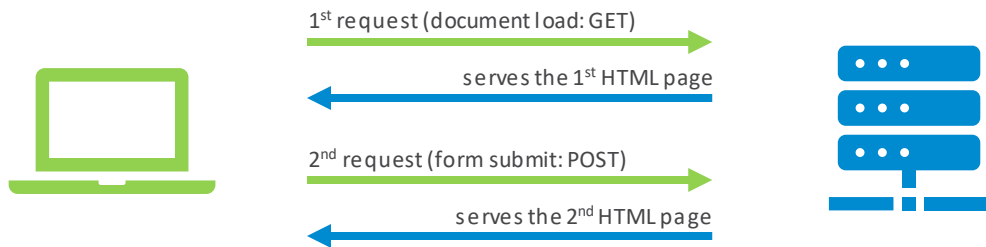


Static HTML pages

Traditionally, webserver served **static HTML pages**

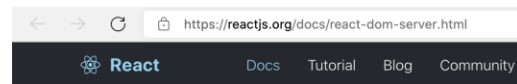
For a page, such as Wikipedia, this makes sense. The main interaction from user side is to read and to navigate between the pages via **links**.

Every interaction results in a complete, new HTML document.



Submitting a form also possible, simply with using the **<form> html element**: clicking on the submit button will result in a completely new HTML document, too.

*These documents are simple HTML text files on the server. Even the folder structure on the server is reflected in the document's URL: **docs** could be a real folder in the document root.*



sure, there could be **magic** as well...

Server-Side Scripting

Editing, however, could be a bit of cumbersome in the case of a complex site, based on hundreds of HTML pages

Basically, there are 2 approaches for this problem:

The pages can be [pre-generated](#) beforehand with a [static site generator](#) (the pages must be uploaded to the server then) or on the fly, on the server directly, when a request arrives from the client. This is [server-side scripting](#).

Different technologies can be used for this, such as [php](#), [.net](#), [jsp](#) and even [react](#). These scripts / applications can be created by the side builder team, or, alternatively, a CMS ([Content Management System](#)) can be used.

Many times, these technologies are combined.

From client side (HTML perspective), however, a server-side rendered page is exactly the same as a static page: [every interaction results in a complete, new HTML document](#).

Don't think, that this is the past. A majority of the big sites are generated on the server side.



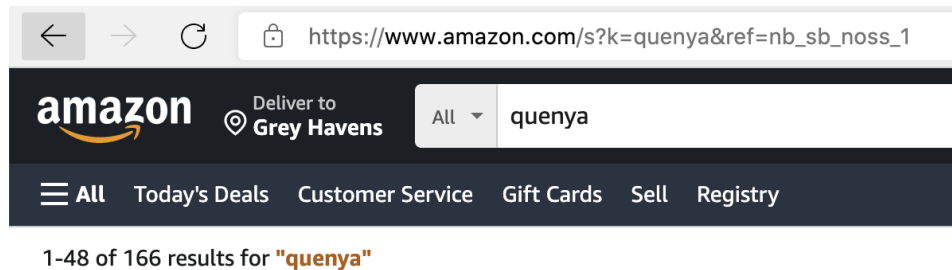
*"But in the end it's only a passing thing, this shadow;
even darkness must pass."*

Server-Side Scripting

From [user perspective](#), however, there is a huge difference!

Rendering pages on the fly opened a new horizon for interactivity.

These URL parameters enables the server to deliver pages which can react to users' interaction: here, this is a [search result page](#)



The life without JavaScript

The image shows a screenshot of an Amazon product page for the book "Quetin i lambë Eldaiva: Cours de Quenya (French) Paperback" by Thorsten Renk. The page displays the book's title, author, and a price of £14.00. It also shows delivery options, including free delivery by Thursday, April 8, and fastest delivery by Tuesday, April 6. The book is in stock and can be added to the basket or bought now. Below the product page, a browser settings menu is open, showing the "Preferences" section. The "Debugger" section is expanded, and the "Disable JavaScript" checkbox is checked. A yellow arrow points from the text "Try it out: you can order from Amazon without JavaScript." to the "Disable JavaScript" checkbox.

Quetin i lambë Eldaiva: Cours de Quenya (French) Paperback – 30 Mar. 2015
by Thorsten Renk (Author), Ambar Eldaron (Author)
★★★★☆ 4 ratings

> See all formats and editions

Paperback
£14.00

1 Used from £12.54
1 New from £14.00

Note: This item is eligible for **FREE click and collect** without a minimum order. [Details](#)
Le fameux cours de Quenya de Thorsten Renk

Print length Language

£14.00

FREE delivery: Thursday, April 8
[Details](#)
FREE Delivery on book orders
dispatched by Amazon over £10.00 .

Fastest delivery: Tuesday, April 6
Order within 12 hrs 49 mins [Details](#)

Select delivery location

In stock.

Dispatched from and sold by Amazon.

Quantity: 1

Add to Basket

Buy Now

Settings

Preferences

Workspace

Experiments

Library Code

Debugger

☒ Disable JavaScript

☐ Disable async stack traces

Try it out: you can order from Amazon **without JavaScript.**

To this point, there is **no need for JavaScript at all.**

For many sites, it is still requested that the main functionality **must work without JS.**

Without JavaScript, the only way to modify *anything** on the page is to reload the page entirely with all its assets.

And it costs a lot: the server bandwidth is expensive, and while you can cache / deliver the static assets (images) with Akamai / Cloudflare, you still need those for that, etc.

* well, we have iframes as well, if you asked

Documents vs Web Applications



Reading Wikipedia: lots of information, less interactivity

While even complex applications, such as Amazon can work without JavaScript, the **document-based approach** is not a natural way for interactivity.

Try to implement
this without ~~magic~~
JavaScript



Using Facebook: full immersion and interactivity.

That's why we need **DOM manipulation: for interactivity.**

The ~~world~~ web has changed: moved from documents to applications.

The only question: how to do it properly?

Yes, it was told by Treebeard and not Galadriel. Just think about it: why would Galadriel feel anything in the water and earth?

Try to access the source of the information: it may happen that it will actually make any sense then.



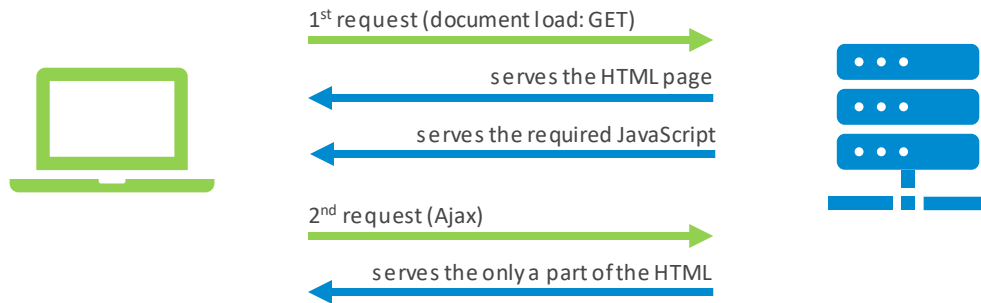
"For the world is changing: I feel it in the water, I feel it in the earth, and I smell it in the air."

Dynamic HTML and Ajax

With JavaScript, we can replace / reload any parts of the document

Changing the HTML markup in JavaScript is done by manipulating on the DOM (Document Object Model).

Basically, the DOM is the HTML representation in JavaScript. And with [the DOM](#) we can [do anything](#) with the markup. The architecture, however, now changes a bit:



Now we [need JavaScript](#) for adding the new / changed content to the page.

DOM: full access to the document

With the DOM, we can change anything in the entire tree

But is it a problem? - I hear you ask.

Let's say, that the branches of this tree is a component (or a function), and the leaves are variables. Now, just think about, in this system, you can access all of the variables in all of the components from any component.

Everything is globally accessible in this system.

Also, let me show you a real-world example of a DOM tree ...

DOM is a [tree](#).
(click, you can play with this)



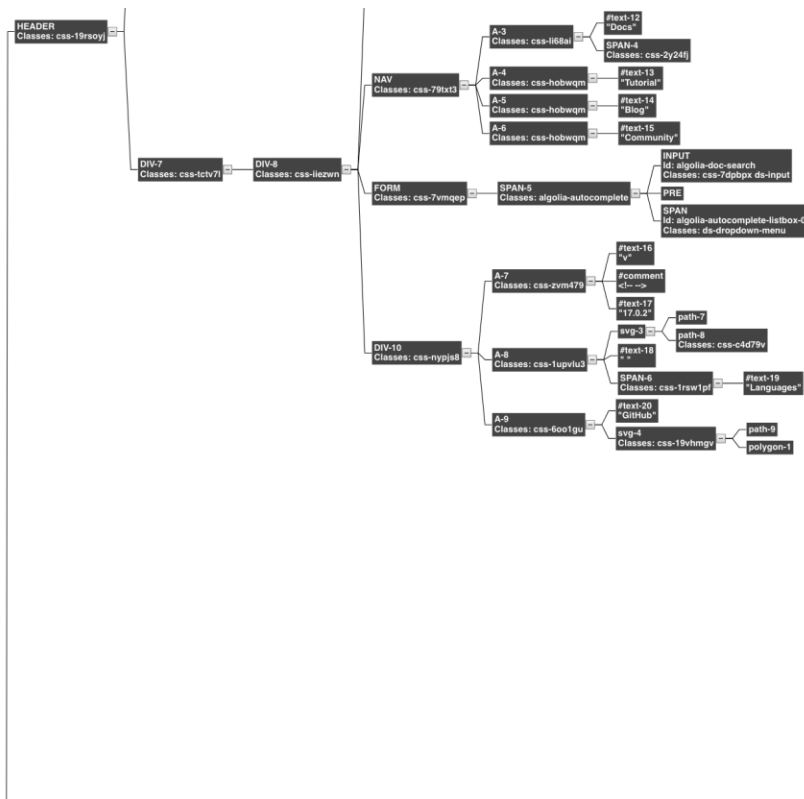
Markup to test ([permalink](#), [save](#), [upload](#), [download](#), [hide](#)):

```
<html>
<head>
  <title>With Treebeard and the Ents</title>
</head>
<body>
  <em>Legolas</em>
  <p>Then are we not to see the merry young hobbits again?</p>
  <em>Gandalf</em>
  <p>Who knows? Have patience. Go where you must go, and hope!</p>
</body>
</html>
```

DOM view ([hide](#), [refresh](#)):

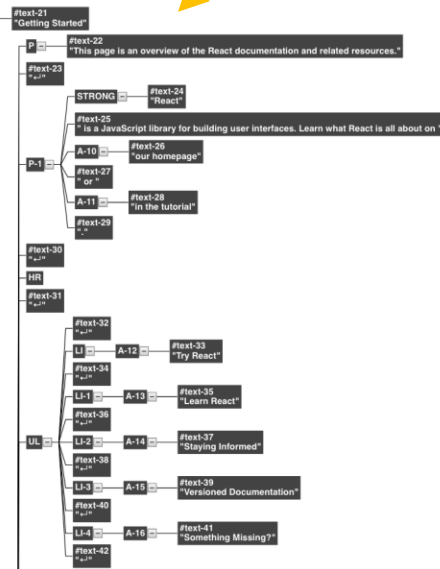
```
HTML
├── HEAD
│   ├── #text:
│   ├── TITLE
│   │   └── #text: With Treebeard and the Ents
│   └── #text:
├── #text:
├── BODY
│   ├── #text:
│   ├── EM
│   │   └── #text: Legolas
│   ├── #text:
│   ├── P
│   │   └── #text: Then are we not to see the merry young hobbits again?
│   ├── #text:
│   ├── EM
│   │   └── #text: Gandalf
│   ├── #text:
│   ├── P
│   │   └── #text: Who knows? Have patience. Go where you must go, and hope!
│   └── #text:
```

DOM: reactjs.org (part)

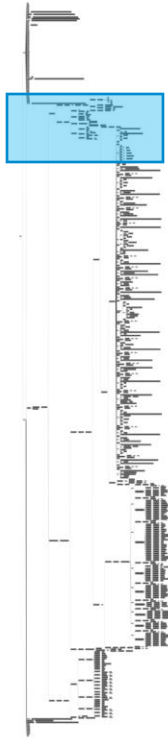


This is a [part of the DOM tree](#) of reactjs.org.

Looks nice and organised, doesn't?



DOM: reactjs.org (a full page)



← this was the part

← the whole tree, which is still very simple
- in reality, DOM trees are many times more complex.



"... when someone enters to the site, is it possible to show a banner on the top of the tower, please? But only if their last visit was 180 days ago and they will have a birthday in the next 30 days, and..."

jQuery

Direct DOM manipulation is not necessarily evil

We were able to manage to develop large-scale, scalable websites, and it was perfectly maintainable in long-term.

One way to achieve that is to break down the application into smaller parts (components), and raise boundaries between them.

Also, to prevent compatibility issues, we used [jQuery](#).

But why needed jQuery for this, instead of using the DOMAPI?

The main answer is: compatibility. There were times (looking at you, Internet Explorer), when it was not a trivial task to achieve browser compatibility.

jQuery provided an excellent solution for this problem.

Still, it was easy to do it wrong with the DOM, therefore new approaches has been emerged:

the [Single-Page Applications](#).



A software engineer (on the left), working on a website with Internet Explorer compatibility is required.

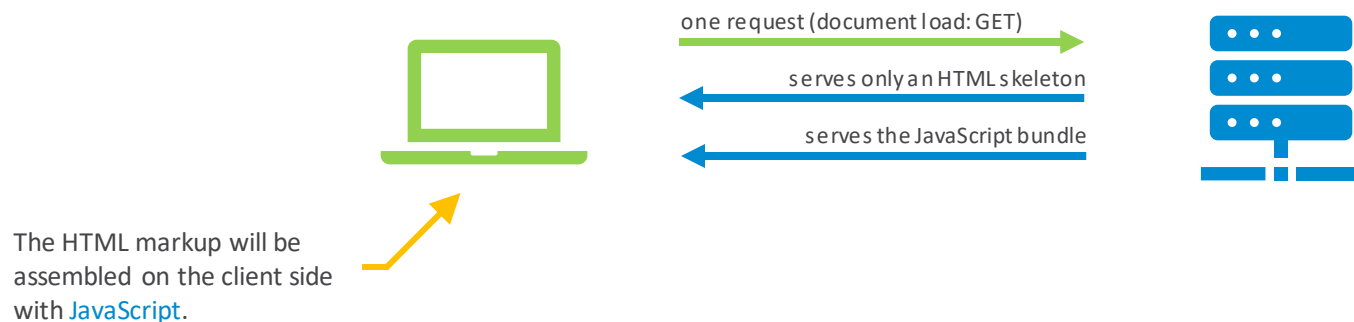
[“With complexity comes errors,”](#) and the DOM code base was a reliability liability. According to an internal investigation, from IE7 to IE11, approximately 28% of all IE reliability bugs originated from code in core DOM components.”

Single Page Applications

Modern SPAs (React and Angular, mainly) promise a scalable and simple solution for web development.

With these, however, the architecture has **completely changed**:

Of course, there are many more requests, however, for the document itself, it is only a single (hence the name) – and even that HTML is merely a skeleton, **the real HTML will be built in the browser**. Every page.



Single Page Applications

So modern SPAs **promise** a simple solution

Is that true?

Not entirely, it is just **differently complex** than jQuery.

SPAs **break down the application into components, based on the DOM tree**. This way, SPAs restrict accessing the DOM directly.

Is this a *better way* for web development? You can argue about that, still, there is a serious fundamental issue here:

the logical boundaries between parts of the application many times does not match with the DOM components.

You will realize this instantly, for example, when you find that you'd need to move too much data between too much components, and when you start to use workarounds to “solve” this problem.

Declarative

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

Component-Based

Build encapsulated components that manage their own state, then compose them to make complex UIs.

Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.



React marketing: *“painless, simple, efficient, declarative, predictable, easier, encapsulated, instead of templates, rich and keep out of DOM!” – yeah, sure.*



We need strict borders (interfaces) between components, that's for sure. The problem is, that many times it is not obvious where are these boundaries.

Do we need to extract this particular component? If so, into what level?

Why we need to learn about DOM then?

- 1., [DOM events](#) are the Alpha and Omega of web development, and you will use events on daily basis.
- 2., Single Page Applications utilize a [special DOM](#) or an abstraction above the HTML DOM, and all use that under the hood. While they are similar, there are differences – it is important not to mismatch them.
- 3., While direct DOM manipulation with the DOM API is used less frequently, the nodes itself are very important.
- 4., SPAs provides [ways to access the HTML DOM](#) and sometimes it is necessary to use them.
- 5., Important problems of the web development are framework / technology agnostic and related to DOM: how to break down / build the component structure, how to handle events?

As a consequence, we must learn about the DOM tree, and how to handle that. DOM is really a fundamental part of the web development.



... as you may already guessed: the [DOM](#)

Before we start...

... there is a key takeaway here

If you work with SPAs (including Web Components) try to think in that way.

If you run into issues with React, the obvious solution is not to work with the DOM directly, rather to restructure your component boundaries and data flow.

If you used DOM manipulation in a wrong way, you have to forget that.



a Team Lead trying to force to unlearn the excessive usage of [refs](#) in a React project.

Do we use jQuery nowadays?

Usually no but you should not be afraid of that. We use it in component-based architectures, or as a targeted solution in special cases.

You came here to learn how to deal with problems on an enterprise level, therefore the technology itself is secondary. Should we use React on a project, that will be completely different in the next project, also, as SPAs are evolving in a fast pace.

That being said, you won't see jQuery in any new project. It is not because SPAs are better from a developers' perspective (they are not), it is mainly because it is easier to govern the technical aspects of the project from a Team Lead / management side.

DOCUMENT OBJECT MODEL

DOM tree

When a web page is loaded, the browser creates the Document Object Model of the page: [a tree](#)

In this tree everything is a node, and every node is an object. The DOM can represent HTML or XML documents.

```
...<html> == $0
  ▼<head>
    <title>With Treebeard and the Ents</title>
  </head>
  ▼<body>
    <em class="elf" name="legolas">Legolas</em>
    <p id="question">Then are we not to see the
      merry young hobbits again?</p>
    <em class="maia" name="gandalf">Gandalf</em>
    <p id="answer">Who knows? Have patience.
      Go where you must go, and hope!</p>
  </body>
</html>
```



```
L HTML
  HEAD
    #text:
    TITLE
      #text: With Treebeard and the Ents
    #text:
  #text:
  BODY
    #text:
    EM class="elf" name="legolas"
      #text: Legolas
    #text:
    P id="question"
      #text: Then are we not to see the merry young hobbits again?
    #text:
    EM class="maia" name="gandalf"
      #text: Gandalf
    #text:
    P id="answer"
      #text: Who knows? Have patience. Go where you must go, and hope!
    #text:
```

[Wikipedia: Tree \(data_structure\)](#)

DOM – elements

Let's break down the DOM!

There are different types of nodes, and the methods return different types of collections.

document is `HTMLDocument` →

nodeList is `HTMLCollection` →

node is `HTMLElement` →

```
> document
< ▼ #document
  <html>
    ▶ <head>...</head>
    ▼ <body>
      <em class="elf" name="legolas">Legolas</em>
      <p id="question">Then are we not to see the
        merry young hobbits again?</p>
      <em class="maia" name="gandalf">Gandalf</em>
      <p id="answer">Who knows? Have patience.
        Go where you must go, and hope!</p>
    </body>
  </html>

> document.toString();
< "[object HTMLDocument]"

> let emNodes = document.getElementsByTagName("em");
< undefined

> emNodes.toString();
< "[object HTMLCollection]"

> emNodes.length;
< 2

> emNodes[0].toString();
< "[object HTMLElement]"

> emNodes[0];
< <em class="elf" name="legolas">Legolas</em>
```

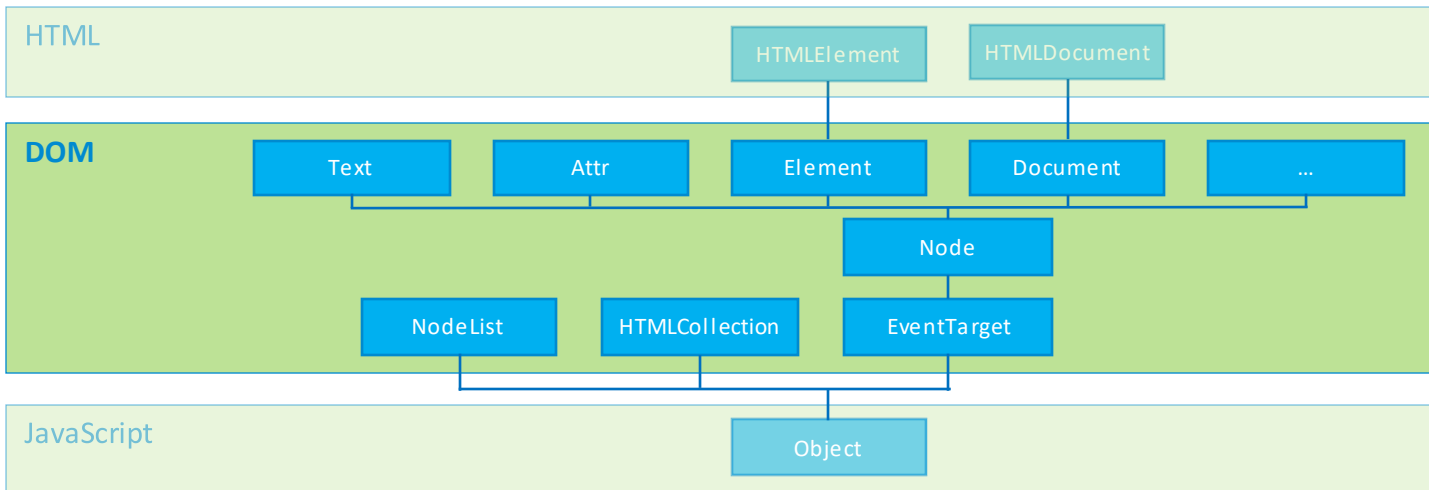
DOM – node types

the document node	Document	
	Element	elements—in HTML: <code><div></code> , <code><p></code> , <code></code> ...
the textual content of an element: <code>Gimli</code>	Text	
	Comment	comments: <code><!-- a troll is still stronger, though --></code>
attributes also nodes: <code><gandalf class="maia">Mithrandir</gandalf></code>	Attr	
	ShadowRoot	using Web Components, parts of the DOM can be separated*
<code><![CDATA[In HTML only used foreign content: MathML/SVG]]></code>	CDATASection	
	ProcessingInstruction	<code><?xml-stylesheet type="text/xsl" href="style.xsl"?></code>
	DocumentFragment	

* it solves a lot of problems...

DOM – object hierarchy

Without getting too deep into the topic, the object hierarchy looks like this:



```

> Object.keys(HTMLCollection.prototype)
< ▶ (3) ["length", "item", "namedItem"]

> Object.keys(Document.prototype)
< (224) ["implementation", "URL", "documentURI", "compatMode", "characterSet", "charset", "inputEncoding", "contentType", "doctype", "documentElement", "xmlEncoding", "xmlVersion", "xmlStandalone", "domain", "referrer", "cookie", "lastModified", "readyState", "title", "dir", "body", "head", "images", "embeds", "plugins", "links", "forms", "scripts", "currentScript", "defaultView", "designMode", "onreadystatechange", "anchors", "applets", "fgColor", "linkColor", "vlinkColor", "alinkColor", "bgColor", "all", "scrollingElement", "onpointerlockchange", "onpointerlockerror", "hidden", "visibilityState", "wasDiscarded", "featurePolicy", "webkitVisibilityState", "webkitHidden", "onbeforecopy", "onbeforecut", "onbeforepaste", "onfreeze", "onresume", "onsearch", "onsecuritypolicyviolation", "onvisibilitychange", "fullscreenEnabled", "fullscreen", "onfullscreenchange", "onfullscreenerror", "webkitIsFullScreen", "webkitCurrentFullScreenElement", "webkitFullscreenEnabled", "webkitFullscreenElement", "onwebkitfullscreenchange", "onwebkitfullscreenerror", "rootElement", "onabort", "onblur", "oncancel", "oncanplay", "oncanplaythrough", "onchange", "onclick", "onclose", "oncontextmenu", "oncuechange", "ondblclick", "ondrag", "ondragend", "ondragenter", "ondragleave", "ondragover", "ondragstart", "ondrop", "ondurationchange", "onemptied", "onended", "onerror", "onfocus", "onformdata", "oninput", "oninvalid", "onkeydown", "onkeypress", "onkeyup", "onload", "onloadeddata", "onloadedmetadata", ...]

> Object.keys(HTMLElement.prototype)
< (117) ["title", "lang", "translate", "dir", "hidden", "accessKey", "draggable", "spellcheck", "autocapitalize", "contentEditable", "isContentEditable", "inputMode", "offsetParent", "offsetTop", "offsetLeft", "offsetWidth", "offsetHeight", "style", "innerText", "outerText", "onabort", "onblur", "oncancel", "oncanplay", "oncanplaythrough", "onchange", "onclick", "onclose", "oncontextmenu", "oncuechange", "ondblclick", "ondrag", "ondragend", "ondragenter", "ondragleave", "ondragover", "ondragstart", "ondrop", "ondurationchange", "onemptied", "onended", "onerror", "onfocus", "onformdata", "oninput", "oninvalid", "onkeydown", "onkeypress", "onkeyup", "onload", "onloadeddata", "onloadedmetadata", "onloadstart", "onmousedown", "onmouseenter", "onmouseleave", "onmousemove", "onmouseout", "onmouseover", "onmouseup", "onmousewheel", "onpause", "onplay", "onplaying", "onprogress", "onratechange", "onreset", "onresize", "onscroll", "onseeked", "onseeking", "onselect", "onstalled", "onsubmit", "onsuspend", "ontimeupdate", "ontoggle", "onvolumechange", "onwaiting", "onwebkitanimationend", "onwebkitanimationiteration", "onwebkitanimationstart", "onwebkittransitionend", "onwheel", "onauxclick", "ongotpointercapture", "onlostpointercapture", "onpointerdown", "onpointermove", "onpointerup", "onpointercancel", "onpointerover", "onpointerout", "onpointerenter", "onpointerleave", "onselectstart", "onselectionchange", "onanimationend", "onanimationiteration", "onanimationstart", ...]

> Object.keys(Element.prototype)
< (129) ["namespaceURI", "prefix", "localName", "tagName", "id", "className", "classList", "slot", "attributes", "shadowRoot", "part", "assignedSlot", "innerHTML", "outerHTML", "scrollTop", "scrollLeft", "scrollWidth", "scrollHeight", "clientTop", "clientLeft", "clientWidth", "clientHeight", "attributeStyleMap", "onbeforecopy", "onbeforecut", "onbeforepaste", "onsearch", "elementTiming", "onfullscreenchange", "onfullscreenerror", "onwebkitfullscreenchange", "onwebkitfullscreenerror", "onbeforexrselect", "children", "firstElementChild", "lastElementChild", "childElementCount", "previousElementSibling", "nextElementSibling", "after", "animate", "append", "attachShadow", "before", "closest", "computedStyleMap", "getAttribute", "getAttributeNS", "getAttributeNames", "getAttributeNode", "getAttributeNodeNS", "getBoundElement", "getClientRects", "getElementsByName", "getElementsByTagName", "getElementsByTagNameNS", "hasAttribute", "hasAttributeNS", "hasAttributes", "hasPointerCapture", "insertAdjacentElement", "insertAdjacentHTML", "insertAdjacentText", "matches", "prepend", "querySelector", "querySelectorAll", "releasePointerCapture", "remove", "removeAttribute", "removeAttributeNS", "removeAttributeNode", "replaceWith", "requestFullscreen", "requestPointerLock", "scroll", "scrollBy", "scrollIntoView", "scrollIntoViewIfNeeded", "scrollTo", "setAttribute", "setAttributeNS", "setAttributeNode", "setAttributeNodeNS", "setPointerCapture", "toggleAttribute", "webkitMatchesSelector", "webkitRequestFullscreen", "webkitRequestFullscreen", "ariaAtomic", "ariaAutoComplete", "ariaBusy", "ariaChecked", "ariaColCount", "ariaColIndex", "ariaColSpan", "ariaCurrent", "ariaDescription", "ariaDisabled", "ariaExpanded", ...]

> Object.keys(Node.prototype)
< (47) ["nodeType", "nodeName", "baseURI", "isConnected", "ownerDocument", "parentNode", "parentElement", "childNodes", "firstChild", "lastChild", "previousSibling", "nextSibling", "nodeValue", "textContent", "ELEMENT_NODE", "ATTRIBUTE_NODE", "TEXT_NODE", "CDATA_SECTION_NODE", "ENTITY_REFERENCE_NODE", "ENTITY_NODE", "PROCESSING_INSTRUCTION_NODE", "COMMENT_NODE", "DOCUMENT_NODE", "DOCUMENT_TYPE_NODE", "DOCUMENT_FRAGMENT_NODE", "NOTATION_NODE", "DOCUMENT_POSITION_DISCONNECTED", "DOCUMENT_POSITION_PRECEDING", "DOCUMENT_POSITION_FOLLOWING", "DOCUMENT_POSITION_CONTAINS", "DOCUMENT_POSITION_CONTAINED_BY", "DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC", "appendChild", "cloneNode", "compareDocumentPosition", "contains", "getRootNode", "hasChildNodes", "insertBefore", "isDefaultNamespace", "isEqualNode", "isSameNode", "lookupNamespaceURI", "lookupPrefix", "normalize", "removeChild", "replaceChild"]

> Object.keys(EventTarget.prototype)
< ▶ (3) ["addEventListener", "dispatchEvent", "removeEventListener"]

```

... and in the darkness [bind](#) them

ACCESSING THE DOM

Accessing the DOM

Method	Description	on the document node	on an element node
<code>.getElementById(DOMString elementId)</code>	Returns the <i>first element</i> within node's descendants whose ID is <i>elementId</i> .	✓	
<code>.getElementsByName(qualifiedName)</code>	Returns a <i>HTMLCollection</i> of all descendant elements whose qualified name is <i>qualifiedName</i> . Case-insensitive.	✓	✓
<code>.getElementsByClassName(classNames)</code>	Returns a <i>HTMLCollection</i> of the elements in the object on which the method was invoked (a document or an element) that have all the classes given by <i>classNames</i> .	✓	✓
<code>.getElementsByTagName(name)</code>	Returns a <i>NodeList</i> of elements in the Document that have a <i>name</i> attribute with the value <i>name</i> .	✓	
<code>.querySelector(selector)</code>	Returns the <i>first Element</i> within the document that matches the specified selector, or group of selectors. If no matches are found, null is returned.	✓	✓
<code>.querySelectorAll(selector)</code>	Returns a <i>static (not live) NodeList</i> representing a list of the document's elements that match the specified group of selectors.	✓	✓

Accessors on a sub tree

`.getElementsByTagName` and
`.getElementsByClassName` can be
used on an element node, too.

can be used on `document` →

and on an `element node` as well →

but `.getElementById` is not →

```
> document
< ▼ #document
  <html>
    ▶ <head>...</head>
    ▼ <body>
      ▶ <div>...</div>
      ▼ <div>
        <em class="maia" name="gandalf">Gandalf</em>
        <p id="answer">Who knows? Have patience.
          Go where you must go, and hope!</p>
      </div>
    </body>
  </html>

> document.getElementsByTagName("p");
< ▼ HTMLCollection(2) [p#question, p#answer, question: p#question, answer: p#answer] ⓘ
  ▶ 0: p#question
  ▶ 1: p#answer
  length: 2
  answer: p#answer
  question: p#question
  __proto__: HTMLCollection

> document.getElementsByTagName("div")[0].getElementsByTagName("p");
< ▼ HTMLCollection [p#question, question: p#question] ⓘ
  ▶ 0: p#question
  length: 1
  question: p#question
  __proto__: HTMLCollection

> document.getElementsByTagName("div")[0].getElementById("answer");
✖ ▶ Uncaught TypeError: document.getElementsByTagName(...)[0].getElementById is not a function
   at <anonymous>:1:41
```

Accessors by id, tag, class, and name attribute

by id (returns a single node)	→	<pre>> document.getElementById("question"); << <p id="question">Then are we not to see the merry young hobbits again?</p></pre>
by tag name (returns an HTMLCollection)	→	<pre>> document.getElementsByTagName("p"); << ▼HTMLCollection(2) [p#question, p#answer, question: ► 0: p#question ► 1: p#answer length: 2 ► answer: p#answer ► question: p#question ► __proto__: HTMLCollection</pre>
by class name (returns an HTMLCollection)	→	<pre>> document.getElementsByClassName("elf"); << ▼HTMLCollection [em.elf, legolas: em.elf] ⓘ ► 0: em.elf length: 1 ► legolas: em.elf ► __proto__: HTMLCollection</pre>
by the name attribute (returns a NodeList)	→	<pre>> document.getElementsByName("legolas"); << ▼NodeList [em.elf] ⓘ ► 0: em.elf length: 1 ► __proto__: NodeList</pre>

Accessors by CSS selector

With the query selectors, a [CSS selector](#) can be used, too.

```
> document
< ▼ #document
  <html>
    ▶ <head>...</head>
    ▼ <body>
      ▼ <div>
        <em class="elf" name="legolas">Legolas</em>
        <p id="question">Then are we not to see the
          merry young hobbits again?</p>
      </div>
    ▶ <div>...</div>
  </body>
</html>

> document.querySelector("body div");
< ▼ <div>
  <em class="elf" name="legolas">Legolas</em>
  <p id="question">Then are we not to see the
    merry young hobbits again?</p>
</div>

> document.querySelectorAll("body div");
< ▶ NodeList(2) [div, div]
```

Modifying text content

we can **get and set the text content** of a node



```
> let question = document.getElementById("question");
< undefined
> question;
< <p id="question">Then are we not to see the
    merry young hobbits again?</p>
> question.innerText = question.innerText + " - said Legolas.";
< "Then are we not to see the merry young hobbits again? - said Legolas."
> question
< <p id="question">Then are we not to see the merry young hobbits again? - said Legolas.</p>
> document
< ▼ #document
    <html>
      ▶ <head>...</head>
      ▼ <body>
        ▼ <div>
          <em class="elf" name="legolas">Legolas</em>
          <p id="question">Then are we not to see the merry young hobbits again? - said Legolas.</p>
        </div>
      ▶ <div>...</div>
    </body>
  </html>
```

it also **appears on the document**, itself



.innerText vs .textContent

We also have `textContent`, but these are significantly different:

`textContent` gets the content of all elements, including `<script>` and `<style>` elements. In contrast, `innerText` only shows “human-readable” elements.

`textContent` returns every element in the node. In contrast, `innerText` is aware of styling and won’t return the text of “hidden” elements.

Moreover, since `innerText` takes CSS styles into account, reading the value of `innerText` triggers a reflow to ensure up-to-date computed styles. (Reflows can be computationally expensive, and thus should be avoided when possible.)

```
> question.textContent;
< "Then are we not to see the merry young hobbits again? – said Legolas."

> document.body.innerText;
< "Legolas

    Then are we not to see the merry young hobbits again? – said Legolas.

    Gandalf

    Who knows? Have patience. Go where you must go, and hope!"

> document.body.textContent;
< "

    Legolas
    Then are we not to see the merry young hobbits again? – said Legolas.

    Gandalf
    Who knows? Have patience.
        Go where you must go, and hope!

"
```

Attributes

We have different methods to **get**, **set**, **check** and for **removing attributes**.

it actually sets an inline style →

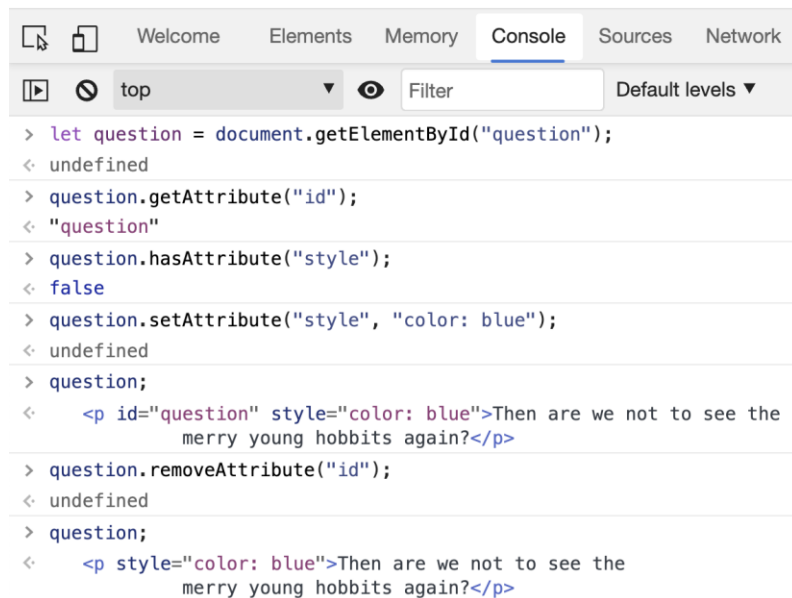
id is also an attribute and
can be removed →

Legolas

Then are we not to see the merry young hobbits again?

Gandalf

Who knows? Have patience. Go where you must go, and hope!



```
> let question = document.getElementById("question");
< undefined
> question.getAttribute("id");
< "question"
> question.hasAttribute("style");
< false
> question.setAttribute("style", "color: blue");
< undefined
> question;
< <p id="question" style="color: blue">Then are we not to see the
  merry young hobbits again?</p>
> question.removeAttribute("id");
< undefined
> question;
< <p style="color: blue">Then are we not to see the
  merry young hobbits again?</p>
```

Attributes vs properties

Some standard attributes are mirrored as a property on the object

`id`, `class`, `style`, etc. However, the mappings are not trivial: `className` becomes `class`, `style properties` become `style`.

Setting them as a node property will affect the attributes as well.

there is `no style attribute`

but there is `a style property`

setting the property ...

... and the attribute, both work

```
> document.body
< <body>
  <em class="elf" name="legolas">Legolas</em>
  <p id="question">Then are we not to see the
    merry young hobbits again?</p>
  <em class="maia" name="gandalf">Gandalf</em>
  <p id="answer">Who knows? Have patience.
    Go where you must go, and hope!</p>
</body>

> let answer = document.body.querySelector("em.maia[name=gandalf] + p#answer");
< undefined

> answer.hasAttribute("style");
< false

> answer.style;
< ▶ CSSStyleDeclaration {alignContent: "", alignItems: "", alignSelf: "", alignme

> answer.style.alignContent = "start";
< "start"

> answer
< <p id="answer" style="align-content: start;">Who knows? Have patience.
  Go where you must go, and hope!</p>

> answer.setAttribute("style", "align-content: center");
< undefined

> answer
< <p id="answer" style="align-content: center;">Who knows? Have patience.
  Go where you must go, and hope!</p>
```

.classList

`classList` provides a convenient way to modify the classes of an element

Working with classes could be a bit of tricky, because the class attribute is a space separated list – having dedicated methods could help.

Please check the [compatibility](#), though.

add →

remove →

replace →

toggle-on →

toggle-off →

```
> let gandalf = document.querySelector("em.maia");
< undefined
> gandalf.classList;
< ▶ DOMTokenList ["maia", value: "maia"]
> gandalf.classList.add("wizard");
< undefined
> gandalf.classList.remove("maia");
< undefined
> gandalf.classList;
< ▶ DOMTokenList ["wizard", value: "wizard"]
> gandalf.classList.replace("wizard", "maia");
< true
> gandalf.classList;
< ▶ DOMTokenList ["maia", value: "maia"]
> gandalf.classList.toggle("hidden");
< true
> gandalf.classList;
< ▶ DOMTokenList(2) ["maia", "hidden", value: "maia hidden"]
> gandalf.classList.toggle("hidden");
< false
> gandalf.classList;
< ▶ DOMTokenList ["maia", value: "maia"]
```

MODIFYING THE DOM STRUCTURE

.innerHTML

With `innerHTML`, you can insert text as parsed HTML into the DOM...

...but you probably should not do that.

If you use `innerHTML` with an unknown content (from a CMS, typically), then you just implemented an **XSS vulnerability**.

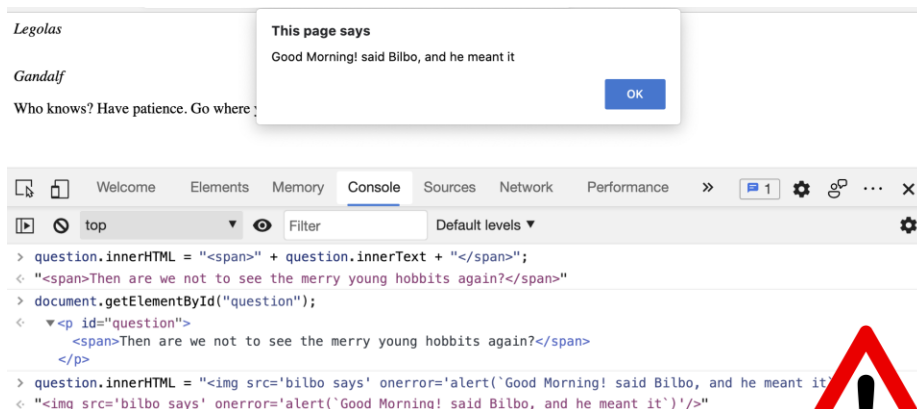
It means that an attacker can run any script on the page, stealing data, starting transactions, anything, really.

You cannot trust a content from a CMS.

Use `innerText`, simply.

XSS = Cross Site Scripting, when the browser runs injected code.

... in a way you really don't want



it works...



.innerHTML does not run script elements, but...

Don't be fooled by the perceived security

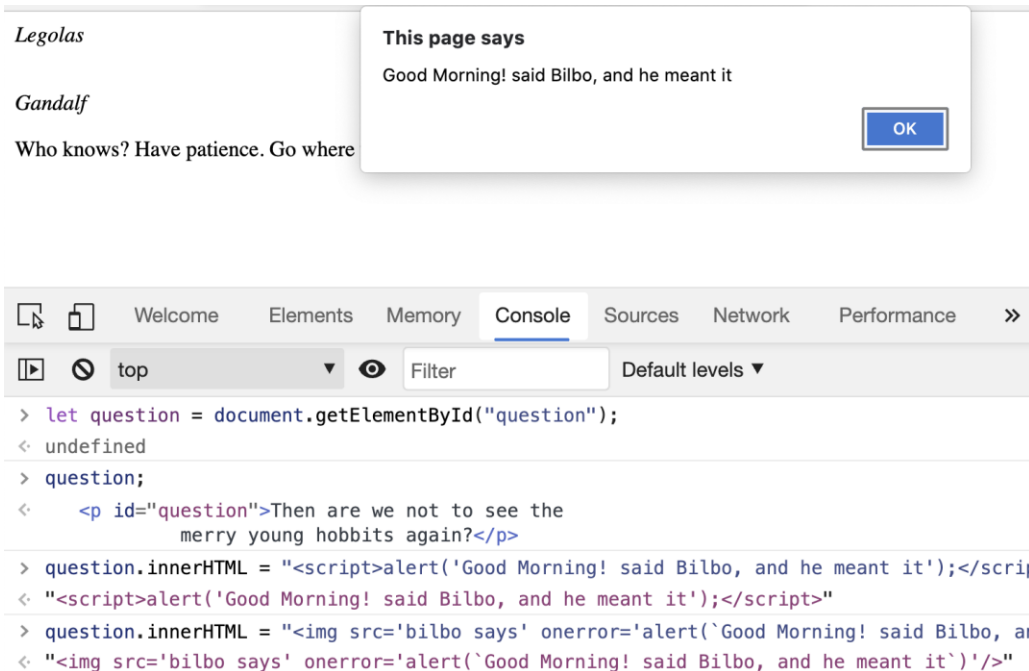
The `innerHTML` does not run the scripts directly, this works in a different way: when the browser tries to request the image from a wrong URL, it triggers an error.

And then the script will run.

Lesson learned: there will be always a tricky way to run a script from an injected HTML – just because you don't know how, it is still possible!

this does not work →

but it does! →





.createElement

New DOM nodes can be added

createElement will create a new element, but it won't add to the DOM.

we are not there yet →

we have to append it to an element →

```
> document.body;
< ▼<body>
  <em class="elf" name="legolas">Legolas</em>
  <p id="question">Then are we not to see the
    merry young hobbits again?</p>
  <em class="maia" name="gandalf">Gandalf</em>
  <p id="answer">Who knows? Have patience.
    Go where you must go, and hope!</p>
</body>

> let bilbo = document.createElement("em");
< undefined

> document.body.getElementsByTagName("em");
< ▶HTMLCollection(2) [em.elf, em.maia, legolas: em.elf, gandalf: em.maia]

> bilbo.className = "hobbit";
  bilbo.setAttribute("name", "bilbo");
  bilbo.innerText = "Good Morning! said Bilbo, and he meant it.";
< "Good Morning! said Bilbo, and he meant it."

> document.body.appendChild(bilbo);
< <em class="hobbit" name="bilbo">Good Morning! said Bilbo, and he meant it.</em>

> document.body
< ▼<body>
  <em class="elf" name="legolas">Legolas</em>
  <p id="question">Then are we not to see the
    merry young hobbits again?</p>
  <em class="maia" name="gandalf">Gandalf</em>
  <p id="answer">Who knows? Have patience.
    Go where you must go, and hope!</p>
  <em class="hobbit" name="bilbo">Good Morning! said Bilbo, and he meant it.</em>
</body>
```

A bit more complex example

web development is **super green** →

a text node is an **object** →

inserting a new node with **insertBefore** →

success →

wait! wrong order →

1., we get the first child, →

2., then we remove that,

3., but as a return value we
have that, so we can
insert it again

```
> let narrator = document.createTextNode("The sun was shining, and the grass was  
very green.");  
< undefined  
> narrator.toString();  
< "[object Text]"  
> bilbo;  
< <em class="hobbit" name="bilbo">Good Morning! said Bilbo, and he meant it.</em>  
> bilbo.insertBefore(narrator, bilbo.firstChild);  
< "The sun was shining, and the grass was very green."  
> bilbo;  
< ▼<em class="hobbit" name="bilbo">  
    "The sun was shining, and the grass was very green."  
    "Good Morning! said Bilbo, and he meant it."  
    </em>  
> bilbo.childNodes;  
< ▶NodeList(2) [text, text]  
> bilbo.appendChild(  
    bilbo.removeChild(  
        bilbo.firstChild  
    )  
);  
< "The sun was shining, and the grass was very green."  
> bilbo  
< ▼<em class="hobbit" name="bilbo">  
    "Good Morning! said Bilbo, and he meant it."  
    "The sun was shining, and the grass was very green."  
    </em>
```

Q&A