Browser Internals Course

Flavio Copes

Updated August 3, 2026

Contents

Learn how a browser loads resources, builds and paints a page, runs JavaScript, isolates sites, stores data, and exposes performance problems.

What you'll learn: Trace a page from URL to pixels and use browser DevTools to explain and fix loading, rendering, memory, and responsiveness problems.

Networking and resource loading

DNS, connections, requests, parsing, priorities, and reuse.

From a URL to a page

Follow the browser from an entered URL through address resolution, connection setup, an HTTP exchange, and the first bytes of HTML.

Entering a URL starts several connected systems.

The browser parses the URL, applies navigation rules, and checks whether a cached response can be reused. If it needs the network, it resolves the hostname, establishes a connection, negotiates TLS for HTTPS, and sends an HTTP request.

The server returns a status, headers, and response bytes. The browser does not normally wait for the final HTML byte. It can decode and parse the stream while more data arrives.

The parser builds the DOM and discovers stylesheets, scripts, images, and fonts. CSS contributes to calculated styles. The browser then determines layout, records paint instructions, rasterizes pixels, and composites the final frame.

JavaScript can pause parsing, change the DOM, or schedule later work. This is why loading, parsing, rendering, and script execution overlap rather than forming one simple queue.

Open DevTools, select Network, and reload a page. Find the document request, then inspect its Timing and Initiator information. The first image on screen may begin loading before the document request has finished downloading.

Exercise: write down the chain from URL to first pixels. Mark which steps can use cached data and which steps can overlap.

DNS, Domain Name System

A high-level overview of how DNS, the Domain Name System, maps domain names to IP addresses through a tree of servers starting at the root DNS server.

You don't usually try to reach for a website using its IP address. You *can*, but it's very rare.

You usually use a **domain name**. Like google.com, or flaviocopes.com.

This is very handy because for example I can change the server and company I use to host a website, while maintaining the same domain name.

The system that maps domain names to IP addresses is called DNS: **Domain Name System**.

DNS is a network of servers. Your provider will have its own DNS, your router already preconfigured to use it.

You can also choose to use Google's DNS server, which has the IP address `8.8.8.8`.

Those DNS servers will receive the requests from your computer, and in turn will ask their own reference DNS server.

The system is organized like a tree. There is one DNS server at the top, called **root DNS server**.

To simplify, it knows the IP address of the DNS servers that are managing each domain extension, like `com`, `net`, `org` and so on, including the country-specific domain extensions and the new ones like `blog`, `dev` or `tech`.

Those DNS servers know the IP addresses mapping of all the domains under their extension.

Of course the system is set up to ensure caching, redundancy and capacity to endure high concurrent requests, but this is the general idea.

If you want to understand the actual records behind a domain (A, CNAME, MX, TXT and so on), I built a free DNS records explainer that walks you through them.

The HTTPS protocol

Learn how the HTTPS protocol encrypts your traffic to stop man-in-the-middle attacks, the role of TLS, what stays unencrypted, and why HTTP/2 makes it fast.

HTTP is insecure by design.

When you open your browser and ask a web server to send you a webpage, your data performs 2 trips: 1 from the browser to the web server, and 1 from the web server to the browser.

Then, depending on the content of the web page, you might have more connections required to get the CSS files, the JavaScript files, images, and so on.

During any of those connections, any network your data is going to cross can be **inspected** and **manipulated**.

The consequences can be serious: you might have all your network activity monitored and logged, by a 3rd party you are not even aware it exist, some networks might inject ads, and you might be subject to a man-in-the-middle attack, a security threat where the attacker can manipulate your data and even impersonate your computer over the network. It's very easy for someone to just listen to HTTP packets being transmitted over a public and unencrypted Wi-Fi network.

HTTPS aims to solve the problem at the root: the entire communication between your browser and the web server is encrypted.

Privacy and security are a major concern in today's internet. A few years ago, you could get away with just using an encrypted connection in login-protected pages, or during an e-commerce checkout. Also because of SSL certificates pricing and complications, most websites just used HTTP.

Today HTTPS is a requirement on any site. More than 50% of the whole Web uses it now. Google Chrome recently started marking HTTP sites as insecure, just to give you a valid reason to have HTTPS mandatory (and forced) on all your websites.

When using HTTP the default server port is 80, and on HTTPS it's 443. It does not need to be explicitly added if the server uses the default port, of course.

HTTPS is also sometimes called *HTTP over SSL*, or *HTTP over TLS*.

The difference between the two is simple: TLS is the successor of SSL.

When using HTTPS, the only thing that is not encrypted is the web server domain, and the server port.

Every other information, including the resource path, headers, cookies and query parameters are all encrypted.

I won't go in the details of analyzing how the TLS protocol works under the hoods, but you might think it's adding a good amount of **overhead**, and you would be right.

Any computation that's added to the processing of network resources causes overhead both on the client, the server, and to the transmitted packets size.

However HTTPS enables the use of the newest protocol **HTTP/2**, which has a huge advantage over HTTP/1.1: it way faster.

Why? There are many reasons, one is header compression, one is resource multiplexing. One is server push: the server can push more resources when one resource is requested. So, if the browser requests a page, it will also receive all the resources needed (images, CSS, JS).

Details aside, HTTP/2 is a huge improvement over HTTP/1.1 **and it requires HTTPS**. This means that HTTPS, despite having the encryption overhead, happens to be way faster than HTTP, if things are properly configured with a modern setup.

How HTTP requests work

Learn what happens when you type a URL and press enter, from the DNS lookup and TCP handshake to sending the HTTP request and parsing the response HTML.

This article describes how browsers perform page requests using the HTTP/1.1 protocol

If you ever did an interview, you might have been asked: "what happens when you type something into the Google search box and press enter".

It's one of the most popular questions you get asked. People just want to see if you can explain

some rather basic concepts and if you have any clue how the internet actually works.

In this post, I'll analyze what happens when you type an URL in the address bar of your browser and press enter.

It's a very interesting topic to dissect in a blog post, as it touches many technologies I can dive into in separate posts.

This is tech that is very rarely changed, and powers one the most complex and wide ecosystems ever built by humans.

The HTTP protocol

First, I mention HTTPS in particular because things are different from an HTTPS connection.

I analyze URL requests only

Modern browsers have the capability of knowing if the thing you wrote in the address bar is an actual URL or a search term, and they will use the default search engine if it's not a valid URL.

I assume you type an actual URL.

When you enter the URL and press enter, the browser first builds the full URL.

If you just entered a domain, like `flaviocopes.com`, the browser by default will prepend `HTTP://` to it, defaulting to the HTTP protocol.

Things relate to macOS / Linux

Just FYI. Windows might do some things slightly differently.

DNS Lookup phase

The browser starts the DNS lookup to get the server IP address.

The domain name is a handy shortcut for us humans, but the internet is organized in such a way that computers can look up the exact location of a server through its IP address, which is a set of numbers like `222.324.3.1` (IPv4).

First, it checks the DNS local cache, to see if the domain has already been resolved recently.

Chrome has a handy DNS cache visualizer you can see at chrome://net-internals/#dns

If nothing is found there, the browser uses the DNS resolver, using the `gethostbyname` POSIX system call to retrieve the host information.

gethostbyname `gethostbyname` first looks in the local hosts file, which on macOS or Linux is located in `/etc/hosts`, to see if the system provides the information locally.

If this does not give any information about the domain, the system makes a request to the DNS server.

The address of the DNS server is stored in the system preferences.

Those are 2 popular DNS servers:

- `8.8.8.8`: the Google public DNS server
- `1.1.1.1`: the CloudFlare DNS server

Most people use the DNS server provided by their internet provider.

The browser performs the DNS request using the UDP protocol.

TCP and UDP are two of the foundational protocols of computer networking. They sit at the same conceptual level, but TCP is connection-oriented, while UDP is a connectionless protocol, more lightweight, used to send messages with little overhead.

> How the UDP request is performed is not in the scope of this tutorial

The DNS server might have the domain IP in the cache. If not, it will ask the **root DNS server**. That's a system (composed of 13 actual servers, distributed across the planet) that drives the entire internet.

The DNS server does *not* know the address of each and every domain name on the planet.

What it knows is where the **top-level DNS resolvers** are.

A top-level domain is the domain extension: `.com`, `.it`, `.pizza` and so on.

Once the root DNS server receives the request, it forwards the request to that top-level domain (TLD) DNS server.

Say you are looking for `flaviocopes.com`. The root domain DNS server returns the IP of the .com TLD server.

Now our DNS resolver will cache the IP of that TLD server, so it does not have to ask the root DNS server again for it.

The TLD DNS server will have the IP addresses of the authoritative Name Servers for the domain we are looking for.

> How? When you buy a domain, the domain registrar sends the appropriate TDL the name servers. When you update the name servers (for example, when you change the hosting provider), this information will be automatically updated by your domain registrar.

Those are the DNS servers of the hosting provider. They are usually more than 1, to serve as backup.

For example:

- `ns1.dreamhost.com`
- `ns2.dreamhost.com`
- `ns3.dreamhost.com`

The DNS resolver starts with the first, and tries to ask the IP of the domain (with the subdomain, too) you are looking for.

That is the ultimate source of truth for the IP address.

Now that we have the IP address, we can go on in our journey.

TCP request handshaking

With the server IP address available, now the browser can initiate a TCP connection to that.

A TCP connection requires a bit of handshaking before it can be fully initialized and you can start sending data.

Once the connection is established, we can send the request

Sending the request

The request is a plain text document structured in a precise way determined by the communication protocol.

It's composed of 3 parts:

- the request line
- the request header
- the request body

The request line The request line sets, on a single line:

- the HTTP method
- the resource location
- the protocol version

Example:

```
GET / HTTP/1.1
```

The request header The request header is a set of `field: value` pairs that set certain values.

There are 2 mandatory fields, one of which is `Host`, and the other is `Connection`, while all the other fields are optional:

```
Host: flaviocopes.com
Connection: close
```

`Host` indicates the domain name which we want to target, while `Connection` is always set to `close` unless the connection must be kept open.

Some of the most used header fields are:

- `Origin`
- `Accept`
- `Accept-Encoding`
- `Cookie`
- `Cache-Control`
- `Dnt`

but many more exist.

The header part is terminated by a blank line.

The request body The request body is optional, not used in GET requests but very much used in POST requests and sometimes in other verbs too, and it can contain data in JSON format.

Since we're now analyzing a GET request, the body is blank and we'll not look more into it.

The response

Once the request is sent, the server processes it and sends back a response.

The response starts with the status code and the status message. If the request is successful and returns a 200, it will start with:

```
200 OK
```

The request might return a different status code and message, like one of these:

```
404 Not Found
403 Forbidden
301 Moved Permanently
500 Internal Server Error
304 Not Modified
401 Unauthorized
```

The response then contains a list of HTTP headers and the response body (which, since we're making the request in the browser, is going to be HTML)

Parse the HTML

The browser now has received the HTML and starts to parse it, and will repeat the exact same process we did for all the resources required by the page:

- CSS files
- images
- the favicon
- JavaScript files
- ...

How browsers render the page then is out of the scope, but it's important to understand that the process I described is not just for the HTML pages, but for any item that's served over HTTP.

How the browser discovers resources

Understand how the HTML parser and preload scanner discover stylesheets, scripts, images, fonts, and other dependencies.

As HTML arrives, the parser creates nodes and discovers referenced resources.

For example, a stylesheet link, script source, and image source expose their URLs directly:

```
<link rel="stylesheet" href="https://flaviocopes.com/styles.css">
<script src="/app.js" defer></script>
<img src="/hero.webp" alt="Mountain trail">
```

Browsers also use a speculative preload scanner. It can look ahead for obvious URLs while the main parser waits on blocking work.

The scanner cannot discover a URL that JavaScript constructs later. A hero image inserted only after a client-side render may start much later than one present in the initial HTML.

Open the Network panel and inspect the Initiator column. Parser-discovered resources should point back to the document. Script-discovered requests show a JavaScript initiator.

Keep critical resources visible in initial HTML when early discovery matters. Do not add preload hints before checking whether normal markup already discovers the resource soon enough.

Exercise: load one image from HTML and create another after `DOMContentLoaded`. Compare their

start times and initiators.

Efficiently load JavaScript with defer and async

Learn how the async and defer attributes change the way scripts load on an HTML page, how they affect parsing and rendering, and which to choose for speed.

When loading a script on an HTML page, you need to be careful not to harm the loading performance of the page.

A script is traditionally included in the page in this way:

```
<script src="script.js"></script>
```

whenever the HTML parser finds this line, a request will be made to fetch the script, and the script is executed.

Once this process is done, the parsing can resume, and the rest of the HTML can be analyzed.

As you can imagine, this operation can have a huge impact on the loading time of the page.

If the script takes a little longer to load than expected, for example if the network is a bit slow or if you're on a mobile device and the connection is a bit sloppy, the visitor will likely see a blank page until the script is loaded and executed.

The position matters

When you first learn HTML, you're told script tags live in the `<head>` tag:

```
<html>
  <head>
    <title>Title</title>
    <script src="script.js"></script>
  </head>
  <body>
    ...
  </body>
</html>
```

As I told you earlier, when the parser finds this line, it goes to fetch the script and executes it. *Then*, after it's done with this task, it goes on to parse the body.

This is bad because there is a lot of delay introduced. A very common solution to this issue is to put the `script` tag at the bottom of the page, just before the closing `</body>` tag.

In doing so, the script is loaded and executed after all the page is already parsed and loaded, which is a **huge improvement** over the `head` alternative.

This is the best thing you can do if you need to support older browsers that do not support two relatively recent features of HTML: `async` and `defer`.

Async and Defer

Both async and defer are boolean attributes. Their usage is similar:

```
<script async src="script.js"></script>
```

```
<script defer src="script.js"></script>
```

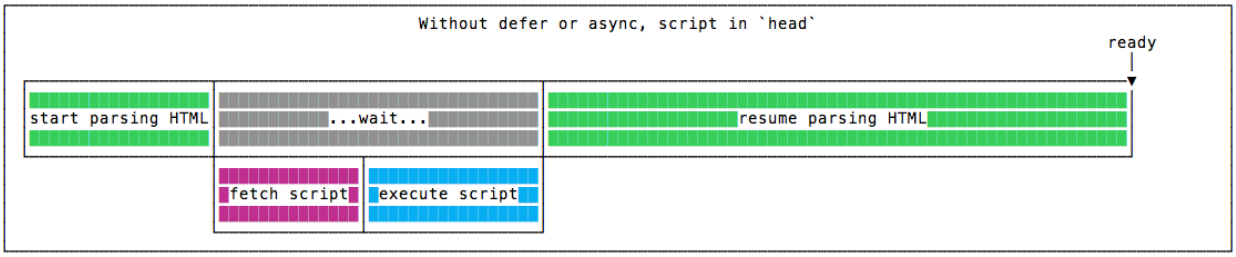
if you specify both, `async` takes precedence on modern browsers, while older browsers that support `defer` but not `async` will fallback to `defer`.

> For the support table, check caniuse.com for async https://caniuse.com/#feat=script-async and for defer https://caniuse.com/#feat=script-defer

These attributes only make sense when using the script in the `head` portion of the page, and they are useless if you put the script in the `body` footer like we saw above.

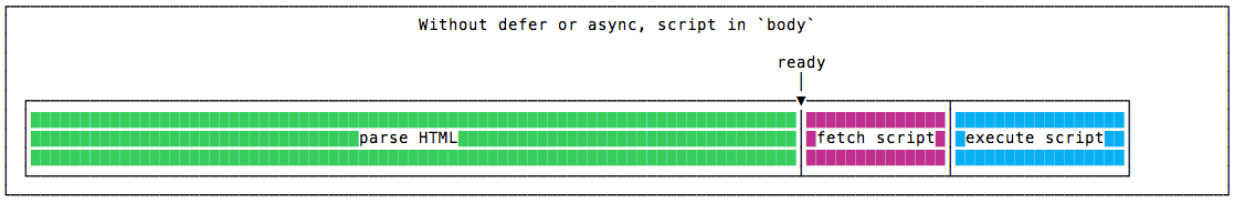
Performance comparison

No defer or async, in the head Here's how a page loads a script without either defer or async, put in the `head` portion of the page:



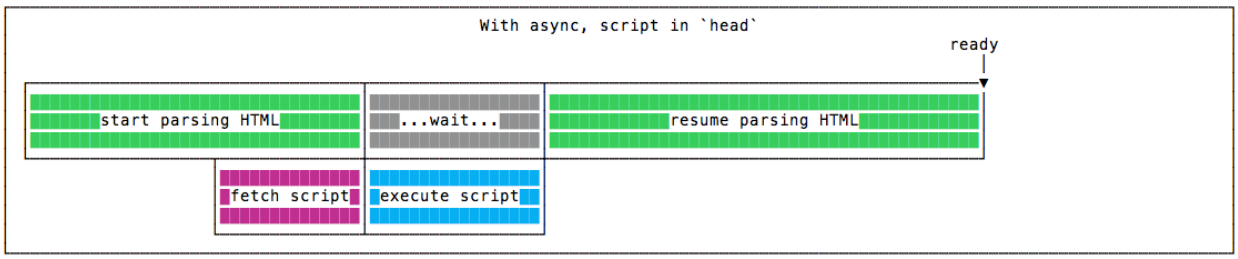
The parsing is paused until the script is fetched, and executed. Once this is done, parsing resumes.

No defer or async, in the body Here's how a page loads a script without defer or async, put at the end of the `body` tag, just before it closes:



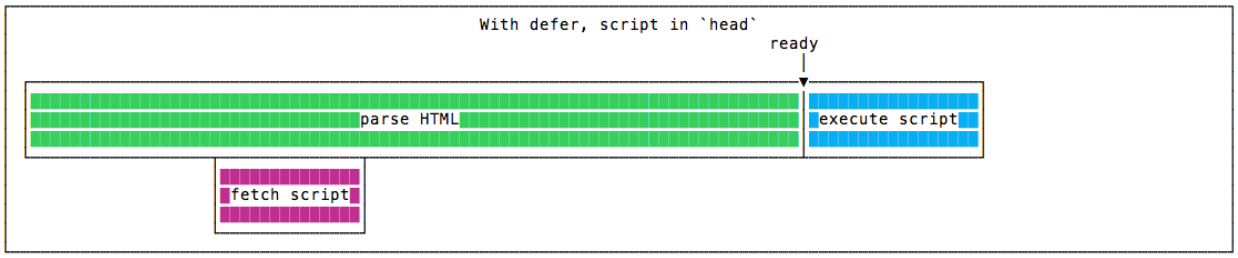
The parsing is done without any pauses, and when it finishes, the script is fetched, and executed. Parsing is done before the script is even downloaded, so the page appears to the user way before the previous example.

With async, in the head Here's how a page loads a script with `async`, put in the `head` tag:



The script is fetched asynchronously, and when it's ready the HTML parsing is paused to execute the script, then it's resumed.

With defer, in the head Here's how a page loads a script with `defer`, put in the `head` tag:



The script is fetched asynchronously, and it's executed only after the HTML parsing is done.

Parsing finishes just like when we put the script at the end of the `body` tag, but overall the script execution finishes well before, because the script has been downloaded in parallel with the HTML parsing.

So this is the winning solution in terms of speed

Blocking parsing

`async` blocks the parsing of the page while `defer` does not.

Blocking rendering

Neither `async` nor `defer` guarantee anything on blocking rendering. This is up to you and your script (for example, making sure your scripts run after the `onLoad`) event.

domInteractive

Scripts marked `defer` are executed right after the `domInteractive` event, which happens after the HTML is loaded, parsed and the DOM is built.

CSS and images at this point are still to be parsed and loaded.

Once this is done, the browser will emit the `domComplete` event, and then `onLoad`.

`domInteractive` is important because its timing is recognized as a measure of perceived loading speed. See the MDN for more.

Keeping things in order

Another case pro `defer`: scripts marked `async` are executed in casual order, when they become available. Scripts marked `defer` are executed (after parsing completes) in the order which they are defined in the markup.

Just tell me the best way

The best thing to do to speed up your page loading when using scripts is to put them in the `head`, and add a `defer` attribute to your `script` tag:

```
<script defer src="script.js"></script>
```

This is the scenario that triggers the faster `domInteractive` event.

Considering the pros of `defer`, is seems a better choice over `async` in a variety of scenarios.

Unless you are fine with delaying the first render of the page, make sure that when the page is parsed the JavaScript you want is already executed.

How module scripts load

Understand the default deferred behavior of module scripts and how their dependency graph changes loading.

A module script is deferred by default:

```
<script type="module" src="/app.js"></script>
```

The browser can continue parsing HTML while it fetches the entry module. It then follows static `import` statements and fetches the dependency graph before evaluating the entry.

Module scripts execute after document parsing. Adding `defer` does not change this default behavior.

The graph matters for performance. A small entry file can still uncover many dependent requests. Deep chains delay the module that depends on their completion.

Dynamic `import()` is different. The browser discovers that module when execution reaches the call, which can reduce initial work but delay the feature using it.

Use the Network panel Initiator chain to follow module requests. In the Performance panel, compare download completion with evaluation time; network completion does not guarantee that the main thread was ready to run the code.

Exercise: create an entry module importing two small files. Reload and trace the dependency graph in DevTools.

Preload important resources carefully

Use resource hints to reveal a genuinely important download without preloading everything.

`preload` tells the browser about a resource needed soon:

```
<link rel="preload" href="https://flaviocopes.com/hero.webp" as="image">
```

The `as` value gives the browser the resource destination. It affects request behavior, priority, and policy checks. Fonts and some cross-origin resources also need matching CORS settings.

Preload only resources that the current page will soon use. An unnecessary preload competes for bandwidth with more important work and may be downloaded without being consumed.

Preload is useful when normal discovery is late, such as a critical font referenced from CSS or an image hidden behind another resource. It does not make a large file cheap or fix a slow server.

Measure the request order in the Network panel before adding a hint. Then add one preload, reload under the same throttling, and compare the resource start time and the page's visible result.

DevTools warns when a preloaded resource is not used soon after load. Treat that as evidence to review the hint.

Exercise: identify one late-discovered critical resource. Explain why changing markup might be better than adding a preload.

The HTTP/2 protocol

Learn how HTTP/2 uses streams, multiplexing, HPACK, flow control, and binary framing, how it is negotiated, and how it differs from HTTP/3.

I suggest reading the HTTP tutorial first.

HTTP/2 is one of the HTTP protocol versions used on the web.

It was first standardized in 2015 as RFC 7540. That document was replaced in 2022 by RFC 9113, which contains the current HTTP/2 specification.

HTTP/3 is also standardized and deployed, while HTTP/1.1 is still in use. These versions coexist. A client and server use a version they both support.

HTTP/2 keeps the same HTTP semantics: methods, status codes, URLs, header fields, and message content still mean the same things. What changes is how those messages are framed and transported.

Its main features are:

- streams and request/response multiplexing
- binary framing
- HPACK header compression
- per-stream and connection-level flow control
- stream prioritization
- optional server push

How HTTP/2 is negotiated

For HTTPS connections, the client and server normally select HTTP/2 during the TLS handshake using ALPN. The protocol identifier is `h2`.

If HTTP/2 is not available, they can negotiate another supported protocol, such as HTTP/1.1. This fallback is why enabling HTTP/2 on a server can be transparent to application code, but HTTP/2 is not simply "100% backwards compatible" at the wire-protocol level.

The specification also defines cleartext HTTP/2, commonly called `h2c`, but browsers normally use HTTP/2 over TLS for public websites.

Streams and multiplexing

An HTTP/2 connection contains multiple independent streams. Each request and response exchange uses its own stream, and frames from several streams can be interleaved on the same TCP connection.

With HTTP/1.1, browsers commonly opened several TCP connections because a slow response could delay other work on the same connection. HTTP/2 multiplexing allows many requests to make progress concurrently without needing one connection per request.

This made old workarounds such as domain sharding counterproductive in many HTTP/2 deployments. Combining files only to reduce the number of requests is also less important than it was with HTTP/1.1, although bundle size, caching, compression, and processing cost still matter.

Multiplexing does not make every site automatically faster. All HTTP/2 streams share one TCP connection, so packet loss can still stall data for every stream while TCP recovers the missing bytes. This is transport-level head-of-line blocking.

Binary framing

HTTP/2 divides messages into binary frames.

For example, `HEADERS` frames carry compressed header fields and `DATA` frames carry message content. Frames include a stream identifier so the receiver can put interleaved data back into the correct

request or response.

”Binary framing” does not mean that HTML, JSON, or other message content changes format. It also does not provide encryption by itself. TLS provides encryption for HTTPS.

Header compression with HPACK

HTTP requests and responses often repeat the same field names and values. Cookies can make those fields especially large.

HTTP/2 uses HPACK, defined by RFC 7541, to compress header fields. It can reuse entries from a table shared by the encoder and decoder instead of sending the same text with every request.

Flow control and prioritization

Flow control prevents a sender from overwhelming the receiver. HTTP/2 applies it both to individual streams and to the connection as a whole.

Prioritization lets an endpoint communicate which streams are more important. The original priority scheme from RFC 7540 was deprecated by RFC 9113, and newer extensible priorities are defined separately in RFC 9218.

What happened to server push?

HTTP/2 server push lets a server send a response that it predicts the client will need before the client makes that request.

It remains an optional part of the specification, but it is difficult to use efficiently. The server can waste bandwidth by pushing something already cached or something the page never uses.

Chrome removed HTTP/2 server push support in version 106 after finding little use and no clear net performance gain. For browser performance work, preload links and `103 Early Hints` are more practical options.

HTTP/2 and HTTP/3

HTTP/3 is no longer experimental. It was standardized in 2022 as RFC 9114.

HTTP/2 runs over TCP. HTTP/3 maps HTTP semantics onto QUIC, which runs over UDP and includes transport security.

QUIC provides independent streams at the transport layer. Packet loss affecting one HTTP/3 stream does not block unrelated streams in the same way it can on a shared HTTP/2 TCP connection.

HTTP/3 uses QPACK instead of HPACK because QUIC streams can deliver data independently.

HTTP/2 remains useful and widely deployed. HTTP/3 improves the transport, but it does not make HTTP/2 obsolete overnight.

Reuse connections and cached responses

Understand why later requests to the same origin can avoid repeated setup and sometimes avoid transferring the body entirely.

A first request may need DNS, transport setup, and a TLS handshake. Later requests can often reuse an existing connection instead of repeating all that work.

HTTP/2 and HTTP/3 can carry several requests over one connection. Reuse reduces setup cost, but server limits, network changes, and connection policies still affect what happens.

Caching can avoid more work. A fresh cached response may be used without contacting the server. A stale response can be revalidated and return a small `304 Not Modified` response instead of another body.

This makes a repeat visit different from a first visit. Test both cases deliberately.

In the Network panel, reload once with the cache disabled and once with normal caching. Compare transferred size, status, and Timing details. Do not treat a warm local reload as representative of a new visitor.

Good cache policy helps stable versioned assets. HTML and frequently changing API data often need different rules.

Exercise: choose one stylesheet and compare its first and second request. Record whether the browser transferred the body again.

Check your understanding: networking and loading

Check DNS, encrypted connections, parser blocking, preload scanning, script attributes, modules, priorities, and connection reuse.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The rendering pipeline

DOM, CSSOM, style, layout, paint, and compositing.

From bytes to pixels

See the full path from parsed HTML and CSS through style, layout, paint, and compositing.

The browser receives bytes, but the screen needs pixels. The rendering pipeline connects them.

HTML bytes become tokens and DOM nodes. CSS becomes rules the browser can match against those nodes.

The browser then performs several kinds of work:

1. **Style calculation** decides which computed values apply.
2. **Layout** calculates box sizes and positions.
3. **Paint** records visual instructions for text, backgrounds, borders, and images.
4. **Rasterization** turns those instructions into pixels.
5. **Compositing** combines layers into the frame shown on screen.

These stages are dependencies, not a promise that every frame repeats all five. A text change may need style, layout, and paint. A suitable transform animation may need only compositing after initial setup.

JavaScript runs on the main thread with much of style and layout work. A long task can therefore delay pixels even when every network request has finished.

Record a page load in the Performance panel. Find Parse HTML, Recalculate Style, Layout, Paint, and Composite-related activity. Your browser version may label or group events differently, but the causal work remains.

Exercise: change an element's width, background, and transform separately. Predict the stages each change needs before recording them.

The Document Object Model (DOM)

Learn what the DOM (Document Object Model) is: the browser's representation of a page as nodes, and how to traverse and edit it with window and document.

The **Document Object Model**, or **DOM**, represents an HTML document as a tree of objects in memory. JavaScript can use this tree to inspect and change the page.

When the browser parses HTML, it creates nodes for the document, elements, text, and comments. The browser exposes APIs to read and change those nodes.

This is also how modern JavaScript frameworks update the page. You can inspect the current DOM using the Browser Developer Tools.

The DOM is not part of the JavaScript language. It is a Web API that browsers make available to JavaScript.

> The DOM is standardized by WHATWG in the DOM Living Standard Spec.

With JavaScript you can interact with the DOM to:

- inspect the page structure
- access page metadata
- edit the CSS styling
- attach or remove event listeners
- edit any node in the page
- change any node attribute

.. and much more.

The two objects you will use most often are `document` and `window`.

The Window object

The `window` object represents the window that contains the DOM document.

`window.document` points to the `document` object loaded in the window.

Properties and methods of this object can be called without referencing `window` explicitly, because it represents the global object. So, the previous property `window.document` is usually called just `document`.

Properties Here is a list of useful properties you will likely reference a lot:

- `console` points to the browser debugging console. Useful to print error messages or logging, using `console.log`, `console.error` and other tools (see the Browser DevTools article)
- `document` as already said, points to the `document` object, key to the DOM interactions you will perform
- `history` gives access to the **History API**

- `location` gives access to the Location interface, from which you can determine the URL, the protocol, the hash and other useful information.
- `localStorage` is a reference to the Web Storage API localStorage object
- `sessionStorage` is a reference to the Web Storage API sessionStorage object

Methods The `window` object also exposes useful methods:

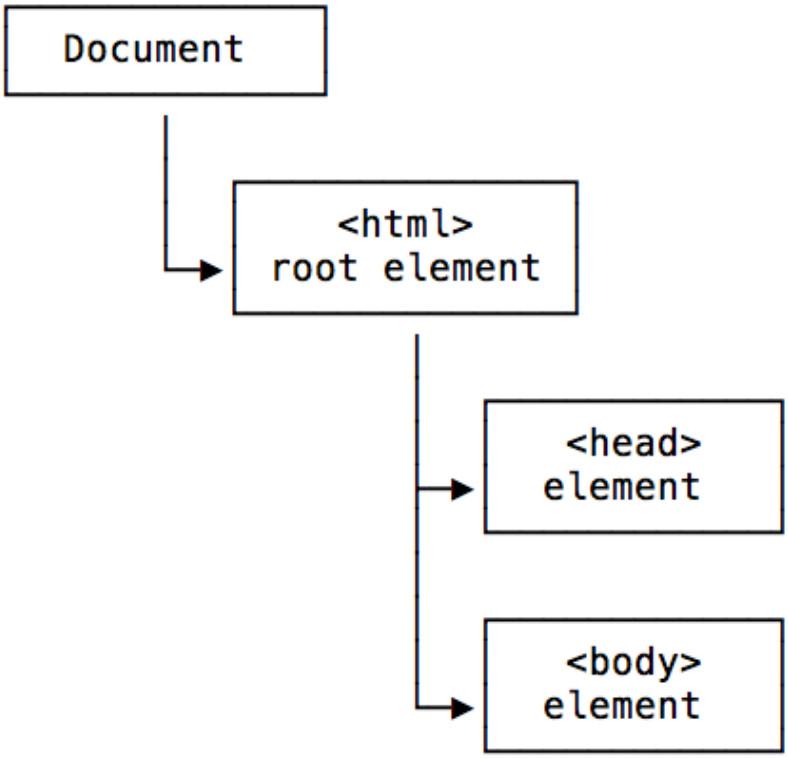
- `alert()`: which you can use to display alert dialogs
- `postMessage()`: sends messages between windows
- `requestAnimationFrame()`: used to perform animations in a way that's both performant and easy on the CPU
- `setInterval()`: call a function every n milliseconds, until the interval is cleared with `clearInterval()`
- `clearInterval()`: clears an interval created with `setInterval()`
- `setTimeout()`: execute a function after 'n' milliseconds
- `addEventListener()`: add an event listener to the window
- `removeEventListener()`: remove an event listener from the window

See the full reference of all the properties and methods of the `window` object at https://developer.mozilla.org/en-US/docs/Web/API/Window

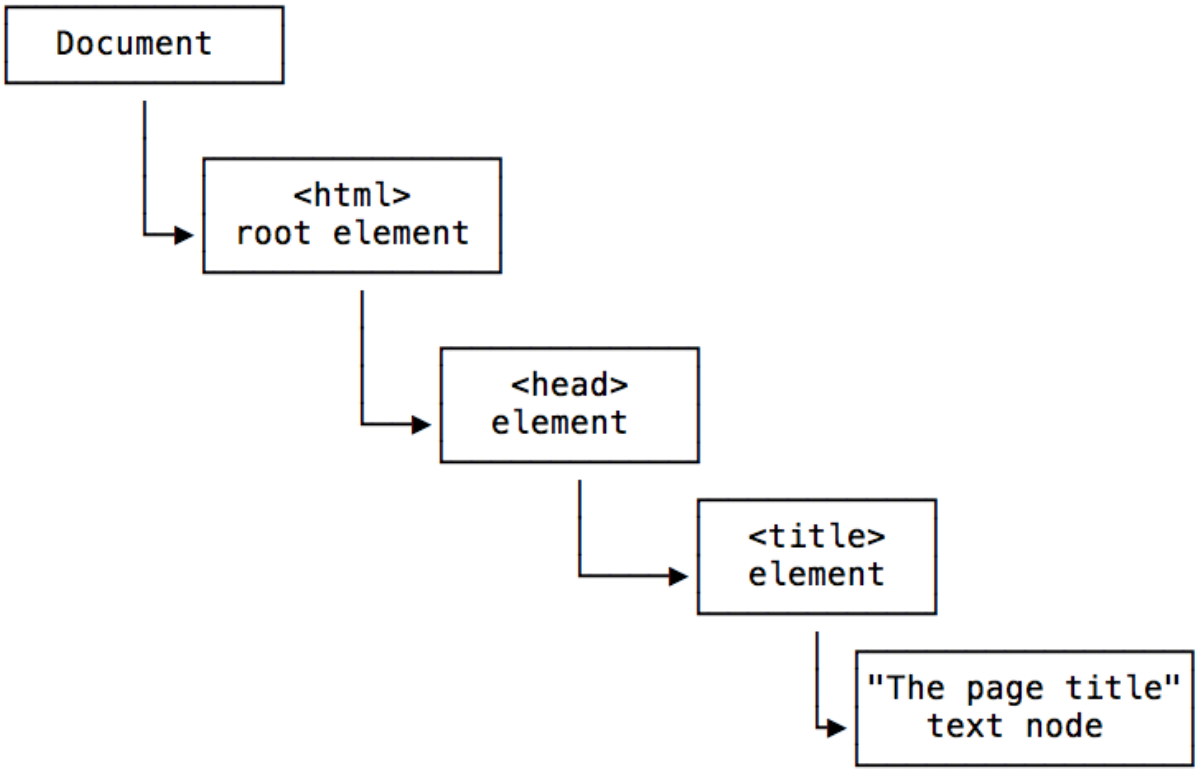
The Document object

The `document` object represents the DOM tree loaded in a window.

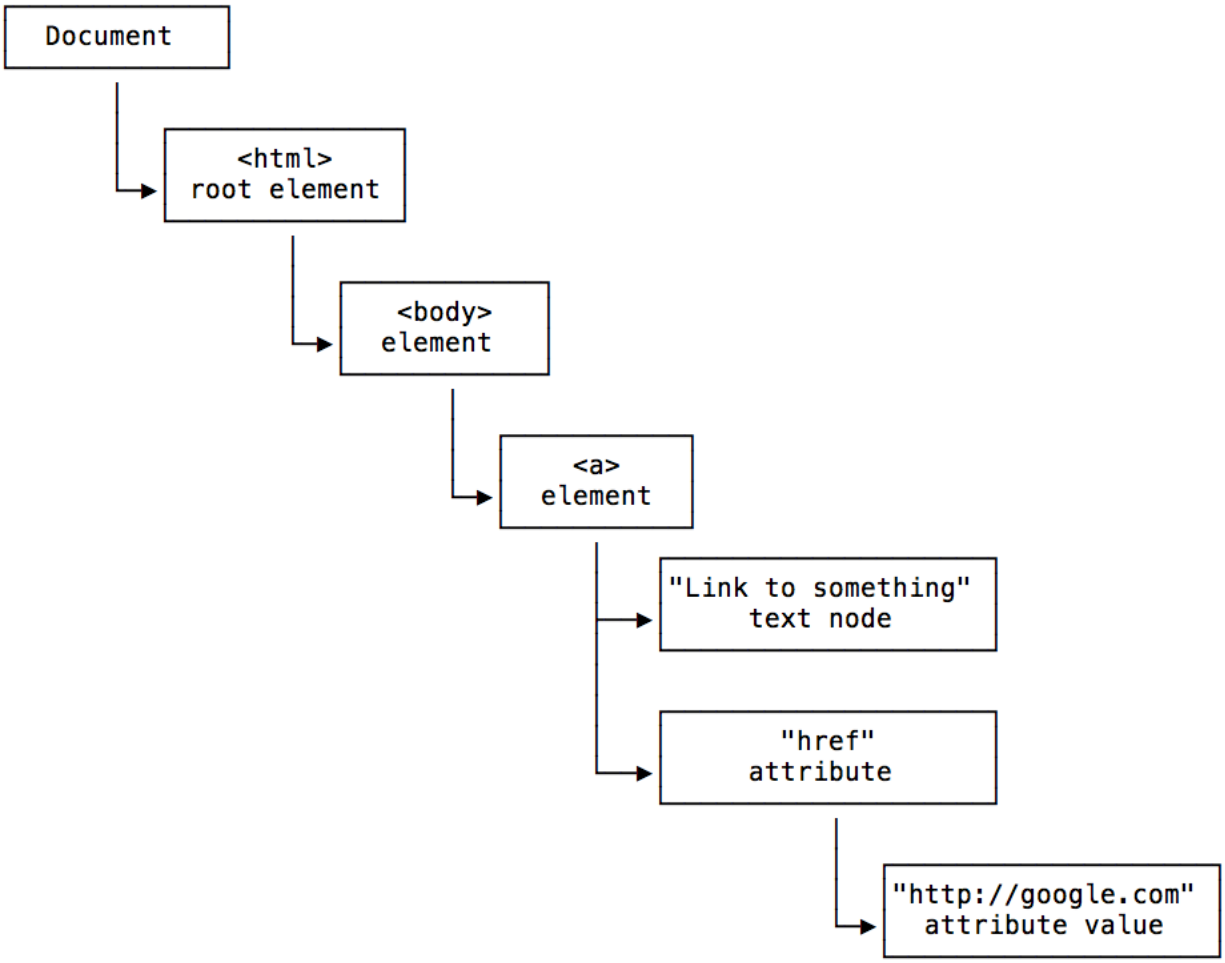
Here is a representation of a portion of the DOM pointing to the head and body tags:



Here is a representation of a portion of the DOM showing the head tag, containing a title tag with its value:



Here is a representation of a portion of the DOM showing the body tag, containing a link, with a value and the href attribute with its value:



The Document object can be accessed from `window.document`, and since `window` is the global object, you can use the shortcut `document` object directly from the browser console, or in your JavaScript code.

This Document object has a ton of properties and methods. The Selectors API methods are the ones you'll likely use the most:

- `document.getElementById()`
- `document.querySelector()`
- `document.querySelectorAll()`
- `document.getElementsByTagName()`
- `document.getElementsByClassName()`

You can get the document title using `document.title`, the URL using `document.URL`, and the referrer using `document.referrer`.

From the `document` object you can get the body and head Element nodes:

- `document.documentElement`: the root `html` Element
- `document.body`: the `body` Element node
- `document.head`: the `head` Element node

```
> document.documentElement
<- <html prefix="og: http://ogp.me/ns#" lang="en" class="gr__localhost">
     ▶<head>…</head>
     ▶<body data-gr-c-s-loaded="true">…</body>
     </html>
> document.head
<- ▶<head>…</head>
> document.body
<- ▶<body data-gr-c-s-loaded="true">…</body>
```

You can also get a list of all the element nodes of a particular type, like an HTMLCollection of all the links using `document.links`, all the images using `document.images`, all the forms using `document.forms`.

The document cookies are accessible in `document.cookie`. The last modified date in `document.lastModified`.

Avoid `document.write()`. It can replace the current document and cause serious performance problems. Use the DOM methods below instead.

See the full reference of all the properties and methods of the `document` object at https://developer.mozilla.org/en-US/docs/Web/API/Document

Types of Nodes

There are different types of nodes, some of which you have already seen in the example images above. The main ones you will encounter are:

- **Document**: the document Node, the start of the tree
- **Element**: an HTML tag
- **Attr**: an element attribute, represented as a node but not stored as a child in the main tree
- **Text**: text inside an Element

- **Comment**: an HTML comment
- **DocumentType**: the Doctype declaration

Traversing the DOM

The DOM is a tree of elements, with the Document node at the root, which points to the `html` Element node, which in turn points to its child element nodes `head` and `body`, and so on.

From each of those elements, you can navigate the DOM structure and move to different nodes.

Getting the parent Every element has *just one* parent.

To get it, you can use `Node.parentNode` or `Node.parentElement` (where Node means a node in the DOM).

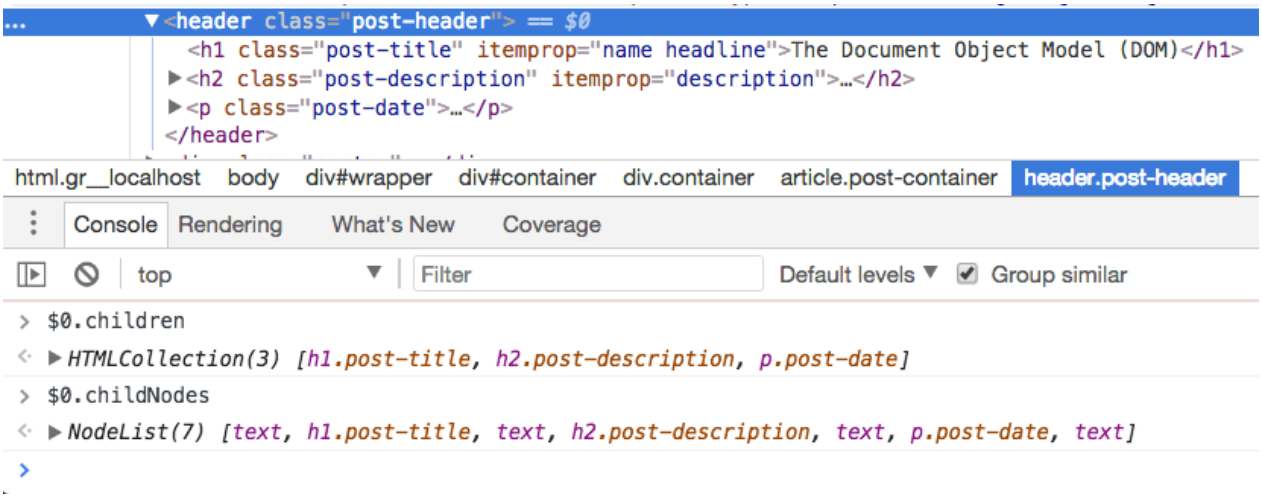
They are almost the same. `parentNode` returns any kind of parent node. `parentElement` returns an Element, or `null` when the parent is not an Element.

Use `parentElement` when you specifically need an element.

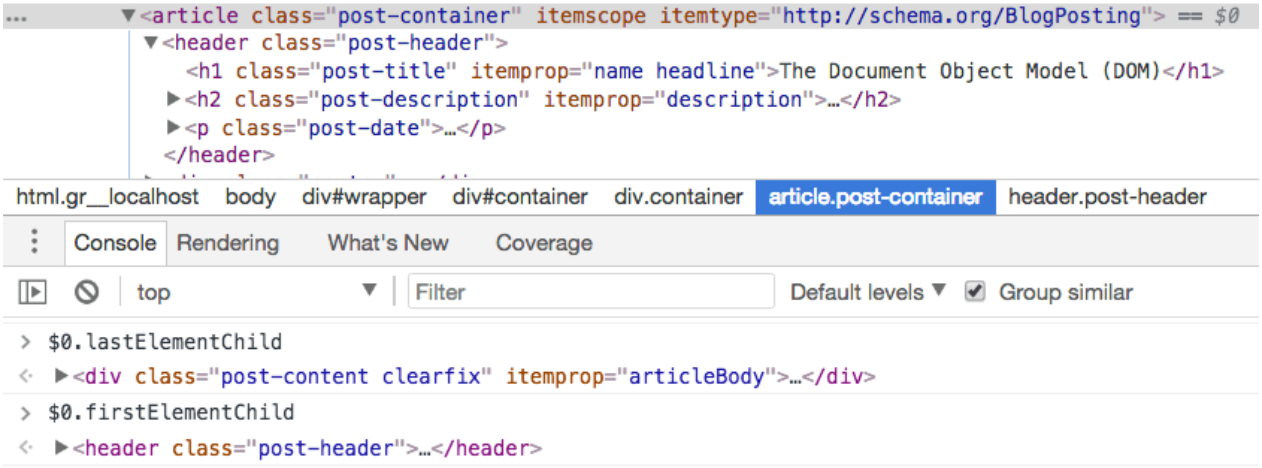
Getting the children To check if a Node has child nodes, use **`Node.hasChildNodes()`** which returns a boolean value.

To access all child nodes, including text and comments, use **`node.childNodes`**.

To access only child elements, use **`element.children`**. It returns an `HTMLCollection` and ignores whitespace text nodes. The MDN DOM anatomy guide shows this distinction.



To get the first child Element, use **`element.firstElementChild`**. To get the last child Element, use **`element.lastElementChild`**:



The DOM also exposes `node.firstChild` and `node.lastChild`. These do not filter for Element nodes, so they can return whitespace Text nodes.

In short, to navigate child elements use

- `element.children`
- `element.firstElementChild`
- `element.lastElementChild`

Getting the siblings In addition to getting the parent and the children, since the DOM is a tree you can also get the siblings of any Element Node.

You can do so using

- **`element.previousElementSibling`**
- **`element.nextElementSibling`**

The DOM also exposes `previousSibling` and `nextSibling`, but as their counterparts described above, they include white spaces as Text nodes, so you generally avoid them.

Editing the DOM

The DOM offers various methods to edit the nodes of the page and alter the document tree.

You can create nodes with:

- `document.createElement()`: creates a new Element Node
- `document.createTextNode()`: creates a new Text Node

Then add them to another element using `appendChild()`:

```
const div = document.createElement('div')
div.appendChild(document.createTextNode('Hello world!'))
document.body.appendChild(div)
```

- `parent.removeChild(child)` removes a child node
- `parent.insertBefore(newNode, existingNode)` inserts a node before another child
- `element.appendChild(newChild)` adds a node after the existing children
- `element.prepend(newChild)` adds a node before the existing children
- `element.replaceChild(newChild, existingChild)` replaces a child node
- `element.insertAdjacentElement(position, newElement)` inserts an element at a specified position. See the possible values

- `element.textContent = 'Hello'` replaces an element's text content

HTML parsing is incremental

Understand how the parser builds the DOM as bytes arrive and why some scripts can pause that process.

The browser does not normally wait for the complete HTML response before building the DOM.

It decodes incoming bytes, tokenizes markup, and creates nodes incrementally. This lets resource discovery and rendering begin while more HTML is arriving.

A classic script can pause the parser:

```
<script src="/app.js"></script>
```

The script may call `document.write()` or inspect the DOM at that exact location. The browser must fetch and execute it before continuing the main parse.

`defer` lets a classic external script download during parsing and run after the document is parsed. `async` runs when ready and does not preserve document order with other async scripts.

Open the Network and Performance panels, throttle the network, and compare a blocking script with the same script using `defer`. Look for the parser pause and the later `DOMContentLoaded` timing.

Exercise: put visible text before and after a delayed classic script. Observe when each part appears.

Build the CSSOM and calculate styles

Understand why CSS must be parsed before the browser can know which rules and inherited values apply to each element.

The browser parses CSS into rules it can match against the document.

For each relevant element, it resolves selectors, the cascade, inheritance, custom properties, and computed values. A declared width such as `50%` may still need layout before it becomes a pixel size.

External stylesheets normally block rendering because the browser needs their rules before painting a stable result. HTML parsing can continue, but the first styled frame may wait.

Not every CSS rule matches every element. Large stylesheets and frequent DOM changes can increase style-calculation work, but selector folklore is not a substitute for a recording.

Use the Elements panel's Computed view to trace a final value back to its declarations. Then use the Performance panel to find Recalculate Style activity after changing a class.

Exercise: create two rules with different specificity, inspect the winning computed value, and explain how the cascade produced it.

The render tree

Distinguish the document tree from the boxes that participate in visual rendering.

The DOM describes document nodes. Rendering needs the styled boxes and visual content that can appear on screen.

Not every DOM node creates a box. An element with `display: none` remains in the DOM but does not participate in layout or paint.

The reverse is also possible. `::before` and `::after` can create visual content without separate DOM nodes.

You will often hear **render tree** used as the conceptual combination of document content and computed styles. Browser engines use several internal structures, so do not depend on one literal universal tree implementation.

Visibility also matters differently. `visibility: hidden` keeps an element's layout space but skips its visible painting. `display: none` removes its box from layout.

Use the Elements panel to toggle both properties. Watch neighboring boxes move for `display: none` but keep their positions for `visibility: hidden`.

Exercise: predict whether a hidden element affects layout, paint, both, or neither for each property.

Layout calculates geometry

Understand how the browser determines the size and position of boxes and why one geometry change can affect many descendants or neighbors.

Layout turns computed styles and content into geometry.

The browser calculates each participating box's size and position. Available width, writing mode, box model, fonts, intrinsic image sizes, and layout algorithms all contribute.

Geometry is connected. Changing a container width can affect line wrapping, child sizes, and every box below it. The browser may recalculate a small subtree or a much larger part of the page.

JavaScript can force layout earlier than the browser planned. If code writes a style and immediately reads `offsetWidth`, the browser may need current geometry before returning the value.

Use the Performance panel to find Layout events. Select one and inspect the affected nodes or initiator information available in your browser version.

Small experiment:

```
const card = document.querySelector('.card')
card.style.width = '400px'
console.log(card.offsetWidth)
```

Record this code and look for synchronous layout between the write and read.

Paint and rasterization

Understand how backgrounds, text, borders, shadows, and images become drawing instructions and pixels.

After layout knows geometry, paint records what should appear inside those boxes.

Text, backgrounds, borders, shadows, and images become drawing instructions. Their order matters because later content can cover earlier content.

Rasterization turns those instructions into pixels. Browsers often rasterize tiles and layers rather than one giant page bitmap.

Changing a background color usually keeps geometry but invalidates painted content. A large shadow or full-page background can make the paint area much larger than the changed element suggests.

In Chrome DevTools, enable paint flashing from the Rendering tools. Change a background color and watch the repainted region. Then record the same action and inspect Paint events.

Paint cost depends on area and visual complexity, not only JavaScript time. Reducing a handler from 2 ms to 1 ms will not help if it triggers an expensive full-page repaint.

Exercise: animate `background-color`, then animate `transform`. Compare paint flashing and the Performance recording.

Compositing layers

Understand how painted layers are positioned and combined into the final frame.

Compositing places painted layers in the correct order and produces the final frame.

Some content gets its own composited layer. The browser can move or fade a suitable layer without repainting its contents, which is why `transform` and `opacity` are often good animation properties.

This is not automatic magic. Layer creation depends on browser decisions, element properties, and surrounding content. A transform can still cause other work during setup or when its contents change.

Layers also consume memory for rasterized content and require management. Adding `will-change` everywhere can make performance worse.

Use the Performance panel's frame and rendering information to compare an animation of `left` with one using `transform`. Paint flashing can show whether frames repaint.

My advice is to express the visual effect clearly, record it, and let evidence guide layer tuning.

Exercise: animate one card both ways under CPU throttling. Compare dropped frames, Layout events, and Paint events.

What a DOM or CSS change triggers

Predict whether a change needs style calculation, layout, paint, compositing, or several stages.

A DOM or CSS change invalidates work the browser previously completed. The next required stage depends on what became stale.

Changing text, width, or font metrics can affect geometry. The browser may need style calculation, layout, paint, and compositing.

Changing a background normally preserves geometry but needs paint. Moving a suitable existing layer with `transform` can often skip layout and repainting.

Removing a class can be more expensive than the changed line suggests if it changes descendants or a large layout container.

Treat these as predictions, not guarantees. Browser engines optimize invalidation, and surrounding page structure matters.

Make three buttons that change `width`, `background-color`, and `transform`. Record one click at a time in the Performance panel. Use paint flashing as a second source of evidence.

For every change, write down:

- which stages you predicted
- which events DevTools recorded

- which part of the page was affected

Performance work starts by connecting a source change to measured browser work.

Check your understanding: the rendering pipeline

Check incremental parsing, CSS blocking, render-tree participation, layout, paint, compositing, and the work caused by common style changes.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The JavaScript engine

Parsing, optimization, the stack, event loop, and memory.

The V8 JavaScript Engine

Understand V8, the JavaScript engine that powers Google Chrome and Node.js, how it provides the runtime, and how it uses just-in-time compilation for speed.

V8 is the name of the JavaScript engine that powers Google Chrome. It's the thing that takes our JavaScript and executes it while browsing with Chrome.

V8 provides the runtime environment in which JavaScript executes. The DOM, and the other Web Platform APIs are provided by the browser.

The cool thing is that the JavaScript engine is independent by the browser in which it's hosted. This key feature enabled the rise of Node.js. V8 was chosen to be the engine that powered Node.js back in 2009, and as the popularity of Node.js exploded, V8 became the engine that now powers an incredible amount of server-side code written in JavaScript.

The Node.js ecosystem is huge and thanks to it V8 also powers desktop apps, with projects like Electron.

Other JS engines

Other browsers have their own JavaScript engine:

- Firefox has **Spidermonkey**
- Safari has **JavaScriptCore** (also called Nitro)
- Edge has **Chakra**

and many others exist as well.

All those engines implement the ECMA ES-262 standard, also called ECMAScript, the standard used by JavaScript.

The quest for performance

V8 is written in C++, and it's continuously improved. It is portable and runs on Mac, Windows, Linux and several other systems.

In this V8 introduction, I will ignore the implementation details of V8: they can be found on more authoritative sites (e.g. the V8 official site), and they change over time, often radically.

V8 is always evolving, just like the other JavaScript engines around, to speed up the Web and the Node.js ecosystem.

On the web, there is a race for performance that's been going on for years, and we (as users and developers) benefit a lot from this competition because we get faster and more optimized machines year after year.

Compilation

JavaScript is generally considered an interpreted language, but modern JavaScript engines no longer just interpret JavaScript, they compile it.

This happens since 2009 when the SpiderMonkey JavaScript compiler was added to Firefox 3.5, and everyone followed this idea.

JavaScript is internally compiled by V8 with **just-in-time** (JIT) **compilation** to speed up the execution.

This might seem counter-intuitive, but since the introduction of Google Maps in 2004, JavaScript has evolved from a language that was generally executing a few dozens of lines of code to complete applications with thousands to hundreds of thousands of lines running in the browser.

Our applications now can run for hours inside a browser, rather than being just a few form validation rules or simple scripts.

In this *new world*, compiling JavaScript makes perfect sense because while it might take a little bit more to have the JavaScript *ready*, once done it's going to be much more performant that purely interpreted code.

Parse, compile, and run JavaScript

Follow source code through parsing, bytecode generation, execution, and later optimization.

The browser cannot run JavaScript source directly. The engine first parses the text and turns it into an internal representation of the program.

This is why a syntax error stops the affected script before its statements run:

```
console.log('before')

function broken( {

console.log('after')
```

Neither log runs. Parsing the script failed, so there was no valid program to execute.

After parsing, an engine can produce bytecode or machine code and start executing it. Frequently executed code may later receive a more specialized compiled version. The exact pipeline differs between engines and changes over time, so application code should not depend on one engine's internal compiler names.

What matters for page performance is the work that reaches the main thread:

1. Download the script.

2. Parse and compile it.
3. Execute top-level code.
4. Run functions later when events, timers, or promises invoke them.

A small compressed file can still be expensive if it contains a lot of code to parse or executes substantial work immediately.

Open DevTools, record a page load in the Performance panel, and inspect the main-thread track. Expand a script-related task and distinguish evaluation from later function calls. Then disable that script temporarily and record again. This connects the source file to actual main-thread cost instead of judging it only by download size.

Optimization and deoptimization

Understand why stable runtime behavior can help optimized code and why an engine may discard an incorrect optimization.

JavaScript is dynamic: the same function can receive numbers, strings, or objects with different properties. An engine must preserve that behavior, but it may generate faster code when runtime observations stay predictable.

Consider this function:

```
function total(price, quantity) {
  return price * quantity
}
```

If a hot call site repeatedly passes numbers, the engine may compile a specialized path guarded by assumptions about those values. If a later call passes something different, the guard fails. The engine falls back to more general code or compiles another version. That transition is commonly called deoptimization.

Deoptimization is a correctness mechanism, not an application error. The dangerous conclusion is that every changing type must be rewritten. Most functions are not hot enough for this detail to matter, and network, rendering, or algorithmic work often dominates instead.

Use this order when investigating slow JavaScript:

1. Record the real interaction.
2. Find a function with meaningful total time.
3. Check whether the algorithm repeats avoidable work.
4. Only then investigate engine-specific optimization behavior.

Microbenchmarks can mislead because they warm up one tiny function under artificial conditions. Test the complete user action and confirm that a code change improves its recording, not just an isolated loop.

The call stack

Track nested function calls and understand why the most recently called function returns first.

Each active function call has a stack frame containing information such as its local variables and where execution should return.

```
function formatName(name) {
  return name.trim().toUpperCase()
```

```
}

function renderUser(user) {
  return `<p>${formatName(user.name)}</p>`
}

renderUser({ name: ' Ada ' })
```

The stack grows in this order:

1. the top-level script
2. `renderUser()`
3. `formatName()`

`formatName()` returns first, then `renderUser()`. The last function called is the first one removed, so the stack is last-in, first-out.

This model explains two practical debugging tools. An error stack trace shows the chain of calls that led to the error. A breakpoint's Call Stack pane lets you select an earlier frame and inspect the values that caller passed.

It also explains stack overflow. Recursion must eventually reach a base case:

```
function countdown(value) {
  if (value === 0) return
  countdown(value - 1)
}
```

Remove the base case and frames keep accumulating until the engine refuses another call.

Set a breakpoint inside `formatName()`, reload the page, and inspect the Call Stack pane. Step out once and watch the `formatName` frame disappear while `renderUser` becomes active again.

The JavaScript Event Loop

The Event Loop is one of the most important aspects to understand about JavaScript. This post explains it in simple terms

Introduction

The **Event Loop** is one of the most important aspects to understand about JavaScript.

> I've programmed for years with JavaScript, yet I've never *fully* understood how things work under the hoods. It's completely fine to not know this concept in detail, but as usual, it's helpful to know how it works, and also you might just be a little curious at this point.

This post aims to explain the inner details of how JavaScript works with a single thread, and how it handles asynchronous functions.

Your JavaScript code runs single threaded. There is just one thing happening at a time.

This is a limitation that's actually very helpful, as it simplifies a lot how you program without worrying about concurrency issues.

You just need to pay attention to how you write your code and avoid anything that could block the thread, like synchronous network calls or infinite loops.

In general, in most browsers there is an event loop for every browser tab, to make every process isolated and avoid a web page with infinite loops or heavy processing to block your entire browser.

The environment manages multiple concurrent event loops, to handle API calls for example. Web Workers run in their own event loop as well.

You mainly need to be concerned that *your code* will run on a single event loop, and write code with this thing in mind to avoid blocking it.

Blocking the event loop

Any JavaScript code that takes too long to return back control to the event loop will block the execution of any JavaScript code in the page, even block the UI thread, and the user cannot click around, scroll the page, and so on.

Almost all the I/O primitives in JavaScript are non-blocking. Network requests, Node.js filesystem operations, and so on. Being blocking is the exception, and this is why JavaScript is based so much on callbacks, and more recently on promises and async/await.

The call stack

The call stack is a LIFO queue (Last In, First Out).

The event loop continuously checks the **call stack** to see if there's any function that needs to run.

While doing so, it adds any function call it finds to the call stack and executes each one in order.

You know the error stack trace you might be familiar with, in the debugger or in the browser console? The browser looks up the function names in the call stack to inform you which function originates the current call:

A simple event loop explanation

Let's pick an example:

I use `foo`, `bar` and `baz` as *random names*. Enter any kind of name to replace them

```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')
  bar()
  baz()
}
```

```
foo()
```

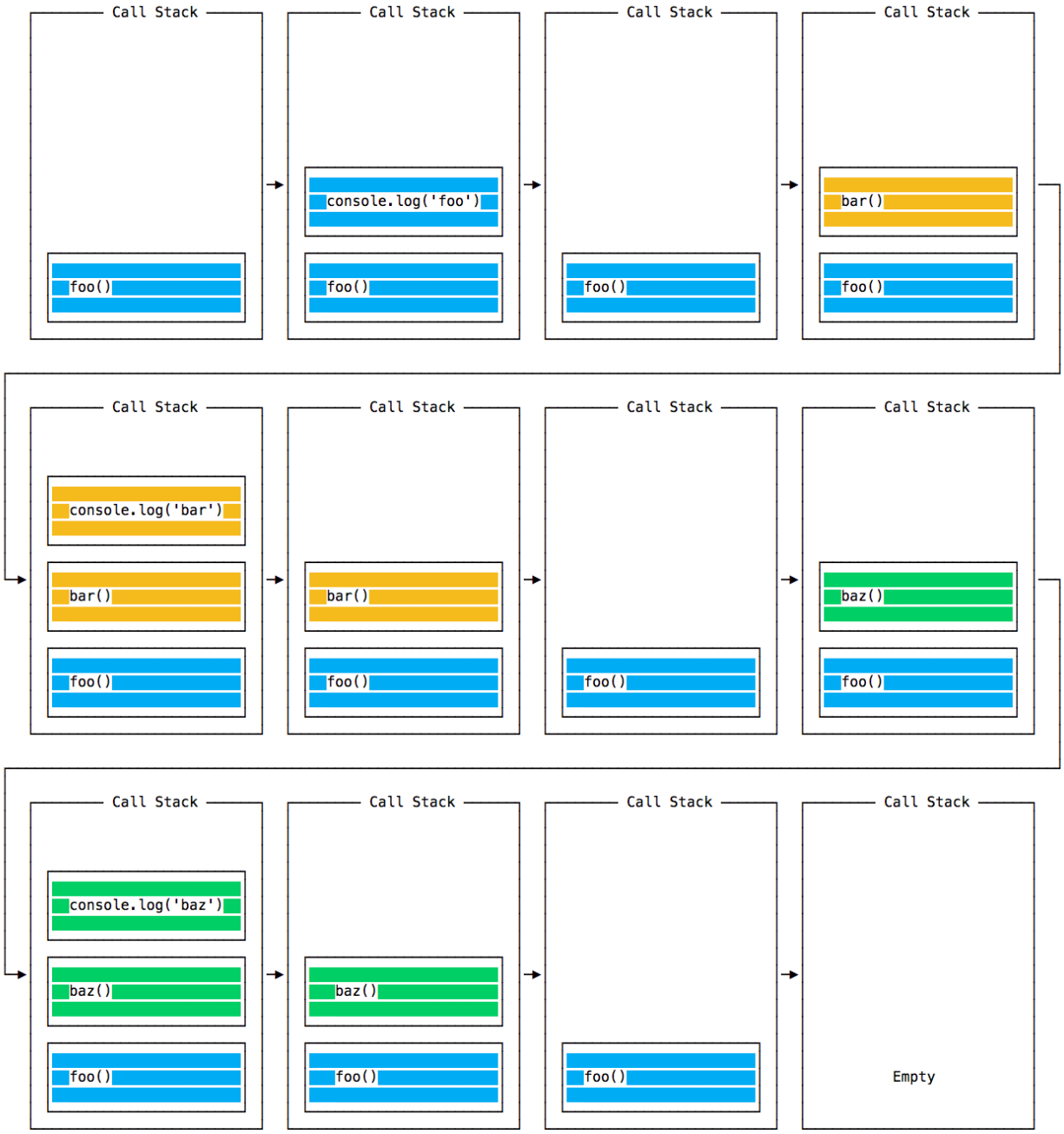
This code prints

```
foo
bar
baz
```

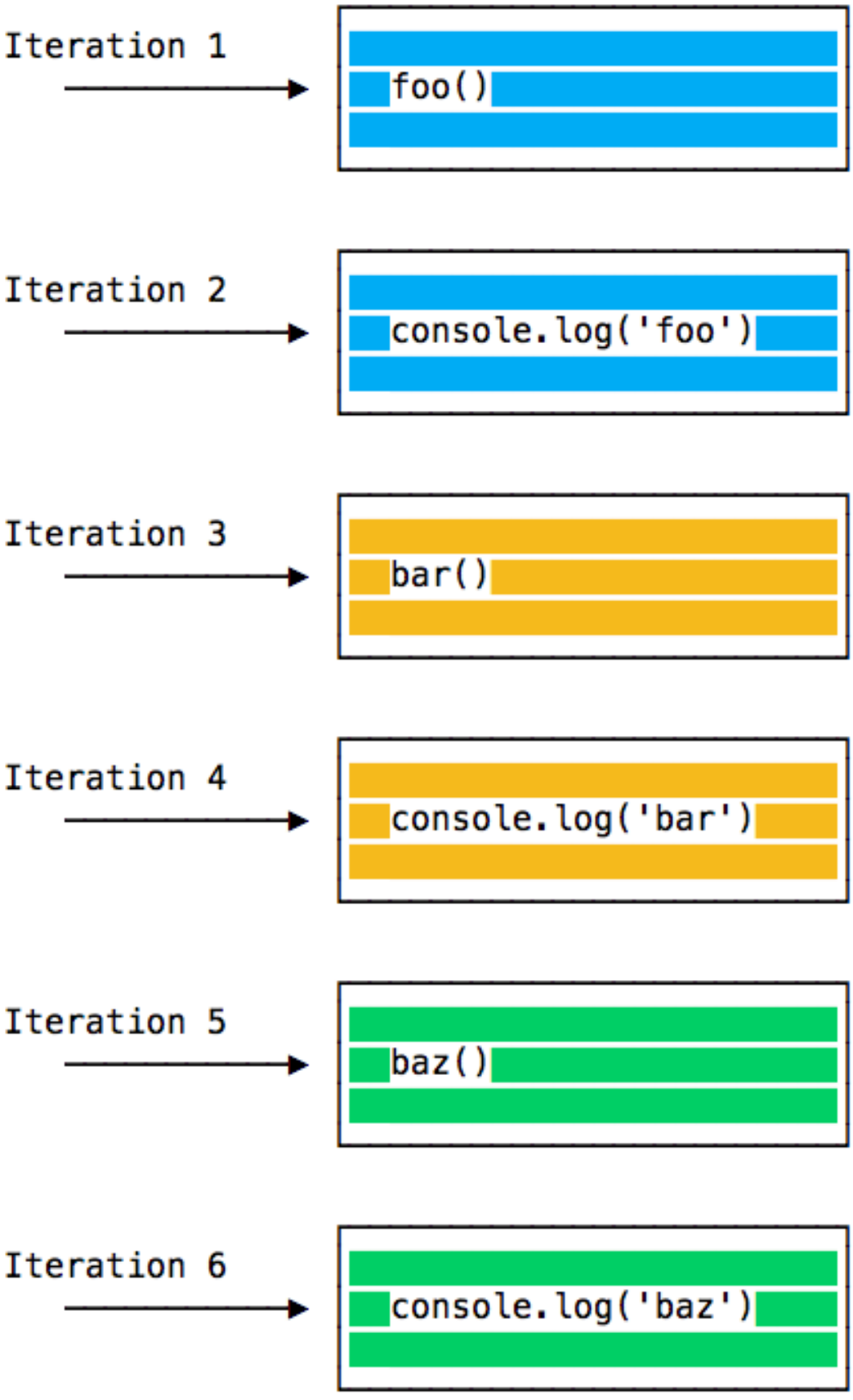
as expected.

When this code runs, first `foo()` is called. Inside `foo()` we first call `bar()`, then we call `baz()`.

At this point the call stack looks like this:



The event loop on every iteration looks if there's something in the call stack, and executes it:


Iteration 1
foo()
Iteration 2
console.log('foo')
Iteration 3
bar()
Iteration 4
console.log('bar')
Iteration 5
baz()
Iteration 6
console.log('baz')

until the call stack is empty.

Queuing function execution

The above example looks normal, there's nothing special about it: JavaScript finds things to execute, runs them in order.

Let's see how to defer a function until the stack is clear.

The use case of `setTimeout(() => {}), 0)` is to call a function, but execute it once every other function in the code has executed.

Take this example:

```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')
  setTimeout(bar, 0)
  baz()
}

foo()
```

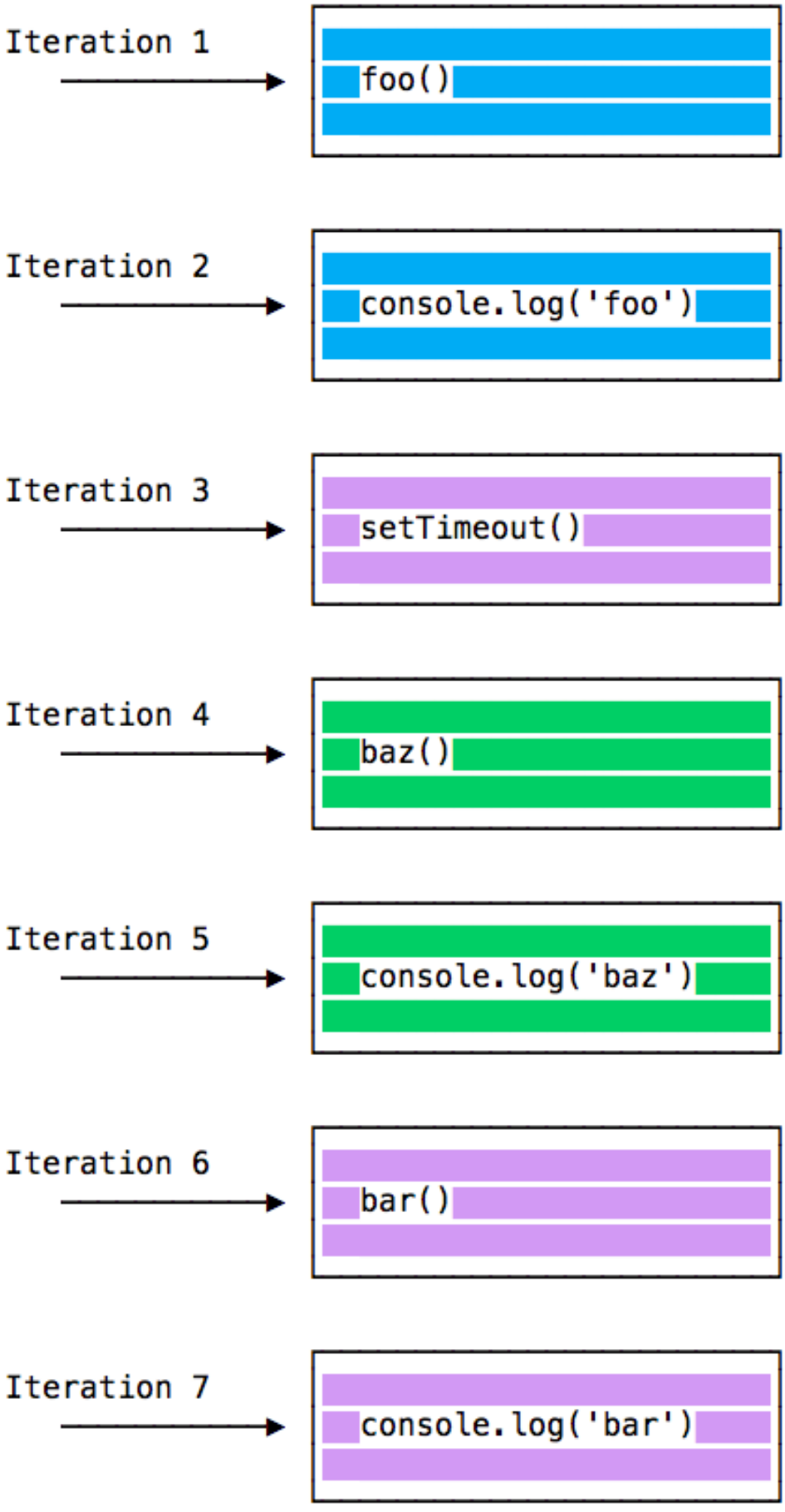
This code prints, maybe surprisingly:

```
foo
baz
bar
```

When this code runs, first foo() is called. Inside foo() we first call setTimeout, passing `bar` as an argument, and we instruct it to run immediately as fast as it can, passing 0 as the timer. Then we call baz().

At this point the call stack looks like this:

Here is the execution order for all the functions in our program:


Iteration 1
foo()
Iteration 2
console.log('foo')
Iteration 3
setTimeout()
Iteration 4
baz()
Iteration 5
console.log('baz')
Iteration 6
bar()
Iteration 7
console.log('bar')

Why is this happening?

The Message Queue

When setTimeout() is called, the Browser or Node.js start the timer. Once the timer expires, in this case immediately as we put 0 as the timeout, the callback function is put in the **Message Queue**.

The Message Queue is also where user-initiated events like click or keyboard events, or fetch responses are queued before your code has the opportunity to react to them. Or also DOM events like `onLoad`.

The loop gives priority to the call stack, and it first processes everything it finds in the call stack, and once there's nothing in there, it goes to pick up things in the message queue.

We don't have to wait for functions like `setTimeout`, fetch or other things to do their own work, because they are provided by the browser, and they live on their own threads. For example, if you set the `setTimeout` timeout to 2 seconds, you don't have to wait 2 seconds - the wait happens elsewhere.

ES6 Job Queue

ECMAScript 2015 introduced the concept of the Job Queue, which is used by Promises (also introduced in ES6/ES2015). It's a way to execute the result of an async function as soon as possible, rather than being put at the end of the call stack.

Promises that resolve before the current function ends will be executed right after the current function.

I find nice the analogy of a rollercoaster ride at an amusement park: the message queue puts you at the back of the queue, behind all the other people, where you will have to wait for your turn, while the job queue is the fastpass ticket that lets you take another ride right after you finished the previous one.

Example:

```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')
  setTimeout(bar, 0)
  new Promise((resolve, reject) =>
    resolve('should be right after baz, before bar')
  ).then(resolve => console.log(resolve))
  baz()
}

foo()
```

This prints

```
foo
```

```
baz
should be right after baz, before bar
bar
```

That's a big difference between Promises (and Async/await, which is built on promises) and plain old asynchronous functions through `setTimeout()` or other platform APIs.

I built a free event loop visualizer where you can step through code like this and watch the call stack, microtask queue and macrotask queue in action.

Tasks and microtasks

Understand how event callbacks, timers, promise reactions, and rendering opportunities are scheduled around the JavaScript stack.

The event loop decides when queued JavaScript may use the main thread. A click handler, a timer callback, and the initial execution of a script are examples of tasks. Promise reactions and `queueMicrotask()` callbacks use the microtask queue.

Predict this output before running it:

```
console.log('start')

setTimeout(() => console.log('task'), 0)

Promise.resolve().then(() => console.log('microtask'))

console.log('end')
```

The result is:

```
start
end
microtask
task
```

The current task runs to completion, so both synchronous logs come first. When its call stack is empty, the browser drains queued microtasks. The timer callback belongs to a later task even though its delay is zero.

Between tasks, after microtasks have run, the browser may get a rendering opportunity. That word "may" matters: a timer is not a promise that a frame will be painted immediately.

Microtasks are useful for finishing a small piece of related work before other tasks. They are dangerous when they keep adding more microtasks:

```
function repeat() {
  queueMicrotask(repeat)
}

repeat()
```

The queue never drains, so input, timers, and rendering can be starved. Keep microtasks bounded. When work is large, split it across later tasks or move suitable computation to a worker.

Long tasks block the page

Understand why a long JavaScript function delays input handlers, timers, style work, layout, and painting.

The page's main thread runs JavaScript and coordinates much of style calculation, layout, and painting. It can do only one piece of that work at a time.

```
button.addEventListener('click', () => {
  const end = performance.now() + 500
  while (performance.now() < end) {
    // Simulate expensive synchronous work.
  }

  status.textContent = 'Done'
})
```

The DOM assignment is near the end of the handler, but the visitor sees no update until the task finishes and the browser gets another rendering opportunity. A second click also waits. This is why a page can look loaded and still feel frozen.

Performance tools commonly flag tasks longer than 50 milliseconds as long tasks. That is a diagnostic boundary, not a performance budget: several shorter tasks can still create a poor interaction.

Record the click in the Performance panel. Look for a wide task on the Main track and expand it until you find the function consuming the time. Then choose a fix that matches the work:

- remove calculations whose result is not needed
- use a better algorithm or process less data
- split deferrable work and yield so input and rendering can run
- move CPU-heavy work without DOM access to a Web Worker

Splitting work has overhead, and a worker adds messaging and data-transfer cost. Measure the complete interaction after the change. The goal is not merely a shorter function; it is a faster next paint and a responsive page.

Garbage collection

Understand reachability, mark-and-sweep collection, generations, and why collection can reclaim only unreachable values.

JavaScript creates objects without asking you to free each one. The engine reclaims memory by finding values that are no longer reachable.

Reachability starts from roots the running program can still access, such as global variables and active call-stack frames. References are then followed from those roots:

```
let user = {
  name: 'Ada',
  preferences: { theme: 'dark' }
}

user = null
```

If nothing else references that object graph, it becomes eligible for collection. “Eligible” does not

mean “freed immediately.” The engine decides when to collect, and the strategy varies by browser and workload.

Modern collectors divide the work in several ways. For example, they can treat new and long-lived objects differently or perform parts of collection incrementally. These are implementation techniques for reducing disruption, not timing guarantees an application should depend on.

Three consequences matter in application code:

1. Temporary allocation still costs work even when it is collected soon.
2. A reachable value cannot be collected, even when the feature no longer needs it.
3. Memory usage does not have to fall immediately after references are released.

Open DevTools Memory tools, take a heap snapshot, create and discard a large group of objects, and take another snapshot after allowing collection. Focus on which objects remain reachable and their retaining paths. Do not build application logic around forcing garbage collection; production code cannot use it as a reliable cleanup mechanism.

How browser memory leaks happen

Recognize listeners, timers, closures, collections, and detached DOM nodes that keep unwanted objects reachable.

A JavaScript memory leak is usually not memory the collector missed. It is memory the program accidentally keeps reachable.

```
const removedPanels = []

function closePanel(panel) {
  panel.remove()
  removedPanels.push(panel)
}
```

The panel is detached from the document, but the array still references it. The panel and everything reachable through it remain in memory.

Common retaining references include:

- event listeners registered on long-lived objects
- intervals, observers, and subscriptions that were not stopped
- closures captured by callbacks
- caches without size or lifetime limits
- collections of detached DOM nodes

Give every long-lived resource an owner and a cleanup point:

```
function mountPanel(panel) {
  const controller = new AbortController()

  window.addEventListener('resize', updatePanel, {
    signal: controller.signal
  })

  return () => controller.abort()
}
```

To investigate, repeat a lifecycle rather than staring at one memory number: open a view, close it, and do that several times. Take heap snapshots before and after. If instances accumulate, inspect their retaining paths. The useful evidence is the reference keeping an object alive, because that points to the cleanup code that is missing.

Some growth is intentional caching, and garbage collection timing is nondeterministic. A leak diagnosis needs repeatable retention after the feature should have released its data.

Web Workers

Learn how to use Web Workers to run JavaScript in a background thread, communicating with the main page through postMessage so heavy work won't block the UI.

Introduction

A Web Worker runs JavaScript in a background thread, separate from the page's main thread.

Use a worker for CPU-heavy work that would otherwise make the page slow or unresponsive.

The main thread still handles the DOM and user interface. The worker cannot access `window` or `document`, so both sides communicate through messages.

This guide focuses on **dedicated workers**, which belong to one page or script. The MDN Web Workers guide also covers shared workers.

They have quite a few limitations:

- no direct access to the DOM
- the initial worker script normally needs to be same-origin
- the page and worker don't share ordinary JavaScript objects
- behavior for pages loaded with `file://` is not consistent, so use a local web server

The global scope inside a worker is a `WorkerGlobalScope`, not `Window`.

Browser support for Web Workers

Dedicated Web Workers are widely available in current browsers.

You can check for Web Workers support using

```
if ('Worker' in window) {
  // Web Workers are available
}
```

Create a Web Worker

Create a classic worker by passing its script URL to the `Worker` constructor:

```
const worker = new Worker('worker.js')
```

You can also create a module worker:

```
const worker = new Worker('worker.js', {
  type: 'module'
})
```

Module workers support `import` statements. The MDN Worker constructor reference explains URL and same-origin rules.

Communication with a Web Worker

There are two main ways to communicate to a Web Worker:

- the postMessage API offered by the Web Worker object
- the Channel Messaging API

Using postMessage in the Web Worker object You can send messages using `postMessage` on the `Worker` object.

By default, message data is copied using the structured clone algorithm. It is not shared.

Some objects, such as `ArrayBuffer`, can be transferred instead. A transferred object is no longer usable by the sender.

main.js

```
const worker = new Worker('worker.js')
worker.postMessage('hello')
```

worker.js

```
self.onmessage = event => {
  console.log(event.data)
}

self.onerror = event => {
  console.error(event.message)
}
```

Send back messages A worker can send messages back to its creator using its global `postMessage()` function:

worker.js

```
self.onmessage = event => {
  console.log(event.data)
  self.postMessage('hey')
}

self.onerror = event => {
  console.error(event.message)
}
```

main.js

```
const worker = new Worker('worker.js')
worker.postMessage('hello')

worker.onmessage = event => {
  console.log(event.data)
}
```

Multiple event listeners If you want to setup multiple listeners for the `message` event, instead of using `onmessage` create an event listener (applies to the `error` event as well):

worker.js

```
self.addEventListener('message', event => {
  console.log(event.data)
  self.postMessage('hey')
})

self.addEventListener('message', () => {
  console.log(`I'm curious and I'm listening too`)
})

self.addEventListener('error', event => {
  console.log(event.message)
})
```

main.js

```
const worker = new Worker('worker.js')
worker.postMessage('hello')

worker.addEventListener('message', event => {
  console.log(event.data)
})
```

Using the Channel Messaging API Instead of using the built-in postMessage API offered by Web Workers, we can choose to use the more general-purpose Channel Messaging API to communicate to them.

main.js

```
const worker = new Worker('worker.js')
const channel = new MessageChannel()

channel.port1.addEventListener('message', event => {
  console.log(event.data)
})
channel.port1.start()

worker.postMessage(
  { port: channel.port2 },
  [channel.port2]
)
```

worker.js

```
self.addEventListener('message', event => {
  const port = event.data.port
  port.postMessage('hello from the worker')
})
```

The `MessagePort` is listed as a transferable object. Ownership moves to the worker when it is posted.

Web Worker Lifecycle

A dedicated worker can receive messages after its initial script finishes. Don't rely on a worker stopping merely because the current function returned.

Stop it explicitly with `terminate()` from the main thread:

main.js

```
const worker = new Worker('worker.js')
worker.terminate()
```

`terminate()` stops the worker immediately. It does not give the worker time to finish.

Inside the worker, call `close()` to stop accepting new tasks:

```
self.onmessage = event => {
  console.log(event.data)
  self.close()
}
```

The HTML specification defines the details of worker lifetime and termination.

Loading libraries in a Web Worker

Classic workers can load scripts synchronously with `importScripts()`:

```
importScripts('../utils/file.js', './something.js')
```

Module workers cannot use `importScripts()`. Use normal static or dynamic imports instead:

```
import { calculate } from './calculate.js'
```

See the MDN `importScripts()` reference for the security and cross-origin rules.

APIs available in Web Workers

The DOM is not available in a Web Worker, so you cannot interact with `window` or `document`.

Many other APIs are available, including:

- the Fetch API
- IndexedDB
- Promises
- the Channel Messaging API
- the Cache API
- the Console API
- timers such as `setTimeout()` and `setInterval()`
- WebSockets
- `crypto`
- `OffscreenCanvas`

Support still varies by API and browser. Check the official list of functions and classes available to workers before relying on one.

Check your understanding: the JavaScript engine

Check parsing, bytecode, optimization, the call stack, tasks, long work, garbage collection, generations, and retained DOM nodes.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Storage and browser security

Origins, cookies, local data, isolation, and sandboxing.

What is an origin

Identify an origin from its scheme, host, and port and understand why paths do not create separate origins.

An origin is the combination of a URL's scheme, host, and port. All three parts must match.

The browser needs a unit of isolation — a rule deciding which pages belong to the same ”site” and may touch each other's data, and which must be kept apart. The origin is that unit. Browsers use this boundary for DOM access, network response access, and storage.

For this URL:

`https://shop.example.com:8443/account?tab=orders`

the origin is:

`https://shop.example.com:8443`

The path, query string, and fragment do not participate. These pages therefore share an origin:

```
https://example.com/account
https://example.com/shop
```

These do not:

```
http://example.com
https://example.com
https://app.example.com
https://example.com:8443
```

Each line fails a different part of the test. The first differs from the second in scheme. The third changes the host — a subdomain counts as a completely different host. The fourth changes the port.

One wrinkle about ports: when a URL omits the port, the scheme's default applies. `https://example.com` and `https://example.com:443` are the same origin, because 443 is the default for `https`. Writing the default explicitly changes nothing; writing any other number creates a new origin.

Check an origin yourself

Open the Console on any page and run:

```
location.origin
```

You can also compute the origin of any URL without visiting it:

```
new URL('https://shop.example.com:8443/account?tab=orders').origin
// 'https://shop.example.com:8443'
```

The mistake to avoid

The common error is grouping URLs by domain name instead of origin. A subdomain is not automatically trusted merely because it belongs to the same parent domain: `api.example.com` and `example.com` are different origins, full stop. Developers assume "it's all our domain, it will work", and then the first cross-origin request or `iframe` access fails.

When that happens, compare the two origins with `location.origin` or `new URL(...).origin` and check the three parts one by one. Asking "do scheme, host, and port match?" is the first step in explaining many CORS and cross-window errors — the following lessons build directly on this definition.

Learn how HTTP Cookies work

Learn how HTTP cookies move between browser and server, how scope and expiration work, and how Secure, HttpOnly, and SameSite protect them.

An HTTP cookie is a small name-value pair that a browser stores for a site.

Servers often use cookies to identify a session. The cookie contains an opaque session ID, while the server keeps the account and session data.

Cookies are also used for preferences and other small values. They are not a replacement for a database.

How cookies travel

A server sets a cookie in an HTTP response:

```
Set-Cookie: session=abc123; Path=/; Secure; HttpOnly; SameSite=Lax
```

The browser stores it. On later matching requests, the browser sends:

```
Cookie: session=abc123
```

The browser decides whether a cookie matches by checking its domain, path, expiration, `Secure`, `SameSite`, and partition rules.

Frontend JavaScript does not need to manually add the `Cookie` header. Browsers manage that header, and do not let scripts set it directly.

Cookie limits

Cookies are intentionally small. Browser limits vary, but an individual cookie is commonly limited to about 4 KB.

Browsers also limit how many cookies an origin or domain can keep. Do not depend on one exact number.

Cookies add bytes to matching HTTP requests. Use Web Storage or IndexedDB for client-only data that does not need to reach the server.

Session and persistent cookies

A cookie without `Expires` or `Max-Age` is a **session cookie**:

```
Set-Cookie: theme=dark; Path=/
```

Session cookies normally last for the browser session. Browser session restore can preserve them across a restart, so closing the browser is not a reliable security boundary.

Use `Max-Age` to set a lifetime in seconds:

```
Set-Cookie: theme=dark; Max-Age=2592000; Path=/
```

This cookie can last for 30 days.

Alternatively, use an HTTP date with `Expires`:

```
Set-Cookie: theme=dark; Expires=Tue, 18 Aug 2026 12:00:00 GMT; Path=/
```

When both are present, `Max-Age` takes precedence.

Limit a cookie by host and path

If you omit `Domain`, the browser creates a **host-only cookie**. A cookie set by `app.example.com` then returns only to `app.example.com`, not to its subdomains or parent domain.

Set `Domain=example.com` only when the cookie must also reach subdomains:

```
Set-Cookie: preference=compact; Domain=example.com; Path=/
```

A site cannot set cookies for an unrelated domain or a public suffix such as `com`.

Leaving out `Domain` is the safer default because it gives the cookie a smaller scope.

`Path` limits which request paths receive the cookie:

```
Set-Cookie: dashboard-view=list; Path=/dashboard
```

This matches `/dashboard` and paths below it.

`Path` is a delivery rule, not a security boundary. Other code on the same origin may still be able to access the cookie.

Protect cookies with attributes

Secure `Secure` tells the browser to send the cookie only over HTTPS:

```
Set-Cookie: session=abc123; Secure
```

Use HTTPS for the entire site. `Secure` protects cookie transport, but does not encrypt the value inside the browser or server.

HttpOnly `HttpOnly` hides a cookie from JavaScript APIs such as `document.cookie`:

```
Set-Cookie: session=abc123; Secure; HttpOnly
```

Only a server can set `HttpOnly` through the `Set-Cookie` response header. This does not work: adding `HttpOnly` to a `document.cookie` assignment cannot create an `HttpOnly` cookie.

An XSS payload cannot read an `HttpOnly` session cookie. It may still perform actions as the signed-in user, so `HttpOnly` reduces impact but does not fix XSS.

SameSite `SameSite` controls whether a cookie is sent with cross-site requests.

`Strict` gives the narrowest cross-site behavior:

```
Set-Cookie: session=abc123; SameSite=Strict
```

`Lax` also permits some top-level cross-site navigations:

```
Set-Cookie: session=abc123; SameSite=Lax
```

`None` permits cross-site use and requires `Secure`:

```
Set-Cookie: widget-session=abc123; SameSite=None; Secure
```

Modern browsers commonly treat a missing `SameSite` as `Lax`, with compatibility details. Set it explicitly so the intended behavior is clear.

`SameSite` helps defend against CSRF, but it is not the only CSRF control an application may need.

Partitioned `Partitioned` requests partitioned storage for a third-party cookie. The browser keeps a separate cookie jar for each top-level site.

It requires `Secure`:

```
Set-Cookie: widget=compact; SameSite=None; Secure; Partitioned
```

Use this only when building a cross-site embedded feature that needs state.

Use cookie name prefixes

Cookie prefixes let supporting browsers enforce attribute rules.

`__Host-` is useful for host-bound cookies:

```
Set-Cookie: __Host-session=abc123; Path=/; Secure; HttpOnly; SameSite=Lax
```

A `__Host-` cookie must use `Secure`, must use `Path=/`, and must not have a `Domain` attribute.

Prefixes add useful checks. They do not replace secure session design.

Read and write cookies in JavaScript

`document.cookie` exposes non-`HttpOnly` cookies available to the current document:

```
console.log(document.cookie)
```

The result is one string:

```
theme=dark; language=en
```

Writing to `document.cookie` adds or updates one cookie. It does not replace every cookie:

```
document.cookie = 'theme=dark; Path=/; Max-Age=2592000; SameSite=Lax; Secure'
```

Encode values that can contain unsupported characters:

```
const value = encodeURIComponent('dark mode')
document.cookie = `theme=${value}; Path=/; SameSite=Lax; Secure`
```

`document.cookie` is synchronous and can be awkward to parse. The asynchronous Cookie Store API is an option where your supported browsers provide it.

Authentication cookies should normally be server-set and `HttpOnly`, which means frontend JavaScript should not read them.

Update or delete a cookie

Update a cookie by setting the same name, domain, and path:

```
document.cookie = 'theme=light; Path=/; Max-Age=2592000; SameSite=Lax; Secure'
```

Delete it by using the same scope and a non-positive `Max-Age`:

```
document.cookie = 'theme=; Path=/; Max-Age=0; SameSite=Lax; Secure'
```

If the path or domain does not match the original cookie, you create or delete a different cookie.

Cookie security rules

For session cookies:

- generate an unpredictable session ID
- keep account data on the server
- use HTTPS and `Secure`
- use `HttpOnly`
- choose an explicit `SameSite` value
- use the smallest practical host and path scope
- rotate the session ID after login and privilege changes
- expire the session on the server, not only in the browser

Do not trust a cookie merely because the browser returned it. Users can modify non-`HttpOnly` cookies, and can send custom HTTP requests.

Signing a value can detect modification. Encryption can hide its contents. Neither one makes authorization checks optional.

See MDN's guide to using HTTP cookies, the `Set-Cookie` reference, and the `document.cookie` reference.

The Web Storage API: local storage and session storage

Learn how the Web Storage API lets you store data in the browser with local storage and session storage, using methods like setItem, getItem, and removeItem.

Introduction

The **Web Storage API** stores string key/value pairs in the browser.

It provides two storage mechanisms:

- `localStorage` persists across browser sessions
- `sessionStorage` lasts for one tab's page session

They are part of the storage options available on the Web Platform, which also include:

- Cookies
- IndexedDB
- The Cache API

Both storage areas are separated by **origin**. The scheme, host, and port must all match. Other origins cannot read the data through Web Storage.

`sessionStorage` maintains data for the duration of the page session. Different tabs have separate page sessions, even when they show the same site.

When the tab closes, its `sessionStorage` data is cleared.

This is useful for separate workflows in different tabs. The MDN Web Storage guide describes how storage is partitioned.

`localStorage` is shared by same-origin pages and persists after the browser closes. The app, the user, or the browser can still clear it. Private browsing data is normally removed when the private session ends.

Web Storage is synchronous. Large reads and writes block JavaScript on the page, so use IndexedDB for larger or structured data.

Web Storage is only accessible in the browser. It's not sent to the server like cookies do.

Do not store passwords, session IDs, or authentication tokens in Web Storage. Any JavaScript running on the origin can read them, including code injected through an XSS vulnerability. The MDN session-management guide recommends keeping session identifiers out of JavaScript-accessible storage.

How to access the storage

Both Local and Session Storage are available on the `window` object, so you can access them using `sessionStorage` and `localStorage`.

Their set of properties and methods is exactly the same, because they return the same object, a Storage object.

The Storage object has a single property, `length`, which is the number of stored items.

Methods

setItem(key, value) `setItem()` adds an item to the storage. Accepts a string as key, and a string as a value:

```
localStorage.setItem('username', 'flaviocopes')
localStorage.setItem('id', '123')
```

If you pass any value that's not a string, it will be converted to string:

```
localStorage.setItem('test', 123) //stored as the '123' string
localStorage.setItem('test', { test: 1 }) //stored as "[object Object]"
```

Use JSON when you want to store an object:

```
const settings = { theme: 'dark' }
localStorage.setItem('settings', JSON.stringify(settings))
```

getItem(key) `getItem()` is the way to retrieve a string value from the storage, by using the key string that was used to store it:

```
localStorage.getItem('username') // 'flaviocopes'
localStorage.getItem('id') // '123'
```

`getItem()` returns `null` when the key does not exist.

Parse a stored JSON value like this:

```
const settings = JSON.parse(localStorage.getItem('settings'))
```

removeItem(key) `removeItem()` removes the item identified by `key` from the storage, returning nothing (an `undefined` value):

```
localStorage.removeItem('id')
```

key(n) `key(n)` returns the name of the key at index `n`.

The order is defined by the browser, so do not rely on a particular order. See the `Storage.key()` reference.

If you reference a number that does not point to a storage item, it returns `null`.

clear() `clear()` removes everything from the storage object you are manipulating:

```
localStorage.setItem('a', 'a')
localStorage.setItem('b', 'b')
localStorage.length //2
localStorage.clear()
localStorage.length //0
```

Storage size limits

Web Storage is limited to about 10 MiB per origin: 5 MiB for `localStorage` and 5 MiB for `sessionStorage`. Browsers throw `QuotaExceededError` when an origin reaches the limit. See MDN's current storage quota guide.

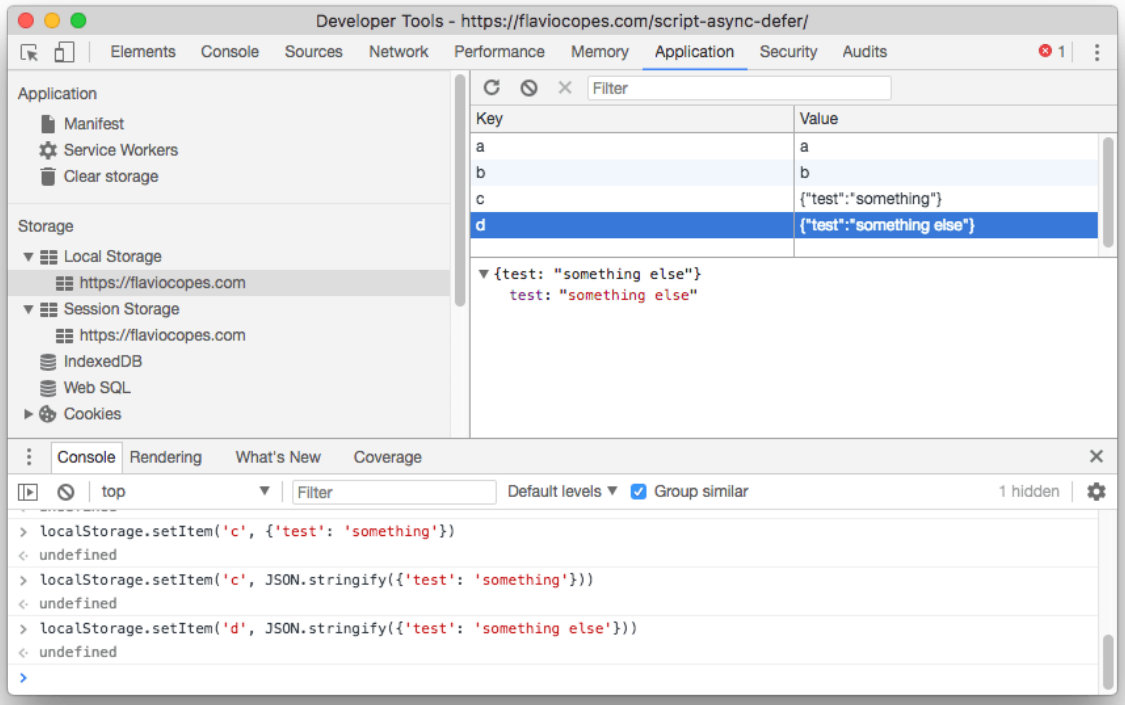
Do not treat stored data as permanent. Users can clear it, private sessions delete it, and browser storage policies can affect it.

Going over quota Handle quota errors, especially if you store a lot of data:

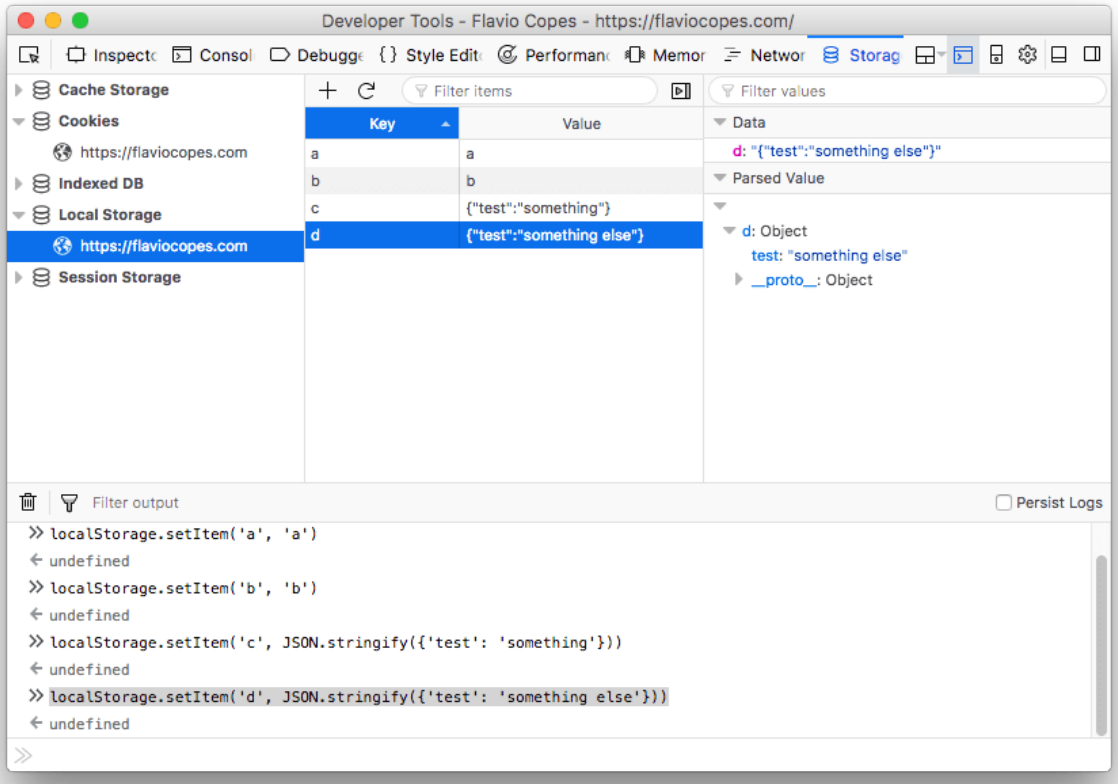
```
try {
  localStorage.setItem('key', 'value')
} catch (error) {
  if (error.name === 'QuotaExceededError') {
    console.error('Storage quota exceeded')
  } else {
    throw error
  }
}
```

Developer Tools

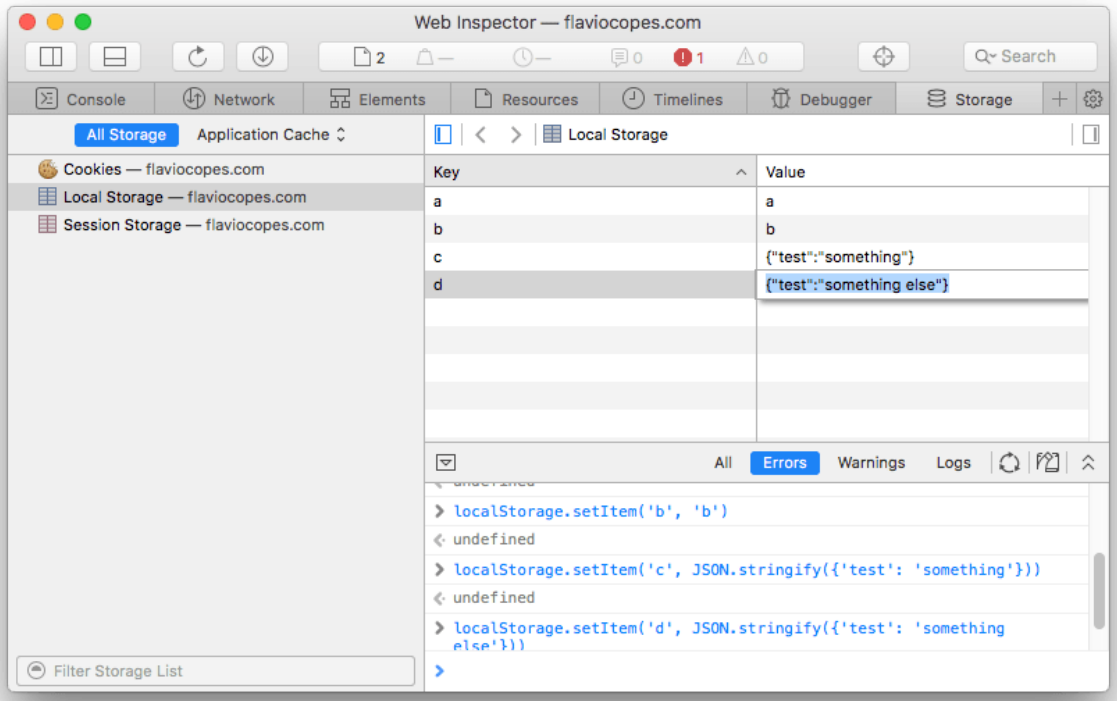
The DevTools of the major browsers all offer a nice interface to inspect and manipulate the data stored in the Local and Session Storage.



Chrome



Firefox



Safari

Dive into IndexedDB

Learn IndexedDB with the idb promise wrapper: create a database, store and query structured data, use transactions, and upgrade schemas.

IndexedDB is the browser's asynchronous database for structured data. It can store JavaScript objects, arrays, dates, files, blobs, and binary data.

Use IndexedDB when browser storage must hold more or richer data than a few string values. Good examples include offline application data, cached documents, drafts, and local indexes.

For a small preference such as a color theme, `localStorage` is simpler.

The native IndexedDB API is event-based. In this guide I use the small `idb` library, which adds promises and convenient shortcuts while preserving IndexedDB's concepts.

Install idb

Install the package:

```
npm install idb
```

Import the APIs you need:

```
import { deleteDB, openDB } from 'idb'
```

The library works in browsers. IndexedDB is not available during server-side rendering, so open the database only in browser code.

Create a database and object store

Open a database with a name and version:

```
import { openDB } from 'idb'

const db = await openDB('notes-app', 1, {
  upgrade(db) {
    db.createObjectStore('notes', {
      keyPath: 'id',
    })
  },
})
```

An **object store** is similar to a table. This example uses each note's `id` property as its key.

The `upgrade()` callback runs when the database is first created and whenever you open it with a higher version number. Schema changes must happen there.

The callback receives a native upgrade transaction. Keep it focused on database changes; do not wait for network requests inside it.

Add or replace data

Use `add()` when a key must not already exist:

```
await db.add('notes', {
  id: 'note-1',
  title: 'Shopping list',
  body: 'Milk and bread',
  updatedAt: new Date(),
})
```

If `note-1` already exists, `add()` rejects with a constraint error.

Use `put()` when you want to insert a new value or replace the value at an existing key:

```
await db.put('notes', {
  id: 'note-1',
  title: 'Shopping list',
  body: 'Milk, bread, and coffee',
  updatedAt: new Date(),
})
```

IndexedDB stores values using the structured clone algorithm. Functions and DOM elements cannot be stored.

Read data

Get one value by key:

```
const note = await db.get('notes', 'note-1')

if (note) {
  console.log(note.title)
}
```

`get()` returns `undefined` when the key does not exist.

Get every value in the store:

```
const notes = await db.getAll('notes')
```

For a large store, do not load everything blindly. Add an index and query only the records the interface needs.

Add an index

An index lets you query by a property other than the primary key.

Indexes are part of the schema, so create one during an upgrade:

```
const db = await openDB('notes-app', 2, {
  upgrade(db, oldVersion, newVersion, transaction) {
    if (oldVersion < 1) {
      db.createObjectStore('notes', {
        keyPath: 'id',
      })
    }

    if (oldVersion < 2) {
      const store = transaction.objectStore('notes')
      store.createIndex('by-updated-at', 'updatedAt')
    }
  },
})
```

Then query through the index:

```
const notes = await db.getAllFromIndex('notes', 'by-updated-at')
```

Version checks allow a user to upgrade directly from any older schema instead of assuming everyone has the previous version.

Use a transaction for related operations

The shortcut methods such as `db.put()` create a transaction for one operation.

When several changes must succeed or fail together, create one read-write transaction:

```
const tx = db.transaction('notes', 'readwrite')
const store = tx.objectStore('notes')

await store.put({
  id: 'note-2',
  title: 'Ideas',
  body: 'Build an offline notes app',
  updatedAt: new Date(),
})

await store.delete('note-1')
await tx.done
```

`tx.done` resolves when the entire transaction completes and rejects if it aborts.

Await requests as you issue them, then await `tx.done`. Do not perform unrelated async work, such

as a fetch request, in the middle of a transaction. IndexedDB transactions can become inactive when control returns to the event loop without a pending database request.

Transactions are atomic: if one request fails and the transaction aborts, none of its changes are committed.

Delete records

Delete one record by key:

```
await db.delete('notes', 'note-2')
```

Clear every record while keeping the object store:

```
await db.clear('notes')
```

Delete the entire database:

```
db.close()
await deleteDB('notes-app')
```

Other open tabs can block a version upgrade or database deletion. Applications that stay open for a long time should close an old connection when a newer version is requested:

```
const db = await openDB('notes-app', 2, {
  upgrade(db, oldVersion, newVersion, transaction) {
    // Apply versioned schema changes here.
  },
  blocking() {
    db.close()
  },
})
```

The page can then tell the user to refresh.

Check whether a store exists

`objectStoreNames` is a list-like property, not a function:

```
if (db.objectStoreNames.contains('notes')) {
  console.log('The notes store exists')
}
```

Create and delete object stores only inside the version-change transaction passed to `upgrade()`.

Storage limits and persistence

IndexedDB capacity and eviction behavior vary by browser, device, available disk space, and browsing mode. Private browsing storage is generally temporary.

Do not assume data will remain forever. Important user data needs a backup or server synchronization strategy.

The Storage API can report estimated usage and quota:

```
const estimate = await navigator.storage.estimate()

console.log(estimate.usage)
```

```
console.log(estimate.quota)
```

An application can request persistent storage with `navigator.storage.persist()`, but the browser decides whether to grant it.

For more detail, see MDN's IndexedDB API overview, Using IndexedDB, and storage quota and eviction criteria.

Choose the right browser storage

Choose cookies, localStorage, sessionStorage, or IndexedDB according to transport, lifetime, size, and access patterns.

Choose browser storage from the job it must do, not from the shortest API.

Storage	Useful when	Main cost or constrai
Cookies	The server must receive a small value with matching HTTP requests	Sent repeatedly; stric
`localStorage`	A small string preference should survive browser restarts	Synchronous access c
`sessionStorage`	A small string value belongs to one tab session	Disappears with that
IndexedDB	The page needs asynchronous access to larger structured data	More code and a tran

For example, a color theme can fit in `localStorage`. An offline product catalog belongs in IndexedDB. A server-managed session commonly uses a cookie with `Secure`, `HttpOnly`, and an appropriate `SameSite` policy rather than a token exposed to every script on the page.

Storage is scoped and constrained by the browser. Quotas, eviction behavior, private browsing, and user settings mean persistent data is not an infallible database. Keep important server data on the server, handle failed writes, and make cached data replaceable.

`localStorage` accepts only strings:

```
localStorage.setItem('preferences', JSON.stringify({ theme: 'dark' }))

const preferences = JSON.parse(
  localStorage.getItem('preferences') ?? '{}'
)
```

Do not store secrets just because an API persists them. Any script running in the origin can generally access JavaScript-readable storage, including malicious script introduced by an XSS bug.

Use the Application panel to inspect the storage used by a real page. Identify the owner, lifetime, maximum plausible size, and cleanup rule for each entry. If none is clear, the storage decision is incomplete.

The same-origin policy

Understand the browser boundary that prevents one origin from freely reading another origin's protected data.

The same-origin policy stops script from one origin from freely reading protected data belonging to another origin.

Imagine visiting `evil.example` while already signed in to a bank. Your browser may attach bank cookies to a request sent to the bank, but the malicious page must not be allowed to read the

account response. The read boundary is the essential protection.

The policy is more precise than "cross-origin requests are blocked." Browsers permit many cross-origin actions:

- an `<img>` can display an image from another origin
- a `<form>` can submit to another origin
- a page can link to or embed another origin
- `fetch()` can send some requests even when script cannot read the response

Controlled exceptions make legitimate applications possible. CORS response headers let a server grant selected origins access to a response. `window.postMessage()` lets cooperating windows exchange messages, but the sender should use a specific target origin and the receiver must validate `event.origin` and the message data.

Try fetching an API from the Console on a different origin and inspect both places:

1. The Network panel shows whether a request was sent and what response arrived.
2. The Console shows whether the page's script was allowed to use that response.

This separates a network failure from a browser access decision. Do not "fix" a CORS error by reflecting every origin or disabling checks. Configure the server to authorize only the origins, methods, and credentials the application actually needs.

CORS, Cross-Origin Resource Sharing

Learn how CORS response headers let browser JavaScript read cross-origin responses, how preflight and credentials work, and how to configure Express.

CORS, or **Cross-Origin Resource Sharing**, is an HTTP-header mechanism that lets a server choose which other origins may read its responses in a browser.

An origin is the combination of scheme, host, and port. These are different origins:

- `https://app.example.com`
- `https://api.example.com`
- `http://app.example.com`
- `https://app.example.com:8443`

Browser JavaScript normally follows the same-origin policy. A `fetch()` or XHR call can send a cross-origin request, but JavaScript cannot read the response unless the server returns the right CORS headers.

The server controls the policy. You cannot fix a missing CORS header only in frontend JavaScript.

If you're stuck on a CORS error right now, I built a free CORS debugger: describe your request and it tells you which headers the server needs to send.

The complete rules live in the MDN CORS guide.

What does CORS protect?

CORS controls whether browser JavaScript can **read a response**. It does not stop a request from reaching the server.

This distinction matters.

A form, `curl`, another server, or a script outside the browser can still send a request to your API.

CORS is not authentication or authorization. You must still protect private routes and check permissions on the server.

Some cross-origin resources can be embedded without CORS, including many images and classic scripts. Reading those resources from JavaScript is different. Web fonts, JavaScript modules, WebGL textures, and images read through Canvas also use CORS rules.

Allow a public resource

For a public response that does not use credentials, the server can return:

`Access-Control-Allow-Origin: *`

The wildcard means any origin may read the response in browser JavaScript.

Do not use this for private data. The browser does not know whether a response is sensitive.

Allow one origin

To allow one frontend, return its exact origin:

`Access-Control-Allow-Origin: https://app.example.com`

If the server chooses the value dynamically from an allowlist, also return:

`Vary: Origin`

This tells caches that the response can change based on the request's `Origin` header. See the `Access-Control-Allow-Origin` reference.

Never reflect any incoming `Origin` without checking it against an allowlist first.

Example with Express

For Express, use the official `cors` middleware.

This route allows one frontend origin:

```
const express = require('express')
const cors = require('cors')

const app = express()

const corsOptions = {
  origin: 'https://app.example.com',
}

app.get('/products', cors(corsOptions), (request, response) => {
  response.json([{ id: 1, name: 'Keyboard' }])
})

app.listen(3000)
```

The middleware sets the response headers. It does not block non-browser clients from calling the route.

If the same policy applies to the whole API, register it once:

```
app.use(cors(corsOptions))
```

Application-level middleware also handles preflight requests for the matching routes.

How preflight requests work

Some cross-origin requests can be sent without a preflight. They must use `GET`, `HEAD`, or `POST`, and only CORS-safelisted headers.

For `POST`, the allowed `Content-Type` values are:

- `application/x-www-form-urlencoded`
- `multipart/form-data`
- `text/plain`

Even a `GET` can trigger preflight if you add a non-safelisted header such as `Authorization`.

For other requests, the browser first sends an `OPTIONS` request. This asks whether the real method and headers are allowed:

```
OPTIONS /products HTTP/1.1
Origin: https://app.example.com
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: authorization
```

The server can approve it with:

```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://app.example.com
Access-Control-Allow-Methods: DELETE
Access-Control-Allow-Headers: Authorization
```

The browser sends the `DELETE` request only after the preflight succeeds.

CORS with cookies

Fetch does not include cross-origin credentials by default. To send cookies, use:

```
fetch('https://api.example.com/account', {
  credentials: 'include',
})
```

The server must return both:

```
Access-Control-Allow-Origin: https://app.example.com
Access-Control-Allow-Credentials: true
```

You cannot combine credentials with `Access-Control-Allow-Origin: *`. The browser will block access to that response.

CORS also does not override cookie rules such as `SameSite` and `Secure`.

CORS does not replace CSRF protection. If a route changes data using cookies, protect it against cross-site request forgery too.

Debugging CORS errors

JavaScript usually receives a generic network error when CORS fails. The browser console contains the useful details.

Check:

- the exact frontend origin
- `Access-Control-Allow-Origin` on the response
- the `OPTIONS` response for preflighted requests
- allowed methods and headers
- credential settings on both client and server

Avoid reaching for `mode: 'no-cors'` as a fix. It gives JavaScript an opaque response whose status, headers, and body cannot be read.

Site isolation and renderer sandboxing

Understand the defense-in-depth boundaries that limit what compromised renderer code can reach.

A browser renders untrusted code from the internet while also handling files, devices, credentials, and operating-system APIs. Modern browsers reduce that risk by separating responsibilities and privileges.

A renderer process handles web content with restricted operating-system access. Its sandbox limits what exploited renderer code can directly reach. Site isolation can also keep content from different sites in separate renderer processes, reducing the cross-site data present in a compromised process.

The exact process model is an implementation detail. One process is not guaranteed per tab, iframe, or origin, and browsers make different choices based on platform and resources.

These layers solve different parts of the problem:

- the same-origin policy restricts what web code may read
- site isolation separates sensitive content at a process boundary
- sandboxing restricts a renderer's access to the operating system

They are defense in depth. They do not make an XSS bug harmless: injected script still runs with the affected page's permissions and can read or modify data available to that page.

In Chrome, open the browser's Task Manager on a page with cross-site iframes and compare its processes. Treat what you see as evidence of that browser's current architecture, not a contract your application can depend on. Your application still needs output escaping, a considered Content Security Policy, safe dependencies, and least-privilege server APIs.

Private browsing storage

Understand that private sessions isolate and discard local browsing data without making network activity anonymous.

A private window normally uses a storage context separated from the regular browser profile. Cookies and other local site data created there are temporary and are discarded when the browser's private session ends.

Exact lifetime and isolation details vary by browser. Several private windows may share one private session until all of them close, so closing a single window is not always the cleanup boundary.

Private browsing protects mainly against leaving local history and site data in the regular profile. It does not make a visitor anonymous:

- the site still receives the connection and request data
- a network administrator or internet provider can still observe traffic metadata
- downloads and bookmarks may remain on the device
- signing in still identifies the account

Test the boundary yourself. In a regular window, set a distinctive `localStorage` value. Open the same origin in a private window and check whether it exists. Set another value privately, close every private window, reopen one, and inspect again.

Application code must handle unavailable or short-lived storage gracefully. Do not use private-mode detection as an authentication or security feature, and do not promise users anonymity the browser cannot provide.

Check your understanding: storage and security

Check cookies, localStorage, sessionStorage, IndexedDB, origins, cross-origin access, private sessions, sandboxing, and site isolation.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Performance and DevTools

Web Vitals, long tasks, layout work, profiling, and leaks.

Measure before optimizing

Start performance work from a user-visible problem and a recording rather than guessing from code alone.

Performance work starts with a user-visible problem, not a list of fashionable optimizations.

"The page is slow" might mean several different things:

- the main content appears late
- a click takes too long to produce a frame
- content moves while someone is reading
- scrolling or animation misses frames
- memory grows after a feature is closed

Each symptom needs different evidence. Network timing can explain a late resource, while a main-thread recording can explain blocked input. A heap snapshot will not diagnose a slow server response.

Build a repeatable baseline:

1. Write down the action and visible symptom.
2. Use a fresh load or a warm load intentionally; do not mix them.
3. Apply realistic network and CPU throttling when appropriate.
4. Record the smallest window that contains the problem.

5. Save the metric, trace, or screenshot that demonstrates it.

Then change one likely cause and record the same scenario again. If several variables change at once, an improvement does not tell you which change mattered.

Lab tools give repeatable diagnosis. Field data shows what real visitors experience across devices, networks, locations, and page states. Use field data to find important problems and lab recordings to reproduce and explain them.

As an experiment, profile the same page once with cache disabled and once with a warm cache. Label both recordings. The difference demonstrates why an unlabeled "before" measurement is weak evidence.

Core Web Vitals

Use LCP, INP, and CLS as complementary signals for loading, responsiveness, and visual stability.

Core Web Vitals describe three different parts of a visitor's experience:

- Largest Contentful Paint (LCP) reports when the largest eligible piece of content in the viewport is painted during loading.
- Interaction to Next Paint (INP) summarizes the latency of a page's interactions, from input through the next presented frame.
- Cumulative Layout Shift (CLS) scores unexpected movement of visible content.

One number cannot substitute for another. A page can show its hero quickly and still respond poorly to clicks. It can respond quickly while images shift the article under the reader.

Field and lab measurements answer different questions. Field data aggregates real visits and includes device, network, cache, geography, and behavior differences. A lab run is controlled and inspectable, making it useful for tracing a bad result to network or main-thread work.

Use the Performance panel's live metrics or a recording to connect a metric to page activity. Locate the LCP candidate, interact with the page while watching responsiveness, and inspect layout-shift entries. The metric identifies the experience; the trace supplies the explanation.

Published "good" thresholds are useful for monitoring, but do not optimize only to cross a line. Check the current Web Vitals documentation when setting targets, because definitions and tooling evolve. More importantly, validate that the page feels better and that field distributions improve, especially for slower visitors.

Investigate a slow LCP

Break the largest-content timing into server response, resource discovery, download, and rendering delay.

Do not start by compressing every image. First identify the LCP element and find where its time went.

An LCP resource such as a hero image can be delayed in four broad places:

1. The first HTML response arrives late.
2. The browser discovers or prioritizes the resource late.
3. The resource takes a long time to transfer.
4. The resource is ready, but rendering is delayed.

Each delay has a different fix. Server and connection work affect the first response. A hero URL hidden behind client-side rendering or a stylesheet is discovered later than an `<img>` in initial HTML. An oversized file increases transfer time. Long JavaScript, styles, fonts, or hidden content can delay the paint after the bytes arrive.

Use DevTools to build an evidence chain:

1. Record a fresh load in the Performance panel and select the LCP marker.
2. Identify the element and, if applicable, its resource URL.
3. Find that request in Network and inspect its start time, priority, initiator, and duration.
4. Compare the request's completion with the LCP time.

If the request starts late, improve discovery or priority. If it transfers slowly, reduce bytes or improve delivery. If it finishes early but LCP remains late, investigate main-thread and rendering work instead.

Run a small experiment by temporarily replacing the LCP image with a tiny local file. If LCP barely changes, image transfer was not the dominant cause. Revert the experiment and optimize the delay the trace actually shows.

Investigate a slow interaction

Find the input delay, handler work, and rendering delay behind an interaction that feels unresponsive.

An interaction is more than its event handler. Its visible latency can contain three broad parts:

1. Input delay: the event waits because the main thread is busy.
2. Processing time: event handlers and their synchronous work run.
3. Presentation delay: style, layout, paint, and compositing happen before the result appears.

That distinction prevents the wrong fix. Rewriting a short handler will not help when it waits behind an earlier long task. Splitting JavaScript will not help a component that triggers an enormous layout and paint.

Record the slow interaction in the Performance panel. Select its interaction marker and inspect the Main track from input to the next frame:

- Work before the handler points to input delay.
- Wide function calls inside the handler point to processing cost.
- Rendering work after the handler points to presentation delay.

Then reduce the dominant part. Remove unnecessary handler work, update less of the DOM, avoid forced synchronous layout, defer non-visible follow-up work, or split long computation so the browser can present a frame.

INP represents the page's interaction latency over a real visit, so an average handler duration can hide a rare but painful action. Test important interactions and investigate the slow end of the distribution.

Add a deliberate 300 ms loop before a button handler and record the click. Then move that unrelated work to run after the visible update and yield first. Compare the input-to-paint interval, not just the handler's own duration.

Prevent unexpected layout shifts

Reserve space and avoid late changes that move existing content while a visitor is reading or interacting.

Layout shift happens when visible content changes position after it has already been presented. The frustrating part is not movement itself; it is unexpected movement that disrupts reading or causes a visitor to click the wrong target.

Reserve geometry before late content arrives:

```
<img
  src="product.jpg"
  width="1200"
  height="800"
  alt="A red bicycle"
>
```

The browser can derive an aspect ratio from these dimensions even when CSS makes the image responsive. Apply the same idea to video, ads, embeds, and asynchronously loaded widgets.

Other common causes include:

- inserting a banner above existing content
- swapping to a font with substantially different metrics
- expanding a component without reserved space
- changing layout-related properties during an animation

Prefer transforms for purely visual motion because they do not change surrounding layout geometry. This does not mean every transform is free: large moving layers can still increase painting, raster, or memory cost.

Record the page in the Performance panel and inspect the Layout Shifts track. Select a shift to see which elements moved and what changed immediately before it. Fix the cause, not merely the element that happened to be displaced.

As an experiment, remove an image's dimensions, load with cache disabled, and record. Restore the dimensions and repeat. The comparison makes reserved space visible in both the page and trace.

Avoid layout thrashing

Batch DOM reads and writes so JavaScript does not repeatedly force synchronous layout inside a loop.

A DOM or style write can invalidate layout. If JavaScript immediately asks for current geometry, the browser may have to calculate layout synchronously before returning the value.

This loop alternates writes and reads:

```
for (const item of items) {
  item.style.width = `${containerWidth}px`
  console.log(item.offsetHeight)
}
```

Each `offsetHeight` may force pending layout work, and the next iteration invalidates it again. Repeating that pattern is layout thrashing.

Batch the phases instead:

```
const heights = items.map(item => item.offsetHeight)

items.forEach((item, index) => {
  item.style.width = `${containerWidth}px`
  item.dataset.previousHeight = heights[index]
})
```

Real code may need a different calculation, but the rule stays useful: group required reads, compute values, then group writes. Cache dimensions when they are valid rather than asking the browser repeatedly.

Do not remove every geometry read on principle. Measure a representative interaction in the Performance panel and look for repeated layout events or forced-layout warnings tied to the same function. Fixing one measured loop is more valuable than wrapping arbitrary DOM code in scheduling helpers.

Create 500 elements and run both patterns while recording. Compare time spent in layout and the number of layout events. Repeat the recording because tiny examples can be dominated by profiling noise.

Use requestAnimationFrame for visual updates

Schedule work that changes the current frame near the browser's next rendering opportunity.

`requestAnimationFrame()` asks the browser to run a callback before a future paint. It is a good place to apply a visual state that should be synchronized with rendering.

```
let start

function move(timestamp) {
  start ??= timestamp

  const elapsed = timestamp - start
  const x = Math.min(elapsed / 4, 300)

  box.style.transform = `translateX(${x}px)`

  if (x < 300) requestAnimationFrame(move)
}

requestAnimationFrame(move)
```

Use the callback timestamp rather than assuming a fixed number of frames per second. Displays have different refresh rates, frames can be delayed, and browsers usually reduce or pause callbacks in background tabs.

`requestAnimationFrame` does not make expensive work fast. A callback that runs too long still delays the frame. Keep it focused on the state needed for that paint, and move unrelated work elsewhere.

For repeated events such as scroll, store the latest input and schedule at most one pending frame:

```
let scheduled = false
```

```
window.addEventListener('scroll', () => {
  if (scheduled) return
  scheduled = true

  requestAnimationFrame(() => {
    updateHeader(window.scrollY)
    scheduled = false
  })
})
```

Record the animation with the Performance panel. Check frame timing and the duration of each callback. If frames remain late, investigate the callback and the rendering it triggers instead of adding another scheduling layer.

Debounce and throttle repeated input

Choose between waiting for a burst to finish and limiting work to a steady maximum rate.

Repeated events can arrive faster than the associated work should run. Debouncing and throttling express two different product decisions.

A debounce waits until events have stopped for a chosen delay:

```
let timer

search.addEventListener('input', () => {
  clearTimeout(timer)
  timer = setTimeout(() => runSearch(search.value), 250)
})
```

This suits an autocomplete request when intermediate values are not useful. Its tradeoff is deliberate latency: the result cannot start until the pause has elapsed.

A throttle allows work at a limited rate while events continue. It suits progress indicators or analytics that must update during activity without processing every event.

For visual scroll and pointer work, a useful variation is to save the latest state and apply it once in `requestAnimationFrame()`. This aligns updates with rendering and avoids building a backlog of obsolete positions.

Choose from the desired behavior:

- Need only the final value after a burst? Debounce.
- Need periodic updates during the burst? Throttle.
- Need the latest visual state for each frame? Schedule one frame callback.

Cancellation and cleanup are part of the design. Clear pending timers or frame callbacks when their component is destroyed, and decide whether the first or final event must always run.

Record a fast typing or scroll interaction before and after applying the chosen strategy. Confirm that handler work decreases without making the interface feel disconnected.

Overview of the Browser DevTools

An overview of browser DevTools: inspect HTML and CSS, run JavaScript, debug network requests and code, analyze performance, and inspect browser storage.

Browser DevTools are built-in tools for inspecting and debugging a web page.

Chrome, Firefox, Safari, and Edge all provide them. The panel names and keyboard shortcuts vary slightly, but the core workflow is the same.

This guide uses the names from Chrome DevTools. The official Chrome DevTools overview links to detailed guides for every panel.

Open DevTools

Right-click an element and choose **Inspect** to open DevTools with that element selected.

You can also use:

- `Command` + `Option` + `I` on macOS
- `Control` + `Shift` + `I` on Windows and Linux
- `F12` on many Windows and Linux keyboards

Use `Command` + `Option` + `J` on macOS or `Control` + `Shift` + `J` on Windows and Linux to open the Console directly.

See the official opening instructions for every method.

Elements: inspect HTML and CSS

The Elements panel shows the page's live DOM, not necessarily the original HTML sent by the server. JavaScript may have added, removed, or changed elements after the page loaded.

Select an element to inspect:

- its attributes and child elements
- the CSS rules that match it
- overridden declarations
- computed styles
- layout information such as the box model, grid, and flexbox
- event listeners and accessibility properties

You can edit HTML and CSS in place to test an idea. These edits affect only the current page in your browser and normally disappear when you reload.

Use the element picker in the DevTools toolbar to select something directly from the page.

Device mode

The device toolbar simulates different viewport sizes, touch input, and device pixel ratios.

It is useful for finding responsive layout problems, but it is not a complete replacement for testing on a real phone or tablet. Real devices have different browsers, keyboards, hardware, and operating-system behavior.

Console: inspect values and run JavaScript

The Console shows messages, warnings, and errors produced by the page.

You can also run JavaScript in the context of the current page:

```
document.querySelector('h1')?.textContent
```

Useful console methods include:

```
console.log(value)
console.table(items)
console.error(error)
console.time('load')
console.timeEnd('load')
```

Do not paste code into the Console unless you understand it. Code executed there can access the current page and anything your logged-in session can access.

Network: inspect requests

Open the Network panel and reload the page. DevTools records requests made while it is open.

For each request you can inspect:

- URL and HTTP method
- status code
- request and response headers
- cookies
- request body
- response body
- timing information
- the code that initiated the request

You can filter by resource type, search headers and responses, disable the cache, simulate a slower connection, or work offline.

The panel can export a HAR file, which is useful for sharing a network trace. Review a HAR before sharing it because it may contain cookies, authorization headers, query parameters, and response data.

The Network panel guide covers its current tools.

Sources: debug JavaScript

The Sources panel is the JavaScript debugger.

Set a breakpoint on a line, trigger the behavior you want to inspect, and execution pauses before that line runs. While paused, you can:

- step over, into, or out of functions
- inspect local and global variables
- evaluate expressions
- view the call stack
- add watch expressions
- add conditional breakpoints

You can also pause on caught or uncaught exceptions, DOM changes, event listeners, and network requests.

Source maps connect generated code back to the source files written in TypeScript or processed by

a bundler. Make sure your development build produces source maps if you want useful debugging.

Read the Sources panel documentation for the full debugger workflow.

Application: inspect storage and web app state

The Application panel groups browser-managed state and application features, including:

- cookies
- local storage and session storage
- IndexedDB
- Cache Storage
- service workers
- manifests

It can inspect, edit, and clear stored data for the current site. This is useful when an application behaves differently because of stale cache data, an old service worker, or an unexpected stored value.

Storage can contain private data and authentication tokens. Be careful when taking screenshots or sharing exported information.

See the Application panel overview.

Performance and memory

Use the Performance panel to record what the browser does while a page loads or while you interact with it.

The recording can reveal:

- long JavaScript tasks
- excessive rendering work
- layout shifts
- slow event handlers
- animation bottlenecks

Use the Memory panel when you suspect memory leaks or excessive allocation.

Performance recordings are detailed. Start with one specific slow interaction, record only that interaction, and inspect the busiest part of the timeline.

Lighthouse

The Lighthouse panel audits a page for areas such as performance, accessibility, best practices, and SEO.

Treat its report as a starting point, not a guarantee that the page is fast or accessible for every user. Run audits in a clean browser profile when extensions or local state might affect the result.

See the Lighthouse overview.

Changes do not automatically update your source files

Most changes made in DevTools are temporary. Reloading the page restores the files served by your application.

The Changes panel can show edits you made, and local overrides or a configured workspace can

persist some changes. For normal development, reproduce the useful edit in your source code and verify it through your application's build process.

Read a performance recording

Use the network timeline, frames, main-thread flame chart, and event details to connect a visible delay to browser work.

A performance recording is a timeline of evidence. Start with a narrow question such as "why did this click take 400 ms to update the page?" A recording of several unrelated minutes makes that answer harder to find.

Capture a short window around the load or interaction, then read from the visible symptom toward its cause:

1. Use screenshots and frame markers to locate when the page visibly changed.
2. Use interaction and Web Vitals markers to locate the relevant event.
3. Inspect the Main track for tasks between input and paint.
4. Expand wide entries to find JavaScript, style, layout, or paint work.
5. Follow Network requests and initiators when the main thread is waiting for bytes.

In the flame chart, width represents duration and vertical stacking represents calls. A wide parent may be expensive because of its children, so distinguish total time from self time. Bottom-up views help group the functions that accumulated the most time; call-tree views preserve how execution reached them.

Look for patterns, not just one red marker:

- one long task delaying an interaction
- repeated layout events caused by alternating DOM reads and writes
- a large paint after a small visual change
- an idle gap ending when a late network request completes
- script evaluation occupying the loading path

Apply CPU or network throttling only when it represents the affected visitors, and record the setting with your result. Throttling is a model, not a perfect copy of a physical device.

Before changing code, write one sentence that links evidence to cause: "The click waited behind a 280 ms task created by `renderResults()`." After the fix, reproduce the same action and show that this specific interval changed.

Find a memory leak

Use heap snapshots and allocation tools to find objects that remain reachable after their feature is closed.

A useful leak test repeats a lifecycle that should return to roughly the same state.

Suppose opening a dialog creates DOM nodes, listeners, and a large data model. Use this procedure:

1. Load the page and take a baseline heap snapshot.
2. Open and close the dialog several times.
3. Allow garbage collection to occur; a DevTools collection button can make snapshot comparison more repeatable, but application code cannot rely on it.
4. Take another snapshot and compare object groups.

5. Repeat the cycle to see whether the same groups keep accumulating.

A larger heap alone is not proof. The browser may retain capacity, the application may populate an intentional cache, and collection timing varies. Look for instances whose feature has been closed but that remain reachable after repeated cycles.

Detached DOM nodes are a clue, not the root cause. Select an unexpected object and follow its retaining path. The path can reveal an event listener on `window`, an active timer, a closure, or a cache entry that still references the object.

Fix ownership at the lifecycle boundary:

- remove listeners or register them with an abort signal
- disconnect observers
- cancel timers and animation frames
- unsubscribe from data sources
- limit and expire cache entries
- release references to detached trees

Run the same cycle after the fix. The strongest evidence is that the unwanted instance count no longer grows while the feature still behaves correctly.

Profile and fix a page

Use everything in the course to explain and improve one page from request start through interaction and cleanup.

This final exercise turns the browser pipeline into one connected explanation. Choose a page with initial HTML, CSS, an important image, JavaScript, an interaction, and some browser storage.

Start with three explicit scenarios:

1. A fresh navigation until the main content appears.
2. One important interaction until its visual result appears.
3. Opening and closing a feature that allocates memory.

For the load, use Network and Performance recordings to answer:

- When did the document request start and finish?
- Which resources did the parser, CSS, or JavaScript discover?
- What became the DOM, CSSOM, and render tree?
- Where did style, layout, paint, raster, and compositing work occur?
- Which element became LCP, and what delayed it?

For the interaction, trace input delay, handler work, microtasks, and rendering before the next frame. Look for a long task, unnecessary DOM work, forced layout, or excessive paint. For memory, repeat the feature lifecycle and follow retaining paths for objects that should have disappeared.

Inspect the Application panel too. Identify why each cookie, `localStorage` entry, or IndexedDB database exists, who owns it, and when it expires or is cleared.

Choose fixes from evidence. Examples include exposing the hero resource in initial HTML, removing unused synchronous work, batching layout reads and writes, reserving image dimensions, or cleaning up a listener when its component closes. Do not apply all of them by default.

Keep a small before-and-after record for each chosen fix:

```
Symptom:
Evidence:
Cause:
Change:
Result under the same test conditions:
Tradeoff or remaining risk:
```

Record the scenarios again under the same cache, network, CPU, and viewport conditions. A successful result is not "the score increased." It is a defensible explanation from request to pixels to interaction to cleanup, supported by traces that show the measured bottleneck changed.

Check your understanding: performance and DevTools

Check LCP, INP, CLS, long tasks, layout thrashing, animation frames, debounce and throttle, flame charts, memory tools, and frame budgets.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/browser-internals/.